Maria Eichlseder

Linear Propagation of Information in Differential Collision Attacks

Master's Thesis

Graz University of Technology

Institute for Applied Information Processing and Communications
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch

Advisor: Dipl.-Ing. Dr.techn. Martin Schläffer
Assessor: Dipl.-Ing. Dr.techn. Florian Mendel

Graz, March 2013

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Differential collision attacks on hash functions often use a guess-and-determine strategy to find conforming message pairs for differential characteristics. Such characteristics are commonly described in terms of generalized conditions. To determine consequences of a guess and detect contradictions, this strategy requires a propagation method. Previous publications use bitsliced propagation for this purpose, which results in very local propagation. In this thesis, we propose linear propagation as an alternative, more global approach for the propagation of information. It is based on solving linear equations using Gauss-Jordan elimination. If only linear conditions and linear functions are involved, this method achieves perfect propagation. We describe the mathematical foundations for both the basic method and extension for nonlinear settings. In particular, linear propagation can be combined with bitsliced propagation for an efficient hybrid method. We also discuss the practical implementation and integration into an existing automated search tool. Evaluation results obtained from this tool show that the linear method provides better propagation quality than bitslicing in most applications. It performs especially well for large, complex linear functions such as the linear layer of the SHA-3 hash standard. While current implementations of our new method are slightly slower than bitsliced propagation, its improved contradiction detection capability can curtail the search process and reduce the overall runtime.

Key words: hash function, collision, guess-and-determine attack, generalized conditions, linear propagation, algebraic attack

Acknowledgements

I owe several contributors to this thesis my thanks.

First, to my advisor Martin Schläffer, for his guidance, help, and constructive reviews of draft versions of this document.

Second, to the IAIK Krypto group, and in particular to Florian Mendel, Tomislav Nad and Vincent Rijmen, for enlightening discussions and valuable comments.

Third, to my parents Eva and Helmut, for their confidence, patience, and warm meals in times of need.

Fourth, to my other half, Daniel Gruß, for his support, and scaring away bugs by mere presence.

Fifth, and last, but least, to my cat Míriël, for all too frequent distraction and various creative, albeit somewhat dyslexic contributions to this thesis, I hope none of which survived until the final version.

Maria Eichlseder

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

This thesis discusses a new linear propagation method for differential guess-and-determine attacks on cryptographic hash functions that was first proposed in [19]. The goal of this attack is to find collisions for modern hash functions. Hash functions are a cryptographic primitive that is necessary for a wide range of security applications and protocols. They are typically employed when properties similar to a real-world fingerprint are desired: a small piece of information, the fingerprint, is stored and used as a unique representative for a much larger and less handy entity, a human individual. If the person is present, it is easy to verify that a stored fingerprint was indeed produced by her. However, given only a fingerprint, it is extremely difficult to find any person with exactly this fingerprint. In case of cryptographic hash functions, the digital fingerprints serve as representatives of larger data items, such as documents or contracts. They are for example useful in digital signature schemes, where a signature of the fingerprint is informationally equivalent to a signature of a complete document. At the same time, it is much easier to compute and process.

Like for their real-world namesake, finding two documents with the same digital fingerprint would thwart the system's security goals. It would be impossible to prove later whether a digital signature belonged to the one or the other document, just like fingerprint analysis becomes useless if two suspects of a crime have identical fingerprints. Such duplicates are referred to as collisions for cryptographic hash functions.

In the past decade, a number of successful collision attacks on the most popular family of hash functions re-awoke the research interest in hash functions. The research focused both on extending and improving the known attacks, and on designing new hash functions resistant to these attacks. In particular, an official new standard, SHA-3, was published. Naturally, it was defined with these recent attacks in mind, and are no longer as susceptible as their predecessors. This motivates the adaptation and development of

new attack methods. For example, SHA-3 defines a huge internal state, which makes some previous attempts infeasible. On the other hand, large parts of each step are connected by a linear layer. We discuss an extension to previous attacks that works especially well for the linear parts of hash functions.

We use our method for differential guess-and-determine attacks. These attacks observe how differences between two inputs to the hash function behave throughout the evaluation of the function. Then, these attacks attempt to guess input pairs that form a collision. We describe how input changes are processed by the function as propagation of these changes. This was previously sometimes modeled using bitsliced propagation. However, while bitsliced propagation is relatively easy to compute, it is not sufficient to effectively find collisions for recent hash functions. Instead of (or supplementary to) bitsliced propagation, we propose a linear propagation method, based on relatively simple methods from linear algebra.

A similar idea of linear propagation was used for the linear message expansion of SHA-1 in the implementation of [15] by De Cannière and Rechberger. The task of this thesis was to elaborate on this initial idea and provide a first practical implementation. We formalized the idea and proved the usefulness of Gaussian elimination for propagation. Ideas for incorporating nonlinear conditions were investigated, but turned out to be too inefficient. A prototype version for comparison with existing propagation methods was implemented. Later, an implementation for an existing attack tool was extended and improved. Statistical evaluation methods were developed and applied to evaluate the new method, focusing on the architecture of SHA-2 and, to a lesser extent, SHA-3. The results of this thesis were also published in [19].

1.2 Related work

Differential attacks in cryptanalysis were first proposed for block ciphers by Biham and Shamir in 1993 [6]. Applications to hash functions date back to 1995 by Dobbertin and others [16–18]. Early attacks, such as Chabaud and Joux' [9] and its extension to SHA-1 by Oswald and Rijmen [38], already define linear or equation-based models of hash functions, but use them for slightly different purposes. The practical breakthrough application of differential concepts to MD5 is due to Wang nearly a decade later [45].

Our work is based on a different model of message differences than Wang's, namely on the generalized conditions introduced by De Cannière and Rechberger [15]. These generalized conditions were used by Mendel et al. for automated guess-and-determine attacks on SHA-256 [30, 32]. We use the same attack structure, but instead of bitsliced propagation, we employ linear propagation. This new propagation method was first proposed in [19], and in particular improves the detection of inconsistencies in guesses made during the search. Other authors similarly extended the propagation method in De Cannière and Rechberger's attack strategy. For example, Leurent defined "1.5 bit"

conditions to more accurately and more globally describe and propagate message differences [27]. His refined description also uncovers inconsistencies that could not be discovered with previously existing methods.

Algebraic methods have also been applied in this context by various authors, such as Courtois [11]. Among the most popular approaches are Gröbner basis algorithms to solve some of the nonlinear equations arising during the analysis [1,2]. While these algorithms, such as the original Buchberger algorithm or the F_5 algorithm [20], are more powerful and in particular not limited to linear parts of the hash function like our approach, they are also much more expensive to calculate. This makes them less attractive for repetitive propagation tasks. Another related technique is the use of SAT-solvers. For this purpose, the equations are translated to conjunctive normal form to benefit from the efficient solving techniques of modern existing SAT-solvers [28].

1.3 Overview

The remainder of this thesis is organized as follows. Chapter 2 formally defines hash functions and gives application examples. Different important hash function architectures are briefly described, followed by a more detailed definition of two recent hash functions that serve as application examples in this thesis, SHA-2 and SHA-3. General classes and properties of attacks on hash functions are also explained.

Chapter 3 introduces the concepts of differential attacks on hash functions. Influential historical attacks and difference models are briefly sketched. We then define generalized differences and the concept of difference propagation. A higher-level overview of our guess-and-determine attack strategy is also given. The chapter concludes with existing propagation methods.

Chapter 4 is dedicated to our new, linear propagation method. The mathematical foundations and basic algorithms are defined. We also discuss generalizations and improvements of the method, for example for nonlinear environments.

Chapter 5 is the practical counterpart to the previous, formal chapter. We discuss our practical implementations of the linear propagation method. The integration into an existing framework for guess-and-determine attacks is described, as well as useful algorithms and data structures for the matrix operations. Finally, the chapter contains practical considerations on the improvements and extensions partly already proposed in the previous chapter.

Chapter 6 shows statistical and graphical evaluations of our implementations' performance. The results are compared to existing methods, and the usefulness of the previously described extensions is discussed. Finally, we conclude with a summary of our findings and some ideas for future work.

Chapter 2

Hash functions

In this chapter, we first motivate the need for hash functions. Section 2.1 lists a number of typical applications that require hash functions with certain security properties. These properties are defined and discussed in more detail in the following Section 2.2. Afterwards, important construction principles and typical building blocks for hash functions are presented in Section 2.3. Two recent hash functions, SHA-2 and Keccak, both of which were attacked for this thesis, are described in more detail in Section 2.4. Finally, we briefly discuss some typical classes of attacks on hash functions in Section 2.5.

2.1 Motivation

Cryptographic hash functions are one of the essential building blocks of cryptographic applications and protocols. They are used to achieve data integrity, message authentication and similar central goals in information security. Typically, they are employed when a large message needs to be represented by a smaller fingerprint. This fingerprint is used to identify the input message, but it should not be possible to re-derive the original input message from just knowing the fingerprint. Thus, cryptographic hash functions should be easy to compute, but hard to invert.

In general, hash functions accept input messages of arbitrary length and map them to output values of a fixed, small length. This output value is usually referred to as hash value, checksum, fingerprint or message digest of the input message. To qualify as a cryptographic hash function (as opposed to conventional hash functions such as used for hash-table lookup), several additional properties are required by cryptographic applications.

To motivate these properties, consider the following typical applications for cryptographic hash functions:

- **Data identification:** The hash value can serve as a compact fingerprint and identifier for a large file. For example, some version control systems for source code such as Git identify file content or commit versions by their SHA-1 digest. Similarly, peer-to-peer filesharing networks and other distributed file management applications often use hash values to identify files in their distributed data management structures, such as location lookup tables.

- **Message integrity:** Since the hash value identifies the exact content of a file or message, computing it before and after a message transmission and comparing the two verifies that the file has not been changed. For this reason, large download files such as Linux distributions are often published including an MD5 or SHA-1 hash value. Such detected changes may be accidental, as in file download errors; or manipulatory, for example when an adversary tries to distribute a modified executable with some additional malicious code.

- **Digital signatures:** Digital signature schemes such as DSA or ECDSA are used to prove the authenticity and integrity of a message by applying methods from public cryptography. Instead of signing the whole message (which would produce a very long signature), only the hash value of the message is signed.

- **Password verification:** For security reasons, it is not advisable to store plain passwords in password-based authentication systems. Many authentication systems store not the password itself, but its hash value. On each authentication attempt, the entered password is hashed and if the resulting hash equals the stored hash, the login is accepted. For example, the Linux `/etc/shadow` file contains DES-based (and salted) hash values of all users' passwords. Thus, the system administrator can change, but not read users' passwords; neither can the information be abused in case the file leaks. This is especially relevant since many users tend to reuse their login names and passwords on other systems.

- **Confirmation of knowledge:** To confirm knowledge without telling it, its hash value can be shared, for example in challenge-response-protocols.

- **Commitment:** Similarly, to commit to specific information without telling it until a later point, the corresponding hash value can be published. For example, in a bet or game, participants may want to keep their choice secret until a later time when the bet or game move is evaluated. So first, the participants publish only their choices' hash values. Later, when the bet is evaluated, the choice can be revealed and must be a preimage of the committed hash value. If the set of choices is small (and an adversary can simply hash all possible choices and compare), appending a random nonce may help. Similar applications also arise in zero-knowledge protocols.

- **Key derivation:** To derive temporary keys from a master key, hashes of the master key and an appended counter or similar can be used. If one temporary key is compromised, neither other keys nor the master key are affected.

- **Random number generation:** In a similar way, a seed with increasing counter can be repeatedly hashed to generate a pseudorandom number stream.

Intuitively, these applications require that hash values are easy to calculate, but inverse problems like finding a message for a given hash or constructing two messages with equal hash values are hard. Such hash functions will also appear relatively "random" and complex since input message patterns should not lead to predictable hash value patterns.

2.2 Definitions

In this section, we define requirements and properties of cryptographic hash functions as motivated by applications like those in Section 2.1. The general model and terminology here and in the next section closely follow that of [33] and [36].

The basic property of hash functions is that these functions map data of arbitrary (finite) length to values of a fixed, small length n.

Definition (Hash function)**.** An n-bit hash function is a function h mapping binary strings (messages) of arbitrary length to binary strings of n bits (hash values), i.e. $h : \{0,1\}^* \to \{0,1\}^n$ (compression). Given $x \in \{0,1\}^*$, it must be easy to compute $h(x)$.

The general concept of hash functions is not restricted to cryptographic contexts, but originates from the study of data structures and algorithms. Such conventional hash functions find applications for example in hash tables, where a hash function is used to assign a table index or key to any data item for storing and finding it in the hash table. Knuth discusses applications and properties of non-cryptographic hash functions extensively in [24, p. 506-549] and dates their first uses to the early 1950s at IBM.

According to [33, p. 376], Wegman and Carter were among the first to note the cryptographic relevance of hash functions in the 1970s in [46]. They combined hash functions with secret keys for a message authentication system. Such keyed hash functions are one of the two large families of cryptographic hash functions; the second family consists of unkeyed hash functions, but with additional security-related requirements compared to conventional hash functions.

By far the most cryptographically relevant types of unkeyed and keyed hash functions are MDCs and MACs, respectively. Some sources, such as [36], even use the terms "keyed hash function" and "MAC" (or "unkeyed" and "MDC") synonymously.

Definition (Modification detection code, MDC)**.** An MDC is a hash function h whose definition is publicly known, and which does not require any secret information for computation.

Additional required properties of MDCs depend on the application and are explored in more detail later in this section.

Definition (Message authentication code, MAC)**.** A MAC is a family of hash functions (h_k), defined for a set of keys k, whose definition is publicly known and whose operation depends on no secret information except for the key k. The family of hash functions must be computation resistant, i.e. given any number of message-key-pairs $(x_i, h_k(x_i))$, it should be infeasible to compute $h_k(x)$ for any $x \neq x_i$ or determine k, even if the messages x_i are chosen by the attacker.

Although the term "code" is not very descriptive for hash functions, and "modification detection" is not the only purpose of MDCs (for example, they can also be used for message authentication!), these are the most common terms to distinguish these important types of hash functions for historical reasons. Occasionally, MDCs are also referred to as message integrity codes (MICs). Most applications listed in Section 2.1 require unkeyed hash functions, i.e. MDCs.

Rabin [37] used hash functions in a first public key-based signature scheme and noted additional requirements for hash functions used in this context. In more recent applications, hash functions are still categorized based on these essential properties.

Definition (Preimage, 2nd preimage and collision resistance)**.** Let $h : X \to Y$ be a hash function.

- h is preimage resistant if it is computationally infeasible to find any input message for a specified hash value, i.e. for essentially all $y \in Y$, given y, it is infeasible to find any $x \in X$ such that $h(x) = y$. Such a function h is also called a one-way function.
- h is 2nd-preimage resistant if it is infeasible to find any second input message with the same hash value as a given input message, i.e. for essentially all $x \in X$, it is infeasible to find any $x' \in X$, $x' \neq x$, such that $h(x') = h(x)$. This is alternatively referred to as weak collision resistance.
- h is collision resistant if it is infeasible to find any two input message with the same hash value, i.e. it is infeasible to find any $x, x' \in X$ such that $h(x') = h(x)$. This is also called strong collision resistance.

The first requirement is perhaps most clearly motivated in applications like password verification, where not the password itself, but its hash value is stored in a database; any password hashing to the stored hash value is accepted. If the hash function is not preimage resistant, an attacker Eve with access to the database may be able to find some password hashing to a hash value stored for a user Bob in this database. This may not be equal to Bob's original password, but Eve will be able to authenticate herself as Bob nevertheless. If Bob reused his password on other platforms employing the same hash functions for password verification, Eve may even be able to impersonate Bob on completely unrelated systems.

In this example, however, the other two requirements are not essential to the system's security - two passwords with the same hash value are in no way more useful than one.

To motivate these additional requirements, consider the application of digital signatures like suggested by Rabin, where the hash value of a message is signed by Alice. By her signature, Alice wants to confirm her assent on the message, such as a contract. A dishonest Eve with access to a contract signed by Alice might want to set up a different contract. If the used hash function is not 2nd-preimage resistant, Eve may come up with an alternative contract resulting in the same hash value as the original contract, attach a copy of the original signature to it and claim that Alice signed this modified contract.

In a slightly modified setting with a hash function offering no collision resistance, assume Eve is able to persuade Alice to sign at least harmless contracts. Eve may then prepare two different contracts with colliding hash values, a nice and a nasty contract (from Alice' perspective). Eve will let Alice sign the nice contract, then attach the obtained signature to the nasty contract and again claim that Alice signed it.

While such attack scenarios may sound contrived, especially collision attacks have been successfully launched at real-world applications employing outdated hash functions like MD5. As an example, consider the attack by Stevens et al. [41] who constructed valid certificates from several international certification authorities granting them excessive rights. They achieved this by obtaining valid, harmless certificates constructed to collide with the rogue certificate and copying the signature.

Based on whether 2nd-preimage and collision resistance apply or not, MDCs are further divided into categories. OWHFs were defined by Merkle in [34] in the late 1970s, CRHFs a decade later by Damgård in [13].

Definition (One-way hash function, OWHF)**.** A one-way hash function (or weak one-way hash function) is a hash function that is additionally preimage resistant and 2nd-preimage resistant.

Definition (Collision resistant hash function, CRHF)**.** A collision resistant hash function (or strong one-way hash function) is a hash function that is additionally 2nd-preimage and collision resistant.

Note that preimage resistance is not explicitly required for CRHFs, and is not implied by the other two requirements, either. However, in practice, CRHFs are almost always preimage resistant.

Since the target of this thesis is to find collisions for computable functions, we are only interested in MDCs. While the results could be relatively easily transferred to MACs with known keys, collision resistance is not a required property for MACs with known keys and thus finding collisions may be easier without applying any such very general methods. For MACs whose keys are not known to the adversary, methods similar to those presented in the following chapters could possibly be applied to retrieve this key or maybe forge hashes, much like differential cryptanalysis in general is applied to block ciphers. However, such adaptations are beyond the scope of this thesis.

Beginning with the next section, the term "hash function" is used to refer only to MDCs,

or more specifically to CRHFs, since finding collisions is no issue for OWHFs.

2.3 Construction methods

While hash functions typically accept inputs of arbitrary length, they are usually constructed using a compression function f for inputs of a fixed block length. The original input is first preprocessed and split into blocks. These blocks are then iteratively compressed by the compression function, using previous results as chaining variables. The hash value is then calculated by postprocessing the result of the last iteration.

If the preprocessed input x is divided into blocks $x = x_1 \| x_2 \| \cdots \| x_t$, then the general operation of most hash functions h can be summarized as follows:

$$H_0 = \mathrm{IV}, \qquad H_i = f(H_{i-1}, x_i) \quad \text{for } 1 \leq i \leq t, \qquad h(x) = g(H_t).$$

Here, IV denotes a (usually fixed) initial value, H_i is the w-bit output after the ith iteration, f is the compression function receiving a chaining variable and input, and g is a postprocessing function that maps the internal w-bit state to the final n-bit hash value.

Figures 2.1 and 2.2 illustrate this general model; first using the graphical notation of [39], and second in a more compact loop representation.

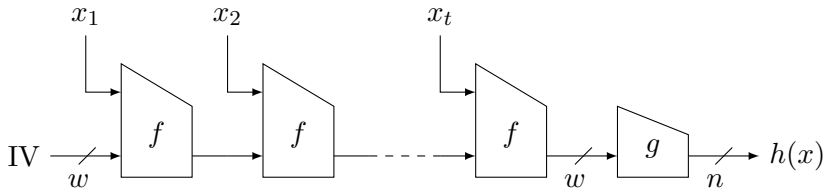


Figure 2.1: General model for hash functions.

The obvious task of the preprocessing step is to split the original input into blocks of adequate size for f and pad the input to a multiple of this block length if necessary. A simple unambiguous padding scheme is to append a 1 bit to the original message and then add 0 bits until the new length is a multiple of the compression function's block size. Additionally, the length of the original message is often appended in the preprocessing step. This ensures that no encoded input is postfix to a different encoded message and is also referred to as Merkle-Damgård strengthening.

The postprocessing step, on the other hand, is often just an identity mapping or a simple truncation of the longer internal state.

The compression function f is the core algorithm of the hash function h, and the security properties – in particular the collision resistance – of h depend heavily on those of f. This motivates attacks on the compression function f (or its building blocks).

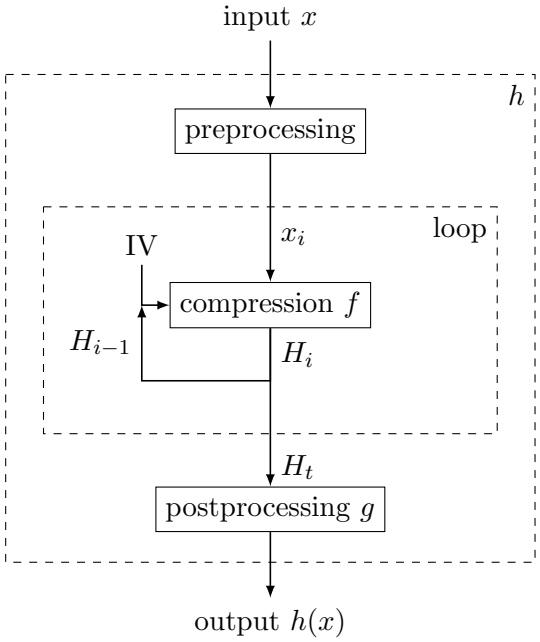


Figure 2.2: General model for hash functions, in compact representation.

2.3.1 Merkle-Damgård construction

Merkle's meta-method for hashing [34], also referred to as the Merkle-Damgård construction, allows to extend collision resistant compression functions f to collision resistant hash functions h. The construction is based on the general model described above, but further specifies some details. Most notable, it requires a suitable preprocessing step, typically by applying Merkle-Damgård strengthening. If the compression function reduces the bitsize by r bits (i.e., maps $n+r$-bit inputs to n-bit outputs), the preprocessing step splits the hash input into r-bit blocks and in each iteration, a concatenation of last iteration's chaining value and the next input block is fed to the compression function to produce the next chaining value. Many popular hash functions, including MD5, SHA-1 and SHA-2, are based on this construction method.

2.3.2 Sponge construction

Another general construction method, notably employed in the SHA-3 competition winner Keccak, is the sponge construction [3]. Originally described as a generalization of hash functions, sponge functions iteratively use a central transformation f to map variable-length input to output of arbitrary length. Like the compression function for Merkle-Damgård constructions, this transformation f is the core of the hash function and determines its security properties. It is defined for fixed-length inputs, and h ex-

tends f to variable-length inputs by iterating. However, unlike the Merkle-Damgård core functions, the sponge core transformations themselves do not compress, but are typically bijections applied to an internal state of h. The compression effect required by the general model is caused by the exclusive-or combination of internal state and message block before each iteration of f.

In sponge terminology, the main loop of applying the core transformation in the general model above is referred to as the absorbing phase where the sponge construction absorbs the input. The postprocessing step is called the squeezing phase since the hash output is squeezed from the (large) internal state, again involving the same core transformation as the absorbing step.

Throughout both phases, the input and output state of f is divided into two parts, the c-bit inner and r-bit outer part. During the absorbing phase, the input message blocks are XORed to the outer part before feeding the complete state to the next iteration of f. In the squeezing phase, the core transformation f is again repeatedly applied to the state and after each iteration, the outer state is appended to the hash output until the required output length is reached. In case of SHA-3, one outer part is already larger than the required hash output length, so the squeezing phase does not need to iterate f. Figures 2.3 and 2.4 illustrate the general sponge construction using notation similar to Figures 2.1 and 2.2.

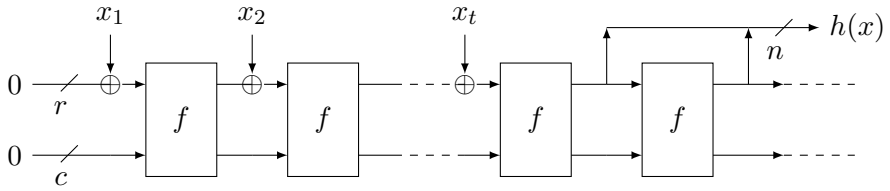


Figure 2.3: Sponge construction for hash functions.

2.3.3 Compression functions

The compression function is the core of any cryptographic hash function, and its security is essential for the whole function. There are several broad categories of compression functions, including functions based on block ciphers, on modular arithmetic and completely customized schemes. While the first two have the advantage that possibly existing implementations or modules can be reused, the latter, dedicated variations are often faster. The distinction between different categories is not always clear; for example, a compression function may reuse building blocks of a block cipher (e.g. AES), but not the encryption routine as a whole, to create a new, dedicated scheme.

There are a number of general constructions for compression functions based on a block cipher encryption routine E_K, such as the Matyas-Meyer-Oseas hash [29], the Davies-Meyer hash [29] and the Miyaguchi-Preneel hash [36], all illustrated in Figure 2.5.

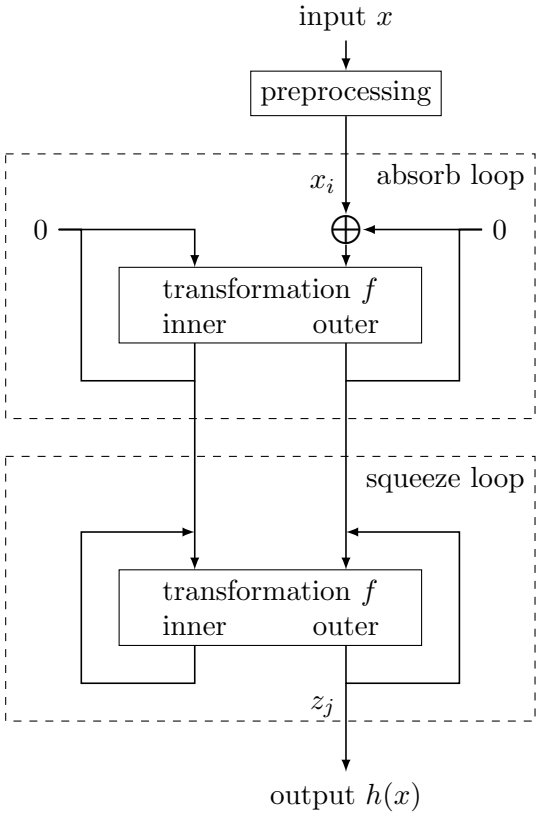
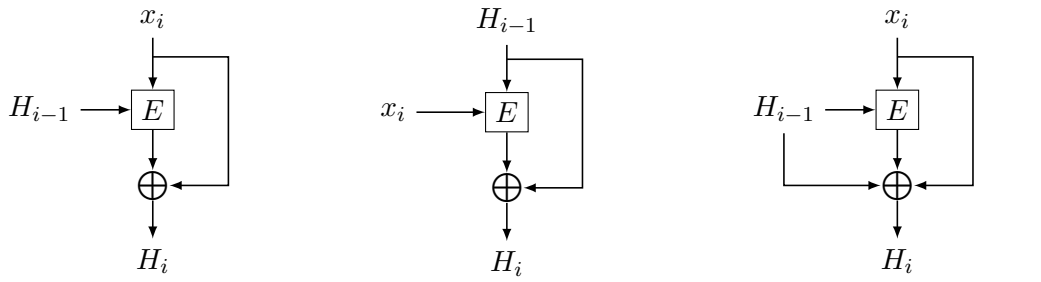


Figure 2.4: Sponge construction for hash functions, in compact representation.

The classical hash functions of the MD4-family such as MD5 and SHA-1 are examples of dedicated hash functions, although they do involve a Davies-Meyer-like construction for their compression function.



(a) Matyas-Meyer-Oseas (b) Davies-Meyer (c) Miyaguchi-Preneel

Figure 2.5: Construction schemes for compression functions, based on block ciphers.

2.4 Popular hash functions

Three of the most well-known and widespread cryptographic hash functions, namely MD5, SHA-1 and RIPEMD-160, date back to the first half of the 1990s. All three are still widely used even in critical applications, although MD5 is considered completely broken and collisions can be found in a matter of seconds [47], and SHA-1 is not considered recommendable anymore by NIST.

Newer hash functions include SHA-2, which reuses design ideas of SHA-1 and extends them to higher bitsizes, and SHA-3 (Keccak), which was selected in a competition in 2012 from a large number of candidates [10]. Both SHA-2 and SHA-3 served as example targets for the attack presented in this thesis (see Chapter 6), so their structure will be described in more detail below.

2.4.1 SHA-2

Like SHA-1, SHA-2 [21] is based on the Merkle-Damgård construction described in Section 2.3. The standard defines variants for hash outputs of 224, 256, 384 and 512 bits, differing in their word sizes and number of rounds, but otherwise nearly identical (except for some constants involved in the computation).

Depending on the desired output bitsize, SHA-2's compression function f accepts an input block size of 512 or 1024 bits. It repeats the same basic step either 64 or 80 times. Besides word permutations and sums, this step involves six word-level helper functions;

in case of SHA-256, they are defined as

$$\begin{aligned}
\sigma_0(X) &= (X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3) \\
\sigma_1(X) &= (X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10) \\
\Sigma_0(X) &= (X \ggg 2) \oplus (X \ggg 13) \oplus (X \ggg 22) \\
\Sigma_1(X) &= (X \ggg 6) \oplus (X \ggg 11) \oplus (X \ggg 25) \\
f_0(X,Y,Z) &= (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z) \\
f_1(X,Y,Z) &= (X \wedge Y) \oplus (\neg X \wedge Z),
\end{aligned}$$

where $\oplus$ denotes the exclusive-or operation, $\ggg$ rotation to the right and $\gg$ the right shift operation. For other bitsizes, the shift constants differ. The σ_i, Σ_i functions are linear, the f_i functions nonlinear.

The σ_i functions are used to expand the words M_j of the input message block to get one word W_i for each round; for SHA-256, the expansion rule is

$$W_i = \begin{cases} M_i & i \in \{0, \ldots, 15\}, \\ \sigma_1(W_{i-2}) \boxplus W_{i-7} \boxplus \sigma_0(W_{i-15}) \boxplus W_{i-16} & i \in \{16, \ldots, 63\}. \end{cases}$$

Additionally, the standard defines round constants K_i. Figure 2.6 shows one step of the compression function f, where A_i to H_i are the words of the internal state of f in round i; $\boxplus$ denotes modular addition of words. The state words are updated as follows:

$$\begin{aligned}
A_{i+1} &= \Sigma_0(A_i) \boxplus f_0(A_i, B_i, C_i) \boxplus \Sigma_1(E_i) \boxplus f_1(E_i, F_i, G_i) \boxplus H_i \boxplus K_i \boxplus W_i, \\
B_{i+1} &= A_i, \\
C_{i+1} &= B_i, \\
D_{i+1} &= C_i, \\
E_{i+1} &= D_i \boxplus \Sigma_1(E_i) \boxplus f_1(E_i, F_i, G_i) \boxplus H_i \boxplus K_i \boxplus W_i, \\
F_{i+1} &= E_i, \\
G_{i+1} &= F_i, \\
H_{i+1} &= G_i.
\end{aligned}$$

After 64 rounds, an additional output transformation is applied to get the chaining variable of the main loop in the Merkle-Damgård construction.

2.4.2 SHA-3

The winner of the SHA-3 competition, Keccak [10], is based on the sponge construction introduced in Section 2.3. Its central transformation f operates on a 5×5 array of 2^ℓ bit words (here called lanes), for any ℓ. For the SHA-3 standard, $\ell = 6$ is defined, i.e. the lane size is 64 bits, and the internal state size is 1600 bits. The block size (or, in sponge terminology, rate r) of SHA-3 depends on the desired hash output size. This is

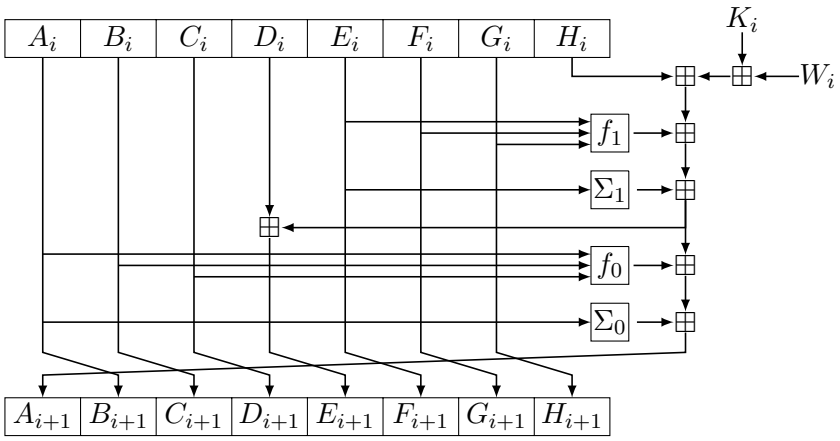


Figure 2.6: SHA-2, one step of the compression function.

necessary since larger hash outputs require larger capacity $c = 1600 - r$ to ensure the security level expected for the desired output size. Thus, larger hash output sizes imply smaller message block sizes. The SHA-3 standard defines $c = 2n$, where n is the hash output bitsize. For instance, for a hash output size of $n = 256$ bits, SHA-3 has capacity $c = 512$ and rate or block size $r = 1088$ bits.

Preprocessing consists of filling the message up to a multiple of the block size by appending a 1 bit, the required number of 0 bits and a final 1 bit. Since $n < r$ for reasonable n as defined in the SHA-3 standard, no additional applications of the core transformation are necessary in the squeezing phase, and the postprocessing reduces to truncating the internal state to n bits.

The core transformation f itself repeats a round of five steps $12 + 2\ell$ times, i.e. for the standard bitsize, the transformation consists of 24 rounds. Each round R contains the following steps, $R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$, operating on the 5×5 matrix $a[i][j]$ of internal state lanes:

- θ: Exclusive-or summation between neighboring lanes,

$$a[i][j] := a[i][j] \oplus \bigoplus_{k=0}^{4} a[i-1][k] \oplus \bigoplus_{k=0}^{4} (a[i+1][k] \ggg 1) \qquad \forall\, i, j.$$

- ρ: Bitwise rotation of each lane (except $a[0][0]$) by different offsets,

$$a[i][j] := a[i][j] \ggg \frac{(t+1)(t+2)}{2} \quad \text{for } \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}^t \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad t = 0, \ldots, 23.$$

- π: Permutation of the state lanes, defined by

$$a[j][2i + 3j] := a[i][j] \qquad \forall\, i, j.$$

- χ: Nonlinear boolean operation across each row,

$$a[i][j] := a[i][j] \oplus (\neg a[i][j+1] \wedge a[i][j+2]) \qquad \forall\, i, j.$$

- ι: Xor of the round constant for round m to one lane,

$$a[0][0] := a[0][0] \oplus K_m.$$

Note that χ is the only nonlinear part of this definition (and of the whole hash function). This means that relatively large parts of the definition can be summarized as a linear operation, which is beneficial for the linear attack we describe in Chapter 4.

2.5 Attacks on hash functions

Attacks on hash functions typically aim to thwart one of the three basic security properties of hash functions: preimage, 2nd preimage and collision resistance. Of these, collision resistance is very often subject to attacks since its a priori computational complexity ranges lowest. This a priori complexity is defined by generic attacks and relative to the hash output size. It sets a security threshold for the hash function, and any attack with lower complexity than this threshold is considered a weakness of the particular hash function, whether it is practically feasible or not. The complexity is usually measured in the expected number of hash function evaluations.

The generic attacks imply the following security thresholds for hash functions with n-bit hash results:

- Preimage: 2^n, by trying random inputs until the desired output is found. Under the assumption that all outputs are equally likely for the random inputs (or more precisely, that the hash function behaves like a random function), the expected number of tries is 2^n.
- 2nd preimage: 2^n, with essentially the same attack as for preimages. In this generic attack, the additional information of a given first preimage for the desired hash output cannot be used.
- Collision: $2^{n/2}$, by a birthday attack. The basic approach is to try random input values and compare each result with all previously computed hash values. If the first $k-1$ results yielded no collision, the probability for the kth value to collide is $\frac{k-1}{2^n}$. Since any pair of all previous tries may collide, the collision probability is far higher than the probability for a preimage to a fixed value for the same number of tries. Probability theory shows that the probability for a collision with N possible outputs is already higher than $\frac{1}{2}$ for roughly $\sqrt{N}$ tries. The expected number of necessary tries to find a collision is also approximately $\sqrt{N}$. The high memory consumption and lookup efforts can be reduced by logarithmic-memory algorithms

like Nivasch's [35] or memoryless techniques such as Brent's algorithm [8], without significantly increasing the expected number of tries.

Many publications do not attack entire hash functions, but simplified scenarios. Typical targets are the compression function, in particular limited-round versions of it. Collisions for the compression function are also referred to as pseudo-collisions. Differential attacks sometimes target the compression function with variable initialization values. Such results are categorized as semi-free-start or free-start collisions, depending on whether the message pair shares a common, nonstandard initialization value or two different IVs are used [26].

Under certain circumstances, some of these limited attacks can be extended for the complete hash function with additional computation effort. For instance, pseudo-collisions can sometimes be combined with near-collisions to get hash collisions with multiple input message blocks.

Chapter 3

Differential cryptanalysis

In this chapter, we introduce the general concept of differential attacks on hash functions. Differential attacks are a popular, but loosely defined category of cryptographic attacks on both hash functions and ciphers. We discuss the application of differential ideas to find collisions for hash functions. Existing methods and previous attacks by different authors are presented. Some of their limitations when applied to newer hash functions, such as SHA-2 and SHA-3, are briefly discussed. This will serve as a basis and motivation for the next chapters, where we present and analyze a new propagation method for differential attacks.

The first section introduces the basic idea and notation of differential attacks. In the second section, we show different implementations and concrete attacks by several authors, based on different definitions of "differences". The following section describes in more detail the difference concept of generalized differences that is the core of our method presented in Chapters 4 and following. Unlike some other difference concepts, generalized differences lend themselves to guess-and-determine attack strategies, as discussed in Section 3.4. Finally, Section 3.5 contains definitions and examples for bitsliced propagation, the prevalent propagation method for generalized differences. The following Chapter 4 proposes a different propagation method to compete with or supplement bitsliced propagation.

3.1 Basic idea

The core idea of differential cryptanalysis is to investigate how differences between two input messages, x and x^*, spread and propagate throughout the execution of a cryptographic primitive, such as a cipher or a hash function. What effect on intermediate results does it have, for example, if two input messages differ only in one bit? How many and which bits of the intermediate results differ after one, two, three rounds? Due to the avalanche effect usually desired in cryptographic primitives, small input changes should

have large effects on the final output. However, what about multiple changes? Can they be arranged in a way to cancel each other out?

In case of (supposedly) collision resistant hash functions, the ultimate goal of differential cryptanalysis is of course to find two such different input messages where all changes cancel out in the course of the hash computation, and the final output hashes of the two messages are equal. However, differential cryptanalysis originates from the analysis of block ciphers like DES, where statistical properties of the encryption outputs of input pairs with particular differences were used to analyze the most probable secret key used in the encryption.

While in practice, the original differential cryptanalysis of DES has little in common with the collision attacks proposed in this thesis, the two share the underlying question of how differences propagate inside cryptographic functions. In both cases, we consider a pair of input messages x and x^* to our cryptographic function f. The function f may be the encryption procedure of a block cipher (parametrized by an unknown secret key), or a hash function, or a building block of one. The messages x and x^* are only characterized by their difference $\Delta(x, x^*)$. The description of such differences and its complexity vary between different attack strategies. Examples include:

- Xor-differences: Bitwise definition as

$$\Delta(x_i, {x_i}^*)_{\text{xor}} = \begin{cases} 0 & x_i = {x_i}^*, \\ 1 & x_i \neq {x_i}^*, \end{cases}$$

  or $\Delta(x, x^*)_{\text{xor}} = x \oplus x^*$.

- Modular differences [18, 45]: Wordwise definition as $\Delta(x, x^*)_{\text{modular}} = x - x^*$.

- Signed differences [45]: Bitwise definition as

$$\Delta(x_i, {x_i}^*)_{\text{signed}} = \begin{cases} 0 & x_i = {x_i}^*, \\ + & x_i - {x_i}^* = 1, \\ - & x_i - {x_i}^* = -1. \end{cases}$$

  This provides more information than modular and xor-difference combined [14].

- Generalized differences [15]: Bitwise definition with 16 cases that describes which subset of pair values $\{(0,0),(0,1),(1,0),(1,1)\}$ is possible for $(x_i, {x_i}^*)$. See Section 3.3.

- Two-bit differences [30, 45]: Conditions on two different bit positions i and j. In analogy to generalized differences, there are 2^{16} generalized two-bit conditions. In practice, simple linear two-bit conditions such as $x_i \neq {x_i}^* = x_j = {x_j}^*$ have been used. They define xor-differences for the four bits $x_i, {x_i}^*, x_j, x_j^*$, which results in only 8 different cases.

After fixing a difference model, the two main tasks of a differential attack are the following:

1. Investigate how input differences propagate inside the function. Propagation may be deterministic or probabilistic. The output of this step is for example a difference pattern with desirable properties such as high probability.

2. Employ this knowledge to achieve the attack goal, e.g. find a collision or discover the secret key. Often, this means finding messages conforming to the difference pattern fixed earlier.

How these tasks are implemented varies widely between different attacks. For example, the original attacks on DES are based on probabilistic propagation of Xor-differences for a large number of pairs and derive information about the key from statistical analysis of the actually resulting ciphertexts. On the other hand, the attack discussed later in this thesis uses only deterministic propagation of generalized differences and guesses intermediate values for one pair of messages until a valid collision is found. For this reason, the next section briefly describes and contrasts several well-known attacks.

3.2 Differential attacks

3.2.1 Biham and Shamir's attack on DES

The foundations of differential cryptanalysis were laid by Biham and Shamir in the late 1980s [4] and early 1990s with attacks against several contemporary block ciphers, such as FEAL. Most well-known among these is their attack of DES [6]. Later publications reveal that resistance against differential attacks was already one of the design criteria for DES, and that IBM already knew about (or suspected) similar attacks as early as 1974.

The attacks are based on the xor difference model, i.e. $\Delta(x, x^*) = x \oplus x^*$. In DES, the S-box layer is the only step to actually change such an input difference. Other building blocks, such as key addition or permutations, have no effect on input differences (except simple permutation). One of the main observations by Biham and Shamir was that some input differences cause specific output differences with higher-than-average probability. Furthermore, for a fixed input and output difference, only a limited number of input messages can produce exactly this combination of input and output differences. For other input messages, the specified input difference leads to a different output difference than the specified one.

This can be quantified in a difference distribution table. Such a table contains, for all possible input and output differences, the number of input messages x satisfying this combination of input and output differences. In other words, it contains the number of

messages x such that $S(x) \oplus S(x \oplus \Delta(x, x^*)) = \Delta(y, y^*)$, where S denotes the S-box, $\Delta(x, x^*)$ is the input difference, and $\Delta(y, y^*)$ is the output difference specified.

This input (or output) difference specifies for each bit of the input (or output) message whether this bit is equal in x and x^* (or y and y^*). Since the DES S-boxes map 6-bit inputs to 4-bit outputs, the number of possible values for $\Delta(x, x^*)$ is $2^6 = 64$. The output difference $\Delta(y, y^*)$ can assume $2^4 = 16$ different values. Thus, such a table has $64 \cdot 16$ entries, each ranging between 0 and 64. The average value of such an entry is $2^6/2^4 = 4$. However, many entries will assume higher or lower values. As a trivial example, if the input difference is 0, then so must the output difference, as equal inputs must lead to equal outputs. Thus, in the table row for input difference 0, all entries will be 0, except for the column corresponding to output difference 0. This entry will be 64, since all 64 possible input messages x combined with input difference 0 will result in output difference 0.

If the table entry corresponding to $\Delta(x, x^*)$ and $\Delta(y, y^*)$ is significantly larger than 4, then $\Delta(x, x^*)$ is said to cause $\Delta(y, y^*)$ with high probability. To extend this probability to the whole round function of DES, simply multiply the probabilities of all S-boxes. Similarly, for multiple rounds, multiply the probabilities for individual rounds. This calculation assumes independence of the differences in different rounds. While such independence is usually not the case, this is a useful approximation. The succession of fixed differences after each round is referred to as a differential characteristic. If only the input and output pair is fixed, we call them a differential.

Such differential characteristics with high overall probability can now be used to actually retrieve bits of the key. The remaining bits can then be found in a brute-force attack. For the typical attack, we are interested in characteristics with high probability that cover all rounds of the encryption except for the last. Then, bits of the last round key can be recovered.

After fixing the input difference for one such characteristic, a large number of message pairs with this input difference is encrypted. Thus, the attack scenario is a chosen-plaintext attack. If the probability of our chosen high-probability characteristic is p, then we expect a fraction of p of all generated pairs to comply with this characteristic. Such pairs are also called right pairs. We need enough pairs to expect at least a small number of right pairs among them. Consequently, the number of necessary pairs is inversely proportional to p.

In typical high-probability differential characteristics, most differences are zero since these differences propagate with probability 1. S-boxes with non-zero differences are called active. The part of last round's key that is added to active S-boxes is now the target subkey. For this partial subkey, brute force search is performed. The corresponding parts of all ciphertext pairs are partially decrypted, i.e. the last round with its subkey addition is undone. If the partial key was guessed correctly, we expect that a fraction of p (or more) of all ciphertext pairs will partially decrypt to our high-probability differential for the penultimate round. Thus, we count how many ciphertexts appear to be

right pairs for each potential partial key. The partial key with most right pairs wins.

3.2.2 Attacks on other block ciphers

Shamir and Biham applied very similar attacks to a number of block ciphers. Extensions of this attack for newer block ciphers include:

- Higher-order differentials, suggested by Lai [25] and Knudsen [22].
- Truncated differentials, suggested by Knudsen [22].
- Boomerang attacks, suggested by Wagner [42].
- Impossible differentials, suggested by Knudsen [23].

For some block ciphers, difference models other than xor differences were developed. Examples include additive differentials for attacks on Twofish, and multiplicative differentials for attacks on MultiSwap by Borisov et al. [7].

3.2.3 Dobbertin's collision attacks on hash functions

Among the first to apply differential attacks to collision resistant hash functions was Dobbertin. He applied the basic method to several hash functions, including MD4 [18], MD5 [16] and RIPEMD [17]. His attack targets only the compression function of these hash functions. In case of MD5, he assumes that the pair of input messages differs in only one word. Due to the messages expansion of MD5, this word influences multiple, but not all rounds of the compression function. The underlying difference model of his attacks are modular differences, since these compression functions use modular additions as their basic operation.

Dobbertin described the compression function in terms of nonlinear equations. Due to the compressing nature of the function, such equations are heavily underdetermined. By taking advantage of this underdetermination, he was able to simplify and solve them with the help of specialized methods. While Dobbertin was able to extend his attack on the compression function to a full collision of the hash function for MD4, this was not possible for MD5.

3.2.4 Chabaud and Joux' collision attack

An alternative approach to handle nonlinear parts is to approximate them with linear functions. Such an approximation is only exact for some of the inputs. For example, a modular addition could be approximated by an xor addition. This approximation is only exact if all carry bits happen to be zero in the original modular addition.

This method was applied by Chabaud and Joux in their theoretic attack on SHA-0 [9]. Like Dobbertin, they attack only the compression function. For compatibility with their linear model, they used the xor-difference model. Their method is to first search for collisions of the linear approximation using linear algebra. This collision specifies a difference pattern for the message pair. Then, it is necessary to find messages such that the linear approximation is exact in terms of this difference pattern, i.e. the message pair must propagate according to the difference pattern of the approximation. This way, the collision for the approximation is also a collision for the original function. Chabaud and Joux describe such a difference pattern with probability 2^{-61}. A randomized search for a compatible message pair thus has a complexity of around 2^{61}.

Biham and Chen later improved this attack with their neutral bit method [5] for practical results on SHA-0 and reduced SHA-1. Rijmen and Oswald [38] also improved the original attack for theoretical results on SHA-1.

3.2.5 Wang et al.'s collision attacks on MD5 and SHA-1

In 2005, Wang et al. were the first to publish full collisions for MD5 [45]. Their method was partly inspired by Dobbertin's results. Like him, they use modular differences obtained by modular subtraction, but their main focus is on bitwise, signed differences instead. The target input message difference is fixed first, and only afterward, a corresponding differential path is searched. A notable difference compared to Dobbertin's method is that Wang splits the search for a useful difference pattern into two parts:

1. Finding a high-probability characteristic for later rounds, and
2. Finding a characteristic for the first round, not necessarily with high probability.

The characteristic for the first round is not required to hold with high probability since Wang uses message modification techniques to increase the probability afterwards. In Wang et al.'s publications, this is achieved by hand. However, it is also possible to automatically perform this search using methods like those presented in [30] or this thesis. Daum [14, p. 112ff.] suggested a basic algorithm for such an automated search. Some publications refer to the high-probability characteristic as L-characteristic, since it is defined by linear conditions, and to the lower-probability characteristic for the first round as NL-characteristic due to its nonlinear conditions.

While for MD4, Wang's method finds collisions with only one evaluation of the compression function [43], the situation is more complicated for MD5. Here, Wang constructs a multi-block collision with two evaluations of the compression function. Wang et al. also published results on SHA-1, based on the same method [44].

This attack has been used to successfully construct meaningful collisions for MD5 [14, p. 118f], such as colliding X.509 certificates [41]. Nevertheless, MD5 is still occasionally used in productive environments.

3.3 Generalized differences

Unlike the attacks described in the previous section, we use generalized differences in our approach. Generalized differences were introduced by De Cannière and Rechberger [15]. They describe the relationship between two messages in a bitwise manner, just like xor-differences or signed differences. However, they allow a finer-grained description of each bit and generalize the latter models. In particular, generalized differences can describe different degrees of determination or knowledge about one pair of bits. For example, a generalized difference can state that the bit in the first message x is 0, that in the second message x^* is 1 (like signed difference -1); or only that the two bits are different (like xor-difference 1); or that the bit in x is 1, but nothing is known about the bit in x^*. The last is an example for a case not covered by previous difference models. Thus, generalized differences can describe the whole process from initially posed conditions down to an actual explicit solution pair for these conditions. The input and output pairs can be described by a generalized differential, as well as any internal variables in a characteristic. Note however that only bitwise conditions can be described by generalized differences. For example, the condition that the 4-bit message x be either 1111 or 0000, and x^* also be either 1111 or 0000 independent of x, does not pose constraints on any single bit, so the generalized difference description of this situation would still be the very vague "no knowledge about any bit".

3.3.1 Definition

In the final, fully determined message pair (x, x^*), each bit pair $(x_i, {x_i}^*)$ at bit position i can have one of four possible pair values since each of the two bits can be either 0 or 1. Thus, the possible pair values are $\{(0,0),(0,1),(1,0),(1,1)\}$. The generalized condition of each bit is the information which of these four pairs are allowed for this particular bit position. This corresponds to a subset of these four values. For example, to describe a collision, the output value pair must be equal in each bit, which corresponds to the subset $\{(0,0),(1,1)\}$. There are $2^4 = 16$ possibilities to choose subsets of pairs from $S = \{(0,0),(0,1),(1,0),(1,1)\}$. These 16 conditions are referred to as generalized conditions. Following the convention used in [15], each generalized condition is abbreviated to a descriptive character. All 16 conditions, $\mathcal{P}(S)$, are listed in Table 3.1, where $\times$ marks which pairs are allowed. The conditions `0`, `1`, `u`, `n`, `x`, `-`, `?` and `#` are also found in other difference models such as Wang's signed differences.

To describe bitwise differences for a pair of n-bit words, a sequence of generalized conditions c_i, $0 \leq i < n$, can be used. This is also referred to as a generalized difference $\Delta(x, x^*) = [c_{n-1} \cdots c_0]$ of x, x^*, if bits are indexed from right to left.

Example 3.1. The generalized 4-bit difference [`uA71`] describes the following set S of

Name	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
`#`	−	−	−	−
`1`	−	−	−	×
`u`	−	−	×	−
`A`	−	−	×	×
`n`	−	×	−	−
`C`	−	×	−	×
`x`	−	×	×	−
`E`	−	×	×	×
`0`	×	−	−	−
`-`	×	−	−	×
`3`	×	−	×	−
`B`	×	−	×	×
`5`	×	×	−	−
`D`	×	×	−	×
`7`	×	×	×	−
`?`	×	×	×	×

Table 3.1: Generalized conditions.

message value pairs:

$$S = \{(1101, 0001), \quad (1101, 0011), \quad (1111, 0001),$$
$$(1101, 0101), \quad (1101, 0111), \quad (1111, 0101)\}$$

3.3.2 Refinement

Since each generalized condition represents a set of possible solutions for one bit, there is a natural partial ordering defined on these conditions. One condition c_0 is called a refinement (or improvement) of another condition c_1, $c_0 \trianglelefteq c_1$, if the solution set corresponding to c_0 is a subset of the c_1 set. Thus, $c_0 \trianglelefteq c_1$ means that c_0 contains more knowledge or is more constrained than c_1. The constraint c_1 is a generalization of the constraint c_0. The greatest element in this poset is `?` or "no information", the least is `#` or "contradiction".

The partial order is extended to differences as a product order, so $[c_{n-1} \cdots c_0] \trianglelefteq [\hat{c}_{n-1} \cdots \hat{c}_0]$ iff $\forall i, 0 \leq i < n : c_i \trianglelefteq \hat{c}_i$. Just as before, this means that $\hat{c}$ allows all message pairs that c allows, and possibly some more.

In our attack strategy, we will typically start with very unspecific generalized differences and gradually refine them until we have an explicit solution pair. Typically, the generalized conditions describe necessary (but not sufficient) conditions, so not all refinements contain any actually valid solutions. For example, to find a collision, we might start with

the condition that the output difference must consist of - bits. All input bits and internal state values are ?. Then, we need a gradual refinement strategy that assures that all refinements still describe a valid differential characteristic. Our refinement strategy is described in Section 3.4.

3.3.3 Propagation

Like the difference models previously described, generalized differences can propagate. However, there is one crucial difference between propagation in our attack and propagation like used for example in the original DES attack. We consider only deterministic propagation, i.e. with probability 1. This is possible since hash functions contain no secret or unpredictable operations such as the key addition in a block cipher.

Consider an elementary operation f inside the hash function, such as an xor operation or an S-box. The input and output message pairs of f are constrained by generalized differences Δ. Now only some of the solutions to these generalized conditions are actually valid input and output combinations of f. We refer to these as "true solutions". These true solutions conform to both the explicit conditions of the generalized difference Δ and the implicit conditions posed by the relation f. Often, other generalized differences $\Delta' \neq \Delta$ admit exactly the same true solutions as Δ. In this case, Δ and Δ' are called equivalent. If, additionally, Δ' is a refinement of Δ, $\Delta' \trianglelefteq \Delta$, then Δ' is called an improvement or a propagation of Δ. The improved Δ' makes some of the implicit conditions of f explicit. This process of finding an improvement Δ' is called propagation. The minimal element in the equivalence class of a difference Δ is called the optimal propagation Δ^{opt}. This minimal element is the intersection of all other elements in the same equivalence class. The following example illustrates these definitions.

Example 3.2. We analyze a simple xor operation f, where $y = f(x) = x_1 \oplus x_2$. The function input size is 2 bits, the output size 1 bit. An input and output message pair for this function is described using generalized differences. Assume we want the output bits to be equal, and the second input bit should differ. There are no explicit constraints on the first input bit. The corresponding input and output differences are

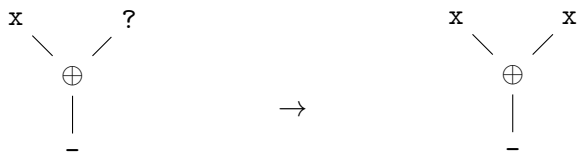
$$\Delta(x, x^*) = [\texttt{?x}], \qquad \Delta(y, y^*) = [\texttt{-}].$$

In more general terms, this elementary operation links 3 bits that can be described by

$$\Delta(z, z^*) = [\texttt{?x-}].$$

This generalized difference allows the following 16 (out of $4^3 = 64$ possible) solutions:

$$\begin{array}{llll} (000,010), & (000,110), & (100,010), & (100,110), \\ (010,000), & (010,100), & (110,000), & (110,100), \\ (001,011), & (001,111), & (101,011), & (101,111), \\ (011,001), & (011,101), & (111,001), & (111,101). \end{array}$$



(a) before propagation (b) after propagation

Figure 3.1: Example difference for xor before and after propagation.

However, 12 of them are no valid input and output combinations for f. The only true solutions are

$$(000, 110), \quad (110, 000), \quad (101, 011), \quad (011, 101).$$

Other generalized differences, such as [x?-] or [xx?], have exactly the same set of true solutions, so they are equivalent to [?x-] (but no improvements).

The minimal generalized difference that contains all the above true solutions is

$$\Delta^{\text{opt}}(z, z^*) = [\texttt{xx-}].$$

This is both an improvement of $\Delta(z, z^*) = [\texttt{?x-}]$ and its optimal propagation.

Figure 3.1 shows a simple graphical representation of the situation.

Again, note that this is only a bitwise description and cannot model more complex information on a message pair, such as conditions concerning two bits. Even the optimal propagation may still describe more solutions than are actually possible.

Example 3.3. In the above Example 3.2, the optimal propagation has 8 solutions, but only 4 true solutions.

Similarly, the generalized difference [??0] is optimally propagated, but only 4 of its 16 solutions are true solutions. The necessary condition that the second bit pair must have the same values as the first bit pair cannot be explicitly translated to generalized differences. However, any additional restrictions posed on the first bit will propagate to the second bit. This motivates the additional definition of 2-bit conditions as described in Section 4.4.

Several propagation methods and more examples are given in Section 3.5.

3.4 Guess-and-determine attacks

Our main task is to find a valid message pair and differential characteristic complying with a predefined generalized input and output difference. For more practical context, we refer to Chapter 5 or [15, 30, 31].

To achieve this, we divide the analyzed hash function into elementary steps. Then, we perform a guess-and-determine attack on the intermediate variables between those steps. The goal of this guess-and-determine attack is to refine the initially mostly undetermined generalized differences of the intermediate variables until a valid, fully determined differential solution is found. Using the terms of Subsection 3.3.3, we are searching for one true solution. Similar guess-and-determine attacks (without generalized conditions) are a general method to solve nonlinear equations. For example, Wang's message modification method [45] is also based on guessing individual bits.

This refinement process is driven by two phases, the guessing phase and the determination (or propagation) phase. The attack repeats these two phases until the solution is fully determined:

1. **Guess**: one intermediate variable bit that is not yet fully determined is chosen according to some guessing strategy. This bit is "guessed" by refining the corresponding generalized condition.
2. **Determine**: find all steps where the guessed bit is involved as an intermediate variable. Try if the other variables of this step can be propagated with our newly won information. Iterate this process for any other updated variables (e.g. with a stack of "dirty" variables). If a contradiction is encountered in the course of this propagation, the last few guesses need to be undone and different guesses made.

From a high level perspective, this approach corresponds to building a guessing tree. The tree root is the initial, mostly undetermined state. Each node corresponds to a guessing choice for one variable. The edges leading to each node's children represent the propagation process for one possible choice of the variable. This tree is randomly traversed in a depth-first way, and each encountered contradiction corresponds to a leaf of the tree. After finding a contradiction, the tree is traversed back up to some node on a higher level (representing a less refined status), and an alternative branch downward is chosen.

There is a number of optimizable design ingredients in this basic algorithm, including:

- **Choice of elementary steps**: it is not necessarily clear how elementary an elementary step should be. Choosing larger steps has the advantage of fewer intermediate variables or preserving connections between intermediate variables otherwise lost. On the other hand, propagation may be much more difficult than if the step is further subdivided. This is especially relevant in connection with linear propagation, and is discussed in Chapter 5.
- **Variable choice**: how is the next variable for guessing selected? It must, of course, be a variable that is not yet fully determined. Additionally, it may be desirable to choose a variable where one expects significant propagation. Again, a specific strategy is discussed in Chapter 5.
- **Guessing strategy**: when a variable for guessing is selected, how exactly should

the generalized condition on it be refined? For example, if the initial condition is "no knowledge" with four possible solutions, then this condition can be refined to either one, two or three solutions. The suggestion in Chapter 5 is to refine four (or three) to two solutions and two to one solution. The choice of the remaining solutions can for example be uniformly distributed. It is also possible to bias towards "nicer" solutions with high probability, such as `-` as opposed to `x`.

- **Propagation method**: a core challenge of the approach is the propagation of guesses and especially the detection of contradictions. After a variable has been guessed, other variables connected by an elementary step may be also determined. If the guess was incorrect and introduced contradictory conditions (i.e. no true solutions remain), this should be detected as early as possible. Otherwise, more time will be spent in the current dead end tree branch, resulting in futile search. While perfect propagation is possible for very small, independent elementary steps, it is effectively equivalent to brute force enumeration of possible messages and thus infeasible in the larger steps of practical applications. Alternatives include bitsliced propagation as described in Section 3.5 and linear propagation, the main topic of this thesis, and introduced beginning with Chapter 4.
- **Backtracking strategy**: After encountering a contradiction, at least the last guess needs to be undone and a different branch chosen. If one expects that contradictions are not detected immediately, it is more effective to undo several previous guesses and restart from a significantly older state. Another question is how to continue from this older state. Guessing the opposite of the last undone guess is possible, but it may also help to choose an entirely different variable for the next guess.

3.5 Propagation methods

Propagation of generalized differences is one of the central tasks in a guess-and-determine attack. Given a generalized difference, we have to decide whether the conditions are contradictory, i.e. if no true solutions exist. If solutions do exist, we want to find the optimal propagation, or at least an improvement, of the given difference. That means we want to find the most detailed possible description of the true solutions in terms of generalized differences.

Ideally, we would like to propagate the complete hash function description at once. Such global propagation would be necessary to guarantee, for example, instant detection of contradictions. Having global optimal propagation would mean that we would never have to backtrack more than one step, and the guessing tree would be basically linear and straightforward. However, for practical hash functions, this is utterly infeasible.

Instead, our attack uses local propagation. This means we split the hash function into elementary steps and try to propagate each step separately. Each step is linked to the

others by shared intermediate variables where the outputs of one step serve as inputs to the next, or several steps share common inputs. If a variable is guessed, each adjacent step is propagated separately. If any other variable conditions are refined in this process, their adjacent steps are again propagated, and so on. While such local propagation is not globally perfect, i.e. not all contradictions and improvements are discovered, it often works reasonably well in practice. Several possible algorithms for local propagation are outlined in the remainder of this section.

3.5.1 Perfect propagation

The straightforward implementation of the definition of propagation would be to enumerate all possible values conforming to the given input and output difference. Filtering out any input and output combinations that are not possible for the given step function leaves only the true solutions. Finding the optimal generalized difference description for these true solutions is now trivially achieved by noting which bit pairs actually occur among the true solutions for each bit position. More practically, it suffices to enumerate only the input values and check whether the output value produced by this input also conforms to the given generalized difference.

The feasibility of this method depends on the input bitsize of the step function (and the number of solutions to the initial input difference). For collision search and when the step function is the full hash function, this approach corresponds to trying input message pairs until one pair is found to be a collision. This performs by far worse than the generic birthday attack since the birthday paradox does not apply. By choosing smaller steps, perfect propagation becomes feasible, but at the cost of "increasingly local" results.

Example 3.4 (Perfect propagation)**.** As a first example, a simple xor operation as step function was already discussed in Example 3.2. A more complex step function is the following 4-bit function Σ, which is structurally very similar to the Σ_0 and Σ_1 functions defined for the SHA-2 hash function. The original SHA-2 functions were defined in Section 2.4. For the Σ function, every output bit depends on three input bits:

$$y = \Sigma(x) = (x \ggg 1) \oplus (x \ggg 2) \oplus (x \ggg 3),$$

where $\ggg$ denotes a rotation to the right of the 4-bit value. Figure 3.2 illustrates this step function.

First consider the initial generalized difference given by

$$\Delta(x, x^*) = [\texttt{11--}], \qquad \Delta(y, y^*) = [\texttt{????}],$$

or, equivalently, $\Delta(z, z^*) = [\texttt{11--????}]$. To propagate perfectly, we enumerate all allowed inputs x and check whether the corresponding output $\Sigma(x)$ fits the difference specified for y. In this example, any outputs will trivially conform to the unconstrained

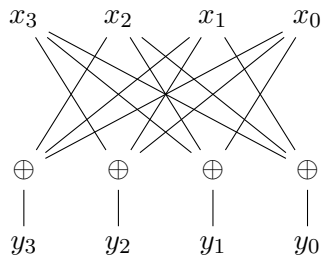


Figure 3.2: The Σ step function.

output format. Thus, by enumerating the 4 input values (instead of all 1024 solutions of $\Delta(z, z^*)$), we get the 4 true solutions

x	(1100,1100)	(1101,1101)	(1110,1110)	(1111,1111)
y	(1100,1100)	(0010,0010)	(0001,0001)	(1111,1111).

The minimal generalized difference to describe these 4 pairs is

$$\Delta^{\text{opt}}(x, x^*) = [\texttt{11--}], \qquad \Delta^{\text{opt}}(y, y^*) = [\texttt{----}].$$

As another example, assume the initial difference for the same step function is

$$\Delta(\tilde{x}, \tilde{x}^*) = [\texttt{????}], \qquad \Delta(\tilde{y}, \tilde{y}^*) = [\texttt{00--}].$$

Stubbornly applying the method would mean to enumerate all $4^4 = 256$ possible input pairs and check if the result fits the given constraints:

- input pair $(0000, 0000)$ yields output pair $(0000, 0000)$, valid, keep.
- input pair $(0000, 0001)$ yields output pair $(0000, 1110)$, not valid, drop.
- input pair $(0000, 0010)$ yields output pair $(0000, 1101)$, not valid, drop.
- ...

However, note that the function Σ is an involution, i.e. the inverse function to Σ is Σ itself. Thus, instead of enumerating inputs, it is also possible to enumerate outputs and apply Σ. In this example, this is an advantage, since there are only 4 possible output pairs. Thus, just like in the first example, we quickly get the 4 true solutions:

$\tilde{x}$	$(0000, 0000)$	$(1110, 1110)$	$(1101, 1101)$	$(0011, 0011)$
$\tilde{y}$	$(0000, 0000)$	$(0001, 0001)$	$(0010, 0010)$	$(0011, 0011)$.

The minimal generalized difference is

$$\Delta^{\text{opt}}(\tilde{x}, \tilde{x}^*) = [\texttt{----}], \qquad \Delta^{\text{opt}}(\tilde{y}, \tilde{y}^*) = [\texttt{00--}].$$

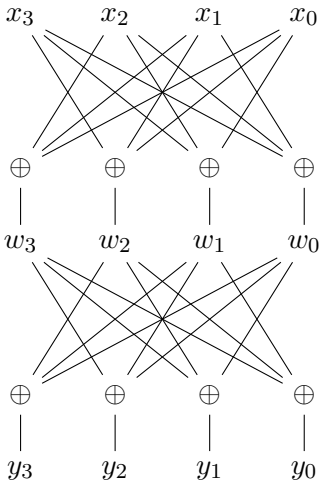


Figure 3.3: The double-Σ step function.

Example 3.5 (Locally perfect is not globally perfect)**.** We now assume a hash function contains two such Σ steps, where the output of the first Σ step serves as input to the second Σ step. Figure 3.3 illustrates the construction. Additionally, let the input and output pair to this construction be constrained by

$$\Delta(x, x^*) = [\texttt{11--}], \qquad \Delta(y, y^*) = [\texttt{00--}].$$

The intermediate variables w are not constrained, as in Figure 3.4.

First, assume this function is separated into two steps corresponding to the two separate Σ building blocks and then propagated locally. These building blocks and generalized differences are exactly the same as in the previous example, so the intermediate variables are propagated to

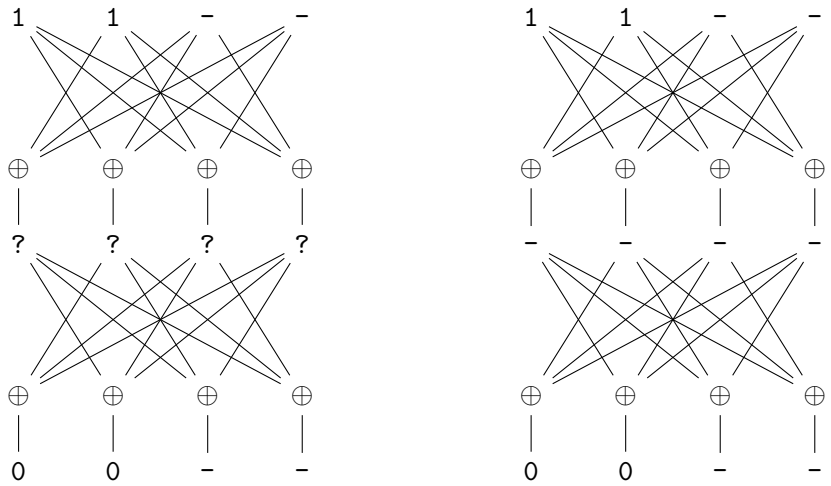
$$\Delta(w, w^*) = [\texttt{----}],$$

as illustrated in Figure 3.4.

This locally perfect propagation still allows 256 solutions. However, none of these are true solutions. This can also be easily seen from the involution property of Σ. Since Σ is its own inverse function, the double-Σ construction is equivalent to the identity function, and any input pair will be mapped to exactly the same output pair. Since the input conditions contradict the output conditions, there can be no solutions. The optimal global propagation is thus

$$\Delta(x, x^*) = [\texttt{####}], \qquad \Delta(y, y^*) = [\texttt{####}].$$

This cannot be deduced using only generalized differences and local propagation of the two separate steps.



(a) before propagation (b) after propagation

Figure 3.4: Example difference for double-Σ before and after local propagation.

3.5.2 Bitsliced propagation

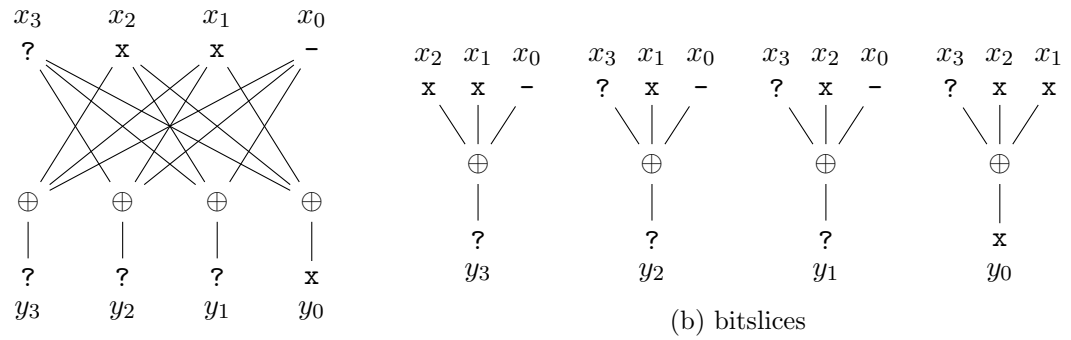
Bitsliced propagation is a practical implementation of perfect propagation for minimal step sizes. Each step has only one or very few output bits, and very few input bits. Any larger steps are split up ("sliced") into several such bitslice steps. Thus, bitsliced propagation has a very local focus, but is locally perfect.

To improve implementation performance and avoid repeatedly enumerating inputs every time, it is possible to precompute the perfect propagation for each possible initial difference. For a small number of input bits, all possible propagations can be precomputed and stored in an exhaustive table that maps any generalized difference to its optimal propagation. If such a table would become too large, evaluation can be performed lazily when necessary and then stored in an associative table for future reference. For symmetrically defined step functions such as xor operations, the necessary table space can be further limited.

Example 3.6. The Σ-function from Example 3.4 has an output size of 4 bits. For bitsliced propagation, this function needs to be subdivided into 4 bitslices, one for each output bit. In this case, each bitslice corresponds to an xor operation with 3 input bits. Assume an initial difference of

$$\Delta(x, x^*) = [\texttt{?xx-}], \qquad \Delta(y, y^*) = [\texttt{???x}].$$

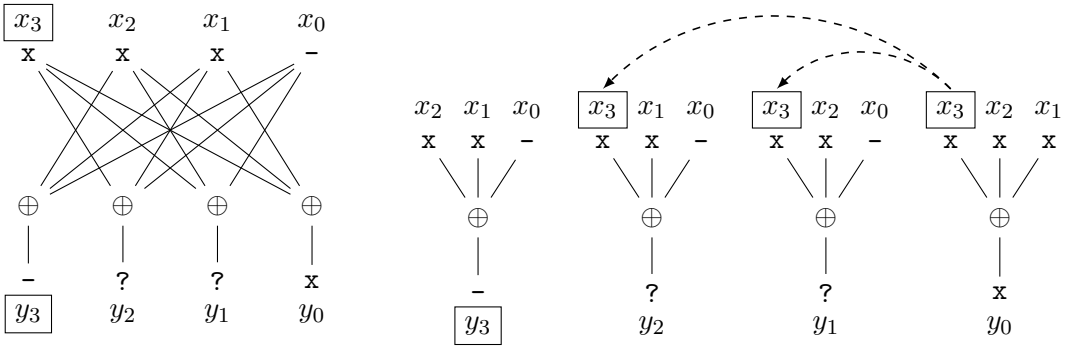
Figure 3.5 shows the initial status of the whole Σ function and of the separate bitslices. The bitslices share some input variables, but are propagated separately using perfect propagation.



(a) whole step

(b) bitslices

Figure 3.5: Example difference for Σ and bitslices, before propagation.



(a) whole step

(b) bitslices

Figure 3.6: Example difference for Σ and bitslices, after one propagation iteration.

The first bitslice with output bit y_3 has 8 possible input pairs that are all mapped to either $(0, 0)$ or $(1, 1)$ since the differences cancel out. The combined input and output difference [xx-?] is propagated to [xx--]. For bitslices y_2 and y_1, nothing propagates. In the last bitslice for y_0, input pairs $(1, 1)$ and $(0, 0)$ for the bit x_3 lead to invalid outputs, so this combined difference [?xxx] propagates to [xxxx]. Since x_3 also appears in two other bitslices, it is updated there, as illustrated in Figure 3.6.

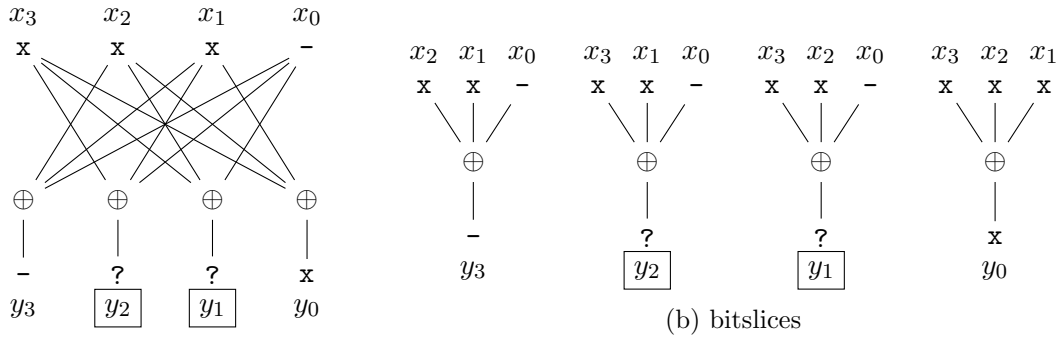
Now x_3 is marked as dirty in the bitslices y_2 and y_1, and these two need to be propagated again. Like the y_3 bitslice earlier, they propagate to [xx--]. The final differences are given in Figure 3.7.

In this example, the final bitsliced propagation after two iterations equals the perfect propagation for the whole step. Whereas perfect propagation requires

$$4 \cdot (4 \cdot 2 \cdot 2 \cdot 2) = 128 \text{ xor evaluations},$$

the naive bitsliced version needs only

$$8 + 16 + 16 + 16 + 8 + 8 = 72 \text{ xor evaluations}.$$



(a) whole step

(b) bitslices

Figure 3.7: Example difference for Σ and bitslices, completely propagated after two iterations.

While this seems a rather mediocre tradeoff considering the loss of optimality, the difference becomes more significant with increasing input size. For example, consider a similar construction with 32 bit inputs and outputs, where each output bit again depends on 3 input bits. Such step functions occur in SHA-2. Here, in the worst case, perfect propagation would require

$$4^{32} = 2^{64} \text{ xor evaluations.}$$

One iteration of bitsliced propagation, on the other hand, needs less than

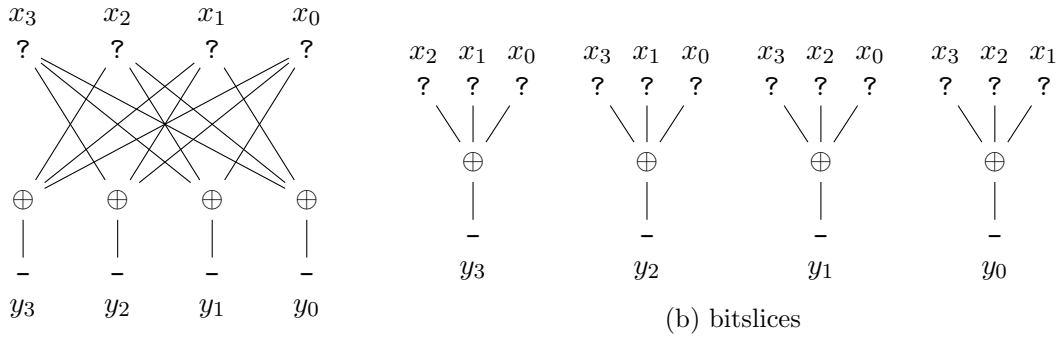
$$32 \cdot 4^3 = 2^{11} \text{ xor evaluations,}$$

and the number of iterations is very limited. Each bitslice can be iterated at most $3 \cdot 4 + 1 = 13 < 2^4$ times. While the first is infeasible, the latter can be computed instantly. Of course, the larger the original step, the worse bitsliced propagation works as an approximation of the perfect propagation. In case of the SHA-2 $\Sigma_0, \Sigma_1, \sigma_0$ and σ_1 functions, bitsliced propagation is very useful, but needs additional techniques to work well [30].

The above example also shows that it is not necessary to always enumerate all inputs to propagate a bitslice. The starting differences of y_2 and y_1 in the second iteration exactly equal the starting difference of y_3 in the first iteration. Instead of repeating the calculation, the earlier propagation result can simply be saved and reused. For a 4-bit bitslice like above, there are $16^4 = 65536 = 2^{16}$ possible starting differences. Each propagation result can be stored using $4 \cdot 4 = 16$ bits, so a complete table of all results requires less than 1 MB.

Example 3.7 (Bitsliced propagation is not perfect)**.** Just like in Example 3.5, (local) bitsliced propagation is often not (globally) perfect. For instance, consider the initial difference

$$\Delta(x, x^*) = [\texttt{????}], \qquad \Delta(y, y^*) = [\texttt{----}]$$



(a) whole step

(b) bitslices

Figure 3.8: Example difference for Σ and bitslices, no propagation possible.

for our 4-bit Σ function, like in Figure 3.8. None of the bitslices propagates.

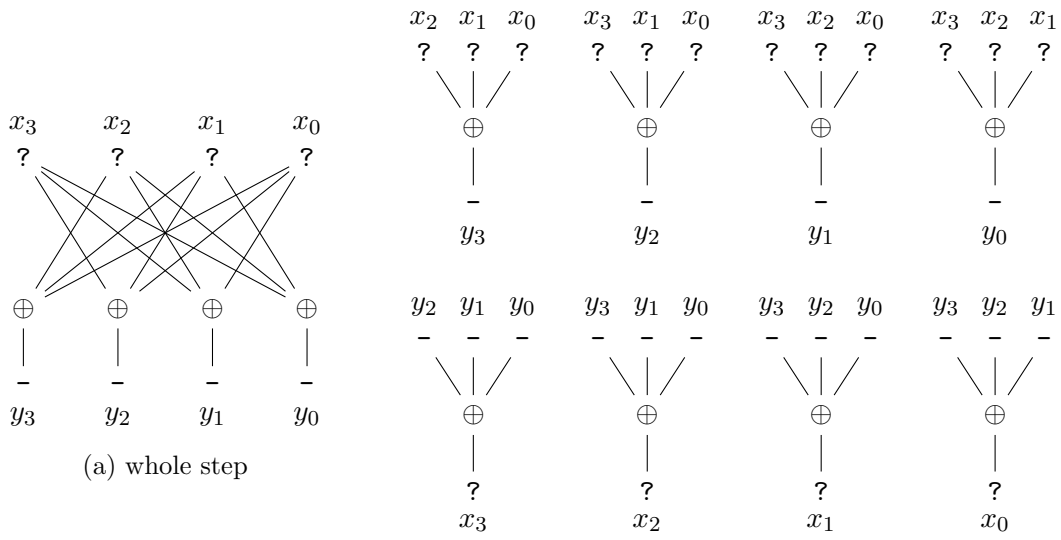
However, since the inverse function to Σ is Σ itself, it is obvious that this difference should propagate to

$$\Delta^{\text{opt}}(x, x^*) = [----], \qquad \Delta^{\text{opt}}(y, y^*) = [----].$$

To fix bitsliced propagation at least in examples similar to this, it is possible to define additional bitslices for the inverse function, with one extra bitslice for each input bit. Figure 3.9 shows such inverse bitslices. With this improvement, this example does propagate correctly. Nevertheless, many cases remain where the improved bitslice method still fails to propagate perfectly.

3.5.3 Linear propagation

Linear propagation is a new propagation method proposed in [19]. The goal is to provide a more global propagation than the bitsliced method, but cheaper than perfect propagation. It is based on linear algebra and equivalent to perfect propagation in case only linear functions and a particular set of generalized conditions are involved. Otherwise, in case of nonlinear step functions, it can be combined with bitsliced propagation for a powerful hybrid propagation method. The theoretical foundations for this linear method are presented in Chapter 4, and practical implementation issues are discussed in Chapter 5.



(a) whole step

(b) normal and inverted bitslices

Figure 3.9: Example difference for Σ and bitslices, both normal and inverted.

Chapter 4

Linear propagation

The focus of this chapter will be the linear propagation algorithm, employing linear equations and linear solving techniques in order to propagate linear generalized differences in a hash function. This chapter covers the mathematical background and definition of the algorithm, while the next chapter discusses implementation considerations.

As previously described in Chapter 3, we consider a function f and a pair of inputs and their respective outputs. The set of possible inputs and outputs is described in terms of generalized differences. Such a differential description permits a number of potential solutions. We want to refine this description by canceling out invalid solutions that cannot occur due to the definition of f. Currently, this propagation of differential information is often achieved by means of a bitsliced description of f. As we demonstrated in Section 3.5, the results of this method are not always satisfactory. Many invalid solutions are not detected because the focus on bitsliced relationships loses the larger, word-level context.

Therefore, Eichlseder et al. proposed in [19] to describe generalized differences in terms of linear equations. Then, linear algebra can be used to eliminate invalid solutions.

In the remainder of this chapter, we will first describe in detail how to model functions and conditions using linear equations in Sections 4.1 and 4.2. Section 4.3 shows that Gaussian elimination can be used for propagation. In Section 4.4, we discuss how multi-bit equations can be used to improve the guessing strategy. Finally, Sections 4.5 and 4.6 investigate how this linear method can also be employed in nonlinear settings.

4.1 Linear conditions

We want to translate generalized conditions to linear equations. When talking about linear equations, we mean binary affine linear equations of the form

$$a_0 \cdot x_0 \oplus a_1 \cdot x_1 \oplus \ldots \oplus a_{m-1} \cdot x_{m-1} = \bigoplus_{j=0}^{m-1} a_j \cdot x_j = b, \qquad a_j, b \in \mathbb{F}_2,$$

where the variables $x_j \in \mathbb{F}_2, 0 \leq j < m$ and coefficients $a_j, b \in \mathbb{F}_2$ are binary values of the field $\mathbb{F}_2 = \{0, 1\}$. Addition in this field means integer addition modulo 2, which is equivalent to the XOR operation $\oplus$. Multiplication modulo 2 corresponds to the binary AND operation.

Our target is to describe both the function f and the differential conditions using such linear equations. We then combine these equations and simplify or "solve" the system to extract refined (propagated) differential conditions.

The variables of these equations are the bits of the input and output pair to the function f. Thus, if we denote the input pair of the m-to-n bit function f by $x = (x_{m-1} \cdots x_0)$ and $x^* = ({x_{m-1}}^* \cdots {x_0}^*)$ and the output message pair by $f(x) = y = (y_{n-1} \cdots y_0)$ and $f(x^*) = y^* = ({y_{n-1}}^* \cdots {y_0}^*)$, we have $2(m + n)$ binary variables $x_i, {x_i}^*, y_j$ and ${y_j}^*$ for $0 \leq i < m$ and $0 \leq j < n$.

For simpler notation, we relabel them to z_j, z_j^*, $0 \leq j < m+n$, where for $0 \leq i < m$ and $0 \leq j < n$,

$$z_i = x_i, \quad z_{m+j} = y_j, \quad {z_i}^* = {x_i}^*, \quad {z_{m+j}}^* = {y_j}^*.$$

A linear equation of k binary variables has between 0 and 2^k solutions. The exact number of solutions is always either 0 or a power of 2, i.e. 2^ℓ for some $\ell \in \{0, \ldots, m\}$, since the solutions span an ℓ-dimensional affine space. This remains true when considering a set of multiple simultaneous linear equations over a common set of k variables.

In case of generalized differences, the conditions for one bit involve two variables, z and z^*.Thus, any linear equation or equation system in these variables permits either 0, 1, 2 or 4 solutions. Each solution space of an equation system corresponds to the solution set of one generalized condition. However, the converse is not true: some generalized conditions allow 3 solutions, a situation that cannot be mapped to linear equations only in the variables z and z^*. We will return to this problem in Section 4.5. For now, we consider only generalized conditions with 0, 1, 2 or 4 solutions. These can be translated to simultaneous linear equations, although the representation is not necessarily unique since linear recombinations of multiple equations in a system permit the same solution space. Table 4.1 lists all generalized differences as introduced in Section 3.3 and their translation to equivalent linear equations where such equations exist.

As seen in the table, generalized conditions with one solution are translated to two equations (both z and z^* are fixed separately); conditions with two solutions to one equation

Name	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$	Conditions
`#`	–	–	–	–	$0 = 1$, contradiction
`1`	–	–	–	×	$z = 1, z^* = 1$
`u`	–	–	×	–	$z = 1, z^* = 0$
`A`	–	–	×	×	$z = 1$
`n`	–	×	–	–	$z = 0, z^* = 1$
`C`	–	×	–	×	$z^* = 1$
`x`	–	×	×	–	$z \oplus z^* = 1$
`E`	–	×	×	×	nonlinear
`0`	×	–	–	–	$z = 0, z^* = 0$
`-`	×	–	–	×	$z \oplus z^* = 0$
`3`	×	–	×	–	$z^* = 0$
`B`	×	–	×	×	nonlinear
`5`	×	×	–	–	$z = 0$
`D`	×	×	–	×	nonlinear
`7`	×	×	×	–	nonlinear
`?`	×	×	×	×	$0 = 0$, no constraints

Table 4.1: Linear equations for all generalized conditions.

(either, z or z^* is unconstrained, or only the difference $z \oplus z^*$ is fixed); conditions with three solutions are nonlinear, and the condition allowing all four solutions is translated to an empty equation system since there are no constraints on any variables. Dropping the (nonlinear) equations that describe the nonlinear conditions is equivalent to allowing all four instead of three solutions for this bit pair.

To translate a generalized difference for the whole input and output words to equations, we simply join the equations for all bits (by setting all coefficients except for the concerned bit to 0). The resulting equation system consists of equations of the form

$$c \cdot z_i \oplus c^* \cdot {z_i}^* = b, \qquad c, c^*, b \in \mathbb{F}_2,$$

or

$$\begin{pmatrix} c & c^* \end{pmatrix} \cdot \begin{pmatrix} z_i \\ {z_i}^* \end{pmatrix} = \begin{pmatrix} b \end{pmatrix}.$$

Each bit contributes to up to two such equations.

If a generalized difference contains only linear conditions, it is called a linear generalized difference; in this case, the equation system constructed as above permits exactly the same solution space as the generalized difference.

4.2 Linear functions

After translating generalized conditions to linear equations, we want to describe the hash function in the same terms. Only the linear parts of the hash function can be translated to such equations. In the following, f will denote such a linear building block of the hash function under attack.

An (affine) linear function $f : \mathbb{F}_2^m \to \mathbb{F}_2^n$, $x = (x_{m-1} \cdots x_0) \mapsto y = (y_{n-1} \cdots y_0)$, is defined by equations of the form

$$f_i(x) = y_i = \bigoplus_{j=0}^{m-1} a_{ij} \cdot x_j \oplus e_i, \qquad 0 \le i < n, \quad a_{ij}, e_i \in \mathbb{F}_2.$$

Similarly to purely linear functions, such affine linear functions f are characterized by the property that

$$\begin{aligned} f(x \oplus x') &= f(x) \oplus f(x') \oplus f(0^m), \qquad \text{and} \\ f(a \cdot x) &= a \cdot f(x) \oplus (1 \oplus a) \cdot f(0^m), \end{aligned}$$

where $x, x' \in \mathbb{F}_2^n$, $a \in \mathbb{F}_2$, and the operators $\oplus$ and $\cdot$ denote the vector addition and scalar multiplication over $\mathbb{F}_2$. The vector 0^n is the zero vector $0^n = (0, \ldots, 0)$, and $f(0^m) = e = (e_{n-1} \cdots e_0)$. In case of purely linear functions, $e = 0^n$.

When considering input pairs x, x^*, the same coefficients a_{ij}, e_i also apply to the ${y_i}^*$ and ${x_j}^*$ variables. Using the $z_j, {z_j}^*$ variables defined previously, this can be reordered to

$$\begin{aligned} \bigoplus_{j=0}^{m-1} a_{ij} \cdot z_j \oplus z_{m+i} &= e_i, \qquad 0 \le i < n, \\ \bigoplus_{j=0}^{m-1} a_{ij} \cdot {z_j}^* \oplus {z_{m+i}}^* &= e_i, \qquad 0 \le i < n. \end{aligned}$$

Thus, an m-to-n-bit function generates $2n$ equations on $2(m+n)$ variables. These equations are all linearly independent since the variables $z_m, \ldots, z_{m+n-1}, {z_m}^*, \ldots, {z_{m+n-1}}^*$ occur only in exactly one equation each.

If the function f is known to be (affine) linear, but not given in the explicit equation form defined above, the matrix $A = (a_{ij})$ and vector $e = (e_{n-1} \cdots e_0)$ can be automatically derived as follows.

4.2.1 Constructing the function matrix

As known from linear algebra, for purely linear f (i.e. $e = 0^n$), the ith column of A equals the image of the ith standard basis vector s_i. s_i contains only zero elements

except for the ith position, where it is set to one. This holds for linear functions f over any vector space and underlying field, also for our case with field elements $\mathbb{F}_2 = \{0, 1\}$, addition $\oplus$ and multiplication $\cdot$.

To extend this for affine linear functions, first note that as seen above, $f(0^m) = e$. Then, $\tilde{f} = f \oplus e$ is purely linear, and $A_{\tilde{f}} = A_f$ can be retrieved as just described above. Summarizing,

$$e = f(0^m), \quad \tilde{f}(x) = f(x) \oplus e, \quad A = \begin{pmatrix} \tilde{f}(s_1) & \cdots & \tilde{f}(s_n) \end{pmatrix}.$$

In the following text, affine linear functions are referred to as linear functions.

4.2.2 Combined matrix

Now, we are able to express both the (linear) function f and the (linear) generalized conditions $\Delta(x, x^*)$ for input and output messages in terms of equivalent simultaneous linear equations with variables $z_0, \ldots, z_{m+n-1}, {z_0}^*, \ldots, {z_{m+n-1}}^*$. This whole system of simultaneous equations can be written in matrix representation as follows:

$$\begin{pmatrix}
a_{00} & \cdots & a_{0\bar{m}} & 1 & & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & & \vdots & & \ddots & & \vdots & & \vdots & \vdots & & \vdots \\
a_{\bar{n}0} & \cdots & a_{\bar{n}\bar{m}} & 0 & & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
0 & \cdots & 0 & 0 & \cdots & 0 & a_{00} & \cdots & a_{0\bar{m}} & 1 & & 0 \\
\vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots & & \ddots & \\
0 & \cdots & 0 & 0 & \cdots & 0 & a_{\bar{n}0} & \cdots & a_{\bar{n}\bar{m}} & 0 & & 1 \\
 & & & & & & & & & & & \\
c_{0,0} & & & & & 0 & c^*_{0,0} & & & & & 0 \\
c_{0,1} & & & & & & c^*_{0,1} & & & & & \\
 & c_{1,0} & & & & & & c^*_{1,0} & & & & \\
 & c_{1,1} & & & & & & c^*_{1,1} & & & & \\
 & & & \ddots & & & & & & \ddots & & \\
 & & & & & c_{\overline{m+n},0} & & & & & & c^*_{\overline{m+n},0} \\
0 & & & & & c_{\overline{m+n},1} & 0 & & & & & c^*_{\overline{m+n},1}
\end{pmatrix}
\cdot
\begin{pmatrix}
x_0 \\ \vdots \\ x_{\bar{m}} \\ y_0 \\ \vdots \\ y_{\bar{n}} \\ {x_0}^* \\ \vdots \\ {x_{\bar{m}}}^* \\ {y_0}^* \\ \vdots \\ {y_{\bar{n}}}^*
\end{pmatrix}
=
\begin{pmatrix}
e_0 \\ \vdots \\ e_{\bar{n}} \\ e_0 \\ \vdots \\ e_{\bar{n}} \\ \\ b_{0,0} \\ b_{0,1} \\ b_{1,0} \\ b_{1,1} \\ \vdots \\ b_{\overline{m+n},0} \\ b_{\overline{m+n},1}
\end{pmatrix},$$

where $\bar{k} = k - 1$. In block matrix notation using

$$
\begin{aligned}
v &= \begin{pmatrix} v_0 & \cdots & v_{\bar{n}} \end{pmatrix}^t, \qquad v \in \{y, y^*, e\} \\
v &= \begin{pmatrix} v_0 & \cdots & v_{\bar{m}} \end{pmatrix}^t, \qquad v \in \{x, x^*\} \\
A &= \begin{pmatrix} a_{00} & \cdots & a_{0\bar{m}} \\ \vdots & & \vdots \\ a_{\bar{n}0} & \cdots & a_{\bar{n}\bar{m}} \end{pmatrix}, \\
I &= \begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{pmatrix}, \qquad 0 = \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{pmatrix}, \\
\operatorname{diag} C &= \begin{pmatrix} c_0 & & 0 \\ & \ddots & \\ 0 & & c_{\overline{m+n}} \end{pmatrix}, \qquad b = \begin{pmatrix} b_0 \\ \vdots \\ b_{\overline{m+n}} \end{pmatrix}, \quad \text{where} \quad c_i = \begin{pmatrix} b_{i,0} \\ b_{i,1} \end{pmatrix},
\end{aligned}
$$

this corresponds to

$$
\begin{pmatrix} A & I & 0 & 0 \\ 0 & 0 & A & I \\ \operatorname{diag} C & & \operatorname{diag} C^* & \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ x^* \\ y^* \end{pmatrix} = \begin{pmatrix} e \\ e \\ b \end{pmatrix}.
$$

The matrix A and vector e define the function f, and C, C^*, b describe the generalized conditions.

The total number k of equations (excluding $0 = 0$ lines) depends on the number and type of the involved conditions and satisfies $2n \leq k \leq 2m + 4n$. The lower bound corresponds to no linear constraints at all, the upper bound to an already completely determined difference where each input condition only permits one solution and so each input and output bit contributes 2 equations, summing up to $2(m + n)$.

Example 4.1. Let $f = \Sigma : \{0,1\}^4 \to \{0,1\}^4$ be the linear function defined in Example 3.4 as

$$y = f(x) = (x \ggg 1) \oplus (x \ggg 2) \oplus (x \ggg 3).$$

Let us consider the input and output difference

$$
\begin{aligned}
\Delta(x, x^*) &= \texttt{[???1]}, \\
\Delta(y, y^*) &= \texttt{[00--]}.
\end{aligned}
$$

The corresponding equation system can be found in Figure 4.1. The first 8 rows describe f, and the remaining rows correspond to the given conditions as in Table 3.1.

If all involved equations are linear, methods similar to Gaussian elimination can be applied to simplify them.

$$\begin{pmatrix}
\cdot & \cdot & 1 & \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & 1 & \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\
1 & \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
1 & \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\
\cdot & 1 & \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1
\end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_0^* \\ x_1 \\ x_1^* \\ x_2 \\ x_2^* \\ x_3 \\ x_3^* \\ y_0 \\ y_0^* \\ y_1 \\ y_1^* \\ y_2 \\ y_2^* \\ y_3 \\ y_3^* \end{pmatrix} = \begin{pmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ 1 \\ 1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \end{pmatrix}$$

Figure 4.1: Matrix for Σ and $\Delta(x, x^*) =$ `[???1]`, $\Delta(y, y^*) =$ `[00--]`.

4.3 Solving simultaneous linear equations

The idea of this approach is to perform some operations on the equation system and then translate it back to generalized conditions.

In the following, we only consider linear differences: nonlinear conditions with three solutions are replaced by the linear condition with no equations and four solutions.

Claim. Let f be a linear function and $\Delta(x, x^*)$ a linear difference. Then the equations for all conditions of the optimal propagation $\Delta^{\text{opt}}(x, x^*)$ of $\Delta(x, x^*)$ can be generated as linear combinations of the equations for f and $\Delta(x, x^*)$ (i.e. by adding up some selected equations).

Proof. Since $\Delta(x, x^*)$ and f are linear, $\Delta^{\text{opt}}(x, x^*)$ can also be expressed in linear equations. If one of the optimal equations would be linearly independent of the original equations, the combined system of original and optimal equations would admit a strictly smaller solution space since increasing the rank of an equation system decreases the dimension of the solution space. This is a contradiction to the definition of $\Delta^{\text{opt}}(x, x^*)$. □

Instead of trying all 2^k possible linear combinations of the k equations, it suffices to use the standard algorithm for solving simultaneous linear equations, Gauss-Jordan elimination.

4.3.1 Gauss-Jordan elimination

Gauss-Jordan elimination transforms an equation system to reduced row echelon form. This form is defined by the property that for each row, the column of the first non-zero element (the pivot element) contains no other non-zero elements. The rows of the equation system are sorted in ascending order by the position index of their pivot element. This form is established as described in Algorithm 1.

Algorithm 1 Gauss-Jordan elimination for binary equation systems

Input: Simultaneous equations $Ax = b$, where $A \in \{0,1\}^{m \times n}$.
Output: $\tilde{A}x = \tilde{b}$ with the same solutions as $Ax = b$ and $\tilde{A}$ in reduced row echelon form.

$C \leftarrow (A|b)$, $j \leftarrow 1$
for $i \leftarrow 1, \ldots, m$ **do**
  while $c_{kj} = 0$ for all $i \leq k \leq m$ **do**
    $j \leftarrow j + 1$
  swap row i with the first row $k \geq i$ such that $c_{kj} = 1$
  for $k \leftarrow i + 1, \ldots, m$ **do**
    if $c_{kj} = 1$ **then**
      row $k \leftarrow$ row $k \oplus$ row i
$m' \leftarrow \max\{m'' : \text{ row } m'' \text{ is not all zero}\}$
for $i \leftarrow m', \ldots, 1$ **do**
  $j \leftarrow \min\{j' : c_{ij'} \neq 0\}$
  for $k \leftarrow i - 1, \ldots, 1$ **do**
    if $c_{kj} = 1$ **then**
      row $k \leftarrow$ row $k \oplus$ row i
$(\tilde{A}|\tilde{b}) \leftarrow C$
return $\tilde{A}, \tilde{b}$

The algorithm consists of a downward elimination phase and an upward elimination phase. During downward elimination, it ensures that the index of the first nonzero (pivot) element increases with increasing row numbers, and also eliminates all occurrences of nonzero elements in the column below each pivot element. Afterwards, in the upward elimination phase, it eliminates all occurrences of nonzero elements in the column above each pivot element. This does not induce any new conflicting nonzero elements in pivot columns or change pivots since in iteration i, only rows with pivot elements left of any nonzero elements in row i are touched.

Downward elimination of row i with pivot index j involves $m - i$ rows with $n - j + 1$ xor operations each in the worst case. Upward elimination involves $i - 1$ rows with $n - j + 1$ xor operations each. In case of a full-rank $m \times m$ matrix, $i = j$ and these operations

sum up to

$$N = \sum_{i=1}^{m} i(i-1) + \sum_{i=1}^{m} (i-1)(m-i+1) = \sum_{i=1}^{m} (i^2 - i + im - i^2 + i - m + i - 1)$$
$$= (m+1)\sum_{i=1}^{m} i - m^2 - m = \frac{(m+1)^2 m}{2} - m^2 - m = \frac{m^3}{2} - \frac{m}{2}.$$

Since the xor operations can also be performed on word level if the equation coefficients are stored accordingly, the number of necessary xor operations decreases by a corresponding factor.

Claim. Let f be a linear function and $\Delta(z, z^*)$ a linear difference. Consider the reduced row echelon form received from Gauss-Jordan elimination of the equation system for f and $\Delta(z, z^*)$. Then the equations for all generalized conditions of the optimal propagation $\Delta^{\text{opt}}(z, z^*)$ of $\Delta(z, z^*)$ are either directly contained in this reduced system, or they can be constructed by adding two rows. In the latter case, the pivot elements of those two rows are a pair $z_j, {z_j}^*$.

Proof. The rows of the reduced row echelon form span the same subspace as the original system. Therefore, all equations of the generalized conditions in $\Delta^{\text{opt}}(z, z^*)$ can be constructed as linear combinations of the reduced row echelon form. Note that after the Gauss-Jordan elimination step, the pivot is the only non-zero element in its column. It follows that a linear combination of k rows will contain at least k non-zero coefficients.

Remember that the equations for each generalized condition are either of the form $z_j \oplus z_j^* = \{0, 1\}$, or consist of one or two equations $z_j = \{0, 1\}$ or $z_j^* = \{0, 1\}$. In the latter case, the equations are directly rows of the reduced system, since they contain only one non-zero element (the pivot). An equation of the form $z_j \oplus z_j^* = \{0, 1\}$ can only be formed if it is either a row of the reduced system, or if both variables are pivots. In the second case, we get the equation by adding the row with pivot z_j to the row with pivot z_j^*. □

Summarizing, the generalized conditions for bit j of the optimal propagation $\Delta^{\text{opt}}(x, x^*)$ of a linear difference $\Delta(x, x^*)$ can be read from the reduced row echelon form by checking only the rows with pivots x_j or ${x_j}^*$.

Example 4.2. Applying Gauss-Jordan elimination to the system in Example 4.1 leads to the matrix in Figure 4.2. For bits x_0, y_0, y_1, y_2 and y_3, the condition equations are explicitly contained in this reduced system. To extract conditions for the remaining bits $z \in \{x_1, x_2, x_3\}$, it is necessary to add up the rows with pivots z and z^* to cancel out the other involved variable y_0^*. Summarized, the propagated conditions are

$$\Delta^{\text{opt}}(x, x^*) = \texttt{[---1]},$$
$$\Delta^{\text{opt}}(y, y^*) = \texttt{[001-]}.$$

This is optimal.

$$\begin{pmatrix}
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1
\end{pmatrix}
\cdot
\begin{pmatrix}
x_0 \\ x_0^* \\ x_1 \\ x_1^* \\ x_2 \\ x_2^* \\ x_3 \\ x_3^* \\ y_0 \\ y_0^* \\ y_1 \\ y_1^* \\ y_2 \\ y_2^* \\ y_3 \\ y_3^*
\end{pmatrix}
=
\begin{pmatrix}
1 \\ 1 \\ \cdot \\ \cdot \\ 1 \\ 1 \\ 1 \\ 1 \\ \cdot \\ 1 \\ 1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot
\end{pmatrix}$$

Figure 4.2: Matrix for Σ and $\Delta(x, x^*) =$ `[???1]` and $\Delta(y, y^*) =$ `[00--]`, after Gaussian elimination.

4.3.2 Incremental elimination

As mentioned earlier, in guess-and-determine attacks, the original conditions are gradually updated by refining single conditions. While it is possible to always recreate the whole equation system and solve it from scratch, it is usually more economic to save the system for the current conditions and only add equations for the updated variables if conditions are refined. Since these updated conditions are refinements of the previous ones, the old equations are still valid.

Similarly, often the same function f is combined with a larger amount of different generalized differences. It may be useful to precompute the equations for f and store them in reduced row echelon form, or at least the unreduced echelon form received from performing only Gaussian (downward) elimination. The equations for any generalized differences can then be inserted into the prepared echelon form, which may be more economic than transforming the whole system from scratch.

To add a new equation E to an echelonized equation system S, first reduce E by adding any equations from S whose pivot variable occurs in E. If E was linearly dependent on S, this procedure will reduce E to zero (or to $0 = 1$ if only the variable part is dependent and not the whole equation). Else, the remaining equation E' has a new pivot variable not yet present in S and can be sorted between the rows with adjacent pivot indices. In this case, upward elimination needs to be completed by adding E' to any equations in S in which the pivot variable of E' occurs. After this procedure, the system includes the

information of E and is again in reduced row echelon form. The method is summarized in Algorithm 2, and we refer to it as ε-Gauss-Jordan elimination.

Algorithm 2 ε-Gauss-Jordan elimination to add equations to a system in reduced row echelon form

Input: Simultaneous equations S in reduced row echelon form with m rows and an equation E to add to the system.
Output: System $\tilde{S}$ equivalent to $S \cup \{E\}$ in reduced row echelon form.

```
E' ← E
for i ← 1, ..., m do
  if pivot variable of row i occurs in E' then
    E' = E' ⊕ row i
    if E' is all zero then
      return S
j ← Pivot index of E'
find k such that Pivot index of row k < j < Pivot index of row k + 1
for i ← 1, ..., k do
  if variable j occurs in row i then
    row i ← row i ⊕ E'
add E' to S between rows k and k + 1 and return this system as S̃.
```

The number of necessary xor operations is in the worst case, if the new pivot of E' is at position j and S is an $m \times m$ system,

$$\sum_{i=1}^{m} i + (j-1)(m-j+1) \leq \frac{m(m+1)}{2} + \frac{(m+1)^2}{4} = \frac{3}{4}m^2 + \mathcal{O}(m).$$

Since many steps, including the transformation of the equation system back to generalized conditions, need to know the pivot index of an equation, storing this value explicitly may improve the performance of this method.

This concludes the basic linear propagation algorithm for generalized differences. Summarizing, we construct the matrix equation system

$$\begin{pmatrix} A & I & 0 & 0 \\ 0 & 0 & A & I \\ \operatorname{diag} C & & \operatorname{diag} C^* & \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ x^* \\ y^* \end{pmatrix} = \begin{pmatrix} e \\ e \\ b \end{pmatrix}$$

and apply Gaussian elimination to transform it to reduced row echelon form. From this representation, we can extract the propagated conditions easily.

4.4 Multi-bit conditions

As indicated earlier, the bitwise generalized conditions do not describe all constraints on a difference perfectly. There may be additional linear conditions concerning multiple bits (not only one pair), such as "$x_1 = x_2$ and ${x_1}^* = {x_2}^*$". These constraints do show up as equations in the system constructed from f and $\Delta(x, x^*)$ if both f and $\Delta(x, x^*)$ are linear (and they are induced by f and not added artificially), but cannot be translated back to generalized differences. However, the information can sometimes still be used. In a guess-and-determine approach, these multi-bit conditions can be employed to improve the guessing strategy. Guessing variables that are involved in a large number of multi-bit conditions improves the propagation since it helps to determine the other involved variables [30]. Choosing such variables preferredly may reduce the total number of variables to be guessed before either a contradiction is encountered or a valid solution is found.

The following type of simple 2-bit conditions has already been proven to be useful in the cryptanalysis of the MD4-family of hash functions by Wang et al. [45]. While more complex conditions may also lead to useful information to propagate, keeping track of this information is usually too expensive.

Definition (Equivalence condition)**.** Let $(z_i, {z_i}^*)$ and (z_j, z_j^*) be two pairs of bits. Then, an equivalence condition on these two bit pairs is defined by the two equations

$$\begin{aligned} z_i + z_j &= c, \quad c \in \{0, 1\}, \\ z_i^* + z_j^* &= c^*, \quad c^* \in \{0, 1\}. \end{aligned}$$

If these two equations can be derived from a system of linear equations (or, in general, a function f and difference $\Delta(z, z^*)$) and neither bit i nor bit j is yet fully determined, we write $i \sim j$.

An alternative, weaker definition of equivalence conditions requires only at least one of the two equations.

This defines an equivalence relation between bits. In particular, the definition is transitive, so if bit i is equivalent to bits j and k, then bits j and k are also equivalent. If one variable in an equivalence class is guessed, all others in the class are also determined. Thus, we are interested in deriving the cardinality of the equivalence class for each bit from the linear system.

A class of r bits $z_1, \ldots, z_r$ is defined by the following $2r - 2$ equations in reduced row

echelon form:

$$\begin{array}{cccccccc}
z_1 & & & \oplus & z_r & = & c_1, & c_1 \in \{0,1\} \\
{z_1}^* & & & \oplus & {z_r}^* & = & c_1^*, & c_1^* \in \{0,1\} \\
& z_2 & & \oplus & z_r & = & c_2, & c_2 \in \{0,1\} \\
& {z_2}^* & & \oplus & {z_r}^* & = & c_2^*, & c_2^* \in \{0,1\} \\
& & \ddots & & \vdots & & & \vdots \\
& & z_{r-1} & \oplus & z_r & = & c_{r-1}, & c_{r-1} \in \{0,1\} \\
& & {z_{r-1}}^* & \oplus & {z_r}^* & = & c_{r-1}^*, & c_{r-1}^* \in \{0,1\}
\end{array}$$

Thus, it is easy to find the number of variables in an equivalence class using z_r as a representative of this class. Unfortunately, not all equations are necessarily explicitly contained in the reduced system. Some of the above equations may only be deducible as the sum of two equations of the reduced system. However, considering only equivalence conditions where at least one of the two necessary equations is explicitly found in the system can be done efficiently and already provides useful estimates for the class sizes.

Example 4.3. In Example 4.1, rows 3 to 8 of the reduced matrix correspond to equivalence conditions after adding row 9. They define the equivalence class $\{x_1, x_2, x_3, y_0\}$ with representative y_0. Thus, each of these variables is involved in 3 equivalence relations.

4.5 Nonlinear conditions

Unfortunately, not all generalized conditions can be expressed as linear equations. More specifically, the four conditions with three solutions each are not linear. Possibilities to handle these cases include dropping these constraints and allowing all four solutions or applying the guess-and-determine approach to add constraints that are only true for two out of the three solutions.

However, an alternative approach is to linearize the nonlinear equations and add an effort to resolve these equations when needed.

4.5.1 Linearization

The nonlinear generalized conditions can be described using nonlinear equations of degree 2, as shown in Table 4.2. These equations include the term $z \cdot z^*$.

To treat such nonlinear equations in a linear system of equations, new variables are introduced representing the nonlinear monomial, i.e. $\dot{z} = z \cdot z^*$. We add (essentially redundant) equations including $\dot{z}$ to linear conditions to link the new variables with the rest of the system. For example, if $z = 0$, then this implies $\dot{z} = 0$.

$\Delta(z_j, z_j^*)$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$	Nonlinear equation
E	–	×	×	×	$z_j \cdot z_j^* \oplus z_j \oplus z_j^* = 1$
B	×	–	×	×	$z_j \cdot z_j^* \oplus z_j^* = 0$
D	×	×	–	×	$z_j \cdot z_j^* \oplus z_j = 0$
7	×	×	×	–	$z_j \cdot z_j^* = 0$

Table 4.2: Equations for nonlinear generalized differences.

$\Delta(z_j, z_j^*)$	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$	Implications
#	–	–	–	–	contradiction
1	–	–	–	×	$\dot{z}_j = 1$
u	–	–	×	–	$\dot{z}_j = 0$
A	–	–	×	×	$\dot{z}_j \oplus z_j^* = 0$
n	–	×	–	–	$\dot{z}_j = 0$
C	–	×	–	×	$\dot{z}_j \oplus z_j = 0$
x	–	×	×	–	$\dot{z}_j = 0$
E	–	×	×	×	$\dot{z}_j \oplus z_j \oplus z_j^* = 1$
0	×	–	–	–	$\dot{z}_j = 0$
-	×	–	–	×	$\dot{z}_j \oplus z_j = 0$
3	×	–	×	–	$\dot{z}_j = 0$
B	×	–	×	×	$\dot{z}_j \oplus z_j^* = 0$
5	×	×	–	–	$\dot{z}_j = 0$
D	×	×	–	×	$\dot{z}_j \oplus z_j = 0$
7	×	×	×	–	$\dot{z}_j = 0$
?	×	×	×	×	no constraints

Table 4.3: Implications on $\dot{z}_j$ for generalized differences.

Table 4.3 lists additional equations for all generalized conditions. Some of these additional equations are in fact sufficient on their own to uniquely describe the condition; for example, the only solution to $\dot{z} = 1$ is $z = z^* = 1$.

As in the linear case in Section 4.1, we apply Gauss-Jordan elimination. To determine the generalized condition for bit z_i, we check all rows with pivots z_i, z_i^* or $\dot{z}_i$ and any linear combinations of these. Any equation containing only the variables z_i, z_i^* and $\dot{z}_i$ can be translated to a generalized condition.

Example 4.4. Let $f : \{0,1\}^4 \to \{0,1\}^4$ be as in Example 4.1,

$$y = f(x) = (x \lll 1) \oplus (x \lll 2) \oplus (x \lll 3),$$

and consider the input and output difference

$$\Delta(x, x^*) = \texttt{[E111]},$$
$$\Delta(y, y^*) = \texttt{[E???]}.$$

$$
\begin{pmatrix}
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\
1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & 1
\end{pmatrix}
\cdot
\begin{pmatrix}
x_0 \\ x_0^* \\ \dot{x}_0 \\ x_1 \\ x_1^* \\ \dot{x}_1 \\ x_2 \\ x_2^* \\ \dot{x}_2 \\ x_3 \\ x_3^* \\ \dot{x}_3 \\ y_0 \\ y_0^* \\ \dot{y}_0 \\ y_1 \\ y_1^* \\ \dot{y}_1 \\ y_2 \\ y_2^* \\ \dot{y}_2 \\ y_3 \\ y_3^* \\ \dot{y}_3
\end{pmatrix}
=
\begin{pmatrix}
\cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1
\end{pmatrix}
$$

Figure 4.3: Matrix for Σ and $\Delta(x, x^*) =$ `[E111]`, $\Delta(y, y^*) =$ `[E???]`, with linearized nonlinear conditions.

With the help of Tables 3.1 and 4.3, this is translated to the system in Figure 4.3. Gauss-Jordan elimination yields the system in Figure 4.4. This translates back to the propagated generalized conditions

$$
\begin{aligned}
\Delta(x, x^*) &= \texttt{[E111]}, \\
\Delta(y, y^*) &= \texttt{[1???]}.
\end{aligned}
$$

4.5.2 Remainder variables

Since the $\dot{z}$ variables are not yet linked with the function f and only appear in at most one equation, the above approach does not propagate better than without linearization.

Unlike y and y^*, it is not necessarily true that $\dot{y} = f(\dot{x})$. Consider, for example, the

$$
\begin{pmatrix}
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1
\end{pmatrix}
\cdot
\begin{pmatrix}
x_0 \\ x_0^* \\ \dot{x}_0 \\ x_1 \\ x_1^* \\ \dot{x}_1 \\ x_2 \\ x_2^* \\ \dot{x}_2 \\ x_3 \\ x_3^* \\ \dot{x}_3 \\ y_0 \\ y_0^* \\ \dot{y}_0 \\ y_1 \\ y_1^* \\ \dot{y}_1 \\ y_2 \\ y_2^* \\ \dot{y}_2 \\ y_3 \\ y_3^* \\ \dot{y}_3
\end{pmatrix}
=
\begin{pmatrix}
1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ \cdot \\ \cdot \\ 1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ 1 \\ 1 \\ 1
\end{pmatrix}
$$

Figure 4.4: Matrix for Σ and $\Delta(x, x^*) =$ `[E111]`, $\Delta(y, y^*) =$ `[E???]`, with linearized nonlinear conditions and after Gaussian elimination.

function f as defined in Examples 4.1 and 4.4. For the first output bit, we have

$$\begin{aligned}\dot{y}_0 &= y_0 \cdot y_0^* = (x_1 \oplus x_2 \oplus x_3) \cdot (x_1^* \oplus x_2^* \oplus x_3^*) \\ &= x_1x_1^* \oplus x_1x_2^* \oplus x_1x_3^* \oplus x_2x_1^* \oplus x_2x_2^* \oplus x_2x_3^* \oplus x_3x_1^* \oplus x_3x_2^* \oplus x_3x_3^* \\ &= \dot{x}_1 \oplus \dot{x}_2 \oplus \dot{x}_3 \oplus x_1x_2^* \oplus x_1x_3^* \oplus x_2x_1^* \oplus x_2x_3^* \oplus x_3x_1^* \oplus x_3x_2^* \\ &= \dot{x}_1 \oplus \dot{x}_2 \oplus \dot{x}_3 \oplus r_0.\end{aligned}$$

Introducing new linearized variables for all such mixed terms $x_i \cdot x_j^*$ would increase the number of variables quadratically, as can be seen in Table 4.4. To avoid this increase, we collect the mixed terms in remainder variables r_i: one additional variable for each output bit of f. The equations $\dot{y}_i = f_i(\dot{x}) + r_i$ are added to the system.

Applying Gaussian elimination may determine some of those remainder variables. The remaining ones need to be symbolically evaluated based on the existing information. Then, the generalized conditions can be read from the system as previously. However, the evaluation of nonlinear equations is in general very costly and a trade-off needs to be made.

In addition to the costs of evaluating the remainder variables, the linearized equation system is significantly larger than for the basic linear approach. Table 4.4 compares the system size to the simple linear system for an n-bit function f in terms of the number of variables and equations. The average number of equations to describe the generalized differences is calculated for uniformly distributed differences.

	number of variables	number of equations	
		f	$\Delta(z_j, z_j{}^*)$
no $\dot{z}_j$	$2(m+n)$	$2n$	$\leq 2(m+n)$
with $\dot{z}_j$	$3(m+n)+n$	$3n$	$\leq 3(m+n)+n$
all $z_iz_j^*$	$(m+n+1)^2-1$	$3n$	$\leq (m+n+1)^2$

Table 4.4: Equation system size for m-to-n-bit functions, with and without $\dot{z}_j$ linearization.

Example 4.5. We revisit Example 4.4. After adding the remainder variable equations to the system, we get the reduced system in Figure 4.5. To deduce any information about y_0, y_1 and y_2, we need to evaluate several remainder variables based on the knowledge we have so far.

$$\begin{aligned}r_0 &= x_1x_2^* \oplus x_1x_3^* \oplus x_2x_1^* \oplus x_2x_3^* \oplus x_3x_1^* \oplus x_3x_2^* \\ &= 1 \oplus x_3^* \oplus 1 \oplus x_3^* \oplus x_3 \oplus x_3 = 0 \\ r_1 &= x_0x_2^* \oplus x_0x_3^* \oplus x_2x_0^* \oplus x_2x_3^* \oplus x_3x_0^* \oplus x_3x_2^* \\ &= 1 \oplus x_3^* \oplus 1 \oplus x_3^* \oplus x_3 \oplus x_3 = 0 \\ r_2 &= x_0x_1^* \oplus x_0x_3^* \oplus x_1x_0^* \oplus x_1x_3^* \oplus x_3x_0^* \oplus x_3x_1^* \\ &= 1 \oplus x_3^* \oplus 1 \oplus x_3^* \oplus x_3 \oplus x_3 = 0.\end{aligned}$$

$$
\begin{pmatrix}
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot & 1 & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1
\end{pmatrix}
\cdot
\begin{pmatrix}
x_0 \\ x_0^* \\ \dot{x}_0 \\ x_1 \\ x_1^* \\ \dot{x}_1 \\ x_2 \\ x_2^* \\ \dot{x}_2 \\ x_3 \\ x_3^* \\ \dot{x}_3 \\ y_0 \\ y_0^* \\ \dot{y}_0 \\ y_1 \\ y_1^* \\ \dot{y}_1 \\ y_2 \\ y_2^* \\ \dot{y}_2 \\ y_3 \\ y_3^* \\ \dot{y}_3 \\ r_0 \\ r_1 \\ r_2 \\ r_3
\end{pmatrix}
=
\begin{pmatrix}
1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ \cdot \\ \cdot \\ 1 \\ \cdot \\ \cdot \\ 1 \\ \cdot \\ \cdot \\ 1 \\ 1 \\ 1 \\ 1 \\ \cdot
\end{pmatrix}
$$

Figure 4.5: Matrix for Σ and $\Delta(x, x^*) =$ `[E111]`, $\Delta(y, y^*) =$ `[E???]`, including remainder variables.

Using this information, the equations

$$y_0 \oplus y_0^* \oplus \dot{y}_0 \oplus r_0 = 1$$
$$y_1 \oplus y_1^* \oplus \dot{y}_1 \oplus r_1 = 1$$
$$y_2 \oplus y_2^* \oplus \dot{y}_2 \oplus r_2 = 1$$

all translate to the condition `E`. Thus, the propagated result is

$$\Delta^{\mathrm{opt}}(x, x^*) = \texttt{[E111]},$$
$$\Delta^{\mathrm{opt}}(y, y^*) = \texttt{[1EEE]}.$$

The same result is obtained by perfect propagation.

4.5.3 Limitations of the linearization approach

This approach of linearizing nonlinear constraints causes several problems that make it unattractive in practical implementations.

First, the large number of additional variables significantly slows propagation, while the benefit in propagation quality is limited. This is especially true in context of guess-and-determine attacks, where nonlinear conditions are relatively rare, as discussed in Chapter 5.

Second, the evaluation of remainder variables is cumbersome, and simpler (and faster) evaluation schemes fail in many cases. Linearizing all nonlinear terms of degree 2 explicitly, while easier to evaluate, makes the Gauss-Jordan elimination effort prohibitive.

Third, in the linearized system of equations, it is no longer true that logical consequences of a system of equations can be deduced as a linear combination of these equations. This limits the use of Gauss-Jordan elimination. To partially solve this problem, it may help to keep lists of implications and at certain times, such as after applying Gauss-Jordan elimination, feed additional implications of the existing equations to the system and iterate the elimination process.

Instead of linearizing nonlinear conditions, it is more practical to combine linear propagation with existing, nonlinear propagation methods such as bitsliced propagation to achieve imperfect, but relatively fast and good propagation for nonlinear conditions. This is further investigated in Chapter 5.

4.6 Nonlinear functions

Like nonlinear conditions, nonlinear functions could be treated by using linearization variables. And just like them, it is practically much more efficient to combine pure linear propagation with existing, non-algebraic methods. The results are not perfect, but

practically very usable. The basic idea is to propagate all linear parts of a function with linear propagation, and any nonlinear steps with bitsliced propagation from Section 3.5. For details on the implementation of this combination, we refer to Chapter 5.

To cover larger parts of the hash function with linear propagation, nonlinear functions can be broken down to isolate any nonlinear product terms and model the rest as a linear function. As an example, consider 2-bit modular addition, a nonlinear function. The example is again motivated by its use in the SHA-2 standard (with larger bitsize). Let a_1a_0 and b_1b_0 denote the function inputs, and r_1r_0 the result. Then the following equations hold:

$$\begin{aligned} r_0 &= a_0 \oplus b_0 \\ r_1 &= a_1 \oplus b_1 \oplus a_0 \cdot b_0. \end{aligned}$$

The carry bit $c_1 = a_0 \cdot b_0$ is responsible for the nonlinearity. To include this addition into a linear layer, we treat c_1 as an additional input variable. The linear model is then

$$\begin{aligned} r_0 &= a_0 \oplus b_0 \\ r_1 &= a_1 \oplus b_1 \oplus c_1. \end{aligned}$$

Externally, the function

$$c_1 = a_0 \cdot b_0$$

is propagated using bitsliced propagation. Any changes for c_1 trigger an update of the linear propagation layer, just like those of a_i, b_i and r_i.

For modular additions of more bits, the initial equations become slightly more complex since the carry has three instead of two inputs. However, this only concerns the carry definitions for the bitsliced part, not the linear model.

Chapter 5

Implementation

This chapter discusses the practical implementation of the linear propagation method proposed in Chapter 4. This includes useful algorithms and data structures, but also other design choices and optimization possibilities. The integration with existing tools and methods such as those described in Chapter 3 also plays a major role. While this chapter does contain remarks on performance of different design choices, the overall performance of different propagation methods is discussed in Chapter 6.

In the course of this thesis, two different implementations of linear propagation were done. The first is an isolated prototype used for basic experiments. It provides an interface to directly compare different propagation methods for random inputs. Some impractical methods, such as perfect propagation or experimental remainder resolution methods for nonlinear propagation, were only implemented in this prototype. The second implementation was integrated into an existing framework that was originally focused on bitsliced propagation [15, 30, 31]. This framework implements complete guess-and-determine attacks and thus provides a more realistic environment for evaluation. Experiments in this framework are based on input differences representative for an actual guessing tree, instead of the uniformly distributed values of the prototype. Both implementations are written in C++ and use data structures from the standard library.

The first two chapters describe the basic structure and purpose of the prototype and the search tool framework. Afterwards, Section 5.3 contains information about the implemention of bitsliced and perfect propagation for the prototype. Section 5.4 is dedicated to linear propagation and discusses different matrix formats and Gaussian elimination, as well as translation between equations and generalized conditions. Nonlinear environments are considered in Section 5.5. Finally, Section 5.6 discusses higher level topics, in particular related to the guessing strategy.

5.1 Prototype design

The purpose of the prototype implementation is to directly compare different propagation methods either for a single initial difference or for a larger number of randomly generated initial differences. The target functions are only linear, simple step functions, such as the Σ function from examples in Chapter 3 or the $\Sigma_0, \Sigma_1, \sigma_0$ and σ_1 functions from the SHA-2 compression function. The different propagation methods are compared in terms of their absolute and relative propagation quality and, to some extent, their runtime. Details on the figures of merit can be found in Section 6.1.

To provide easy extensibility, each propagation method is implemented as a derivation from a common superclass with a simple interface. A new propagation method class only needs to provide a default constructor, a name and a propagation function that maps an initial generalized difference to its propagated difference. Simple common data types and modification functions for generalized differences are provided. If one-time precomputation is necessary, such as the precomputed table for bitsliced propagation, it can be triggered from the constructor using static members or a singleton helper class.

When the main program is invoked, it simply applies all registered propagation methods in turn to either the specified initial generalized difference, or the specified number of randomly generated differences. The collected results can be printed in different formats, or the result quality plotted in a Ti*k*Z graph for TEX documents.

5.2 Framework design

The second implementation was integrated into the automated search tool used by Mendel, Nad and Schläffer for results on SHA-2 [30], based on work by De Cannière and Rechberger [15]. It is based on an initial implementation by the tool authors. This section describes the existing tool. Details on the implementation of linear propagation and different experiments and improvements attempted in the course of this thesis can be found in Section 5.4.

5.2.1 Steps

This search tool uses a guess-and-determine approach as described in Section 3.4 to find differential characteristics for several hash functions. For this purpose, the target hash function is modeled as a composition of its basic steps. During the search, each step is triggered to propagate separately whenever one of its associated variables is refined. The propagation method is defined for each step independently, so it is possible to use different propagation methods depending on the step type. The exact choice of steps and step sizes is not predetermined and depends, among others, on the intended propagation method.

5.2.2 Search

The underlying difference model uses generalized conditions to describe the knowledge about each bit. The general strategy is to first determine a suitable characteristic and then find a message pair conforming to this heuristic. These two phases are not strictly separated, since the second phase may give up on the current characteristic. In this case, the search returns to its first phase and restarts the second phase with a different characteristic. Also, the first phase already fixes some of the message bits.

Both phases employ a guess-and-determine strategy and backtrack in case of contradictions. In addition to generalized conditions, the tool authors define a number of additional checks especially for the application to SHA-2. The main task of the first phase is to determine `?` and `x`, while the second phase primarily determines `-`. The two phases are summarized in Algorithm 3, following the version suggested in [30] for SHA-2.

Algorithm 3 Two phases of the guess-and-determine attack as implemented in the search tool.

Phase 1

Guess: choose a bit with condition `?` or `x`. In case of `?`, guess `-`; in case of `x`, guess either `n` or `u`.

Determine: propagate this guess, then check for contradictions (in the generalized conditions and additional checks). In case of contradictions, start backtracking, else continue guessing.

Backtrack: change the last guess (`-` to `x` or `n` to `u` and vice versa), propagate and check for contradictions. If successful, continue guessing. Else, mark bit as critical and return to previous stages until the critical bit can be guessed without immediate contradictions.

Phase 2

Guess: choose a bit with condition `-` that is likely to propagate well (based on two-bit conditions). Guess `0` or `1`.

Determine: propagate this guess, then check for contradictions (in the generalized conditions and additional checks). In case of contradictions, start backtracking, else continue guessing.

Backtrack: change the last guess (`0` to `1` and vice versa), propagate and check for contradictions. If successful, continue guessing. Else, mark bit as critical and return to previous stages until the critical bit can be guessed without immediate contradictions. If this fails, return to phase 1.

During the search, the tool implicitly builds a guessing tree of previously guessed bits. Each node corresponds to a guessed variable. Its children represent the different possible guess decisions for this one variable, including the caused propagation effects. Previously encountered contradictions show up as leaves. Example 5.1 shows a simple miniature guessing tree for a toy characteristic of the Σ function used in various examples in Chapter 3.

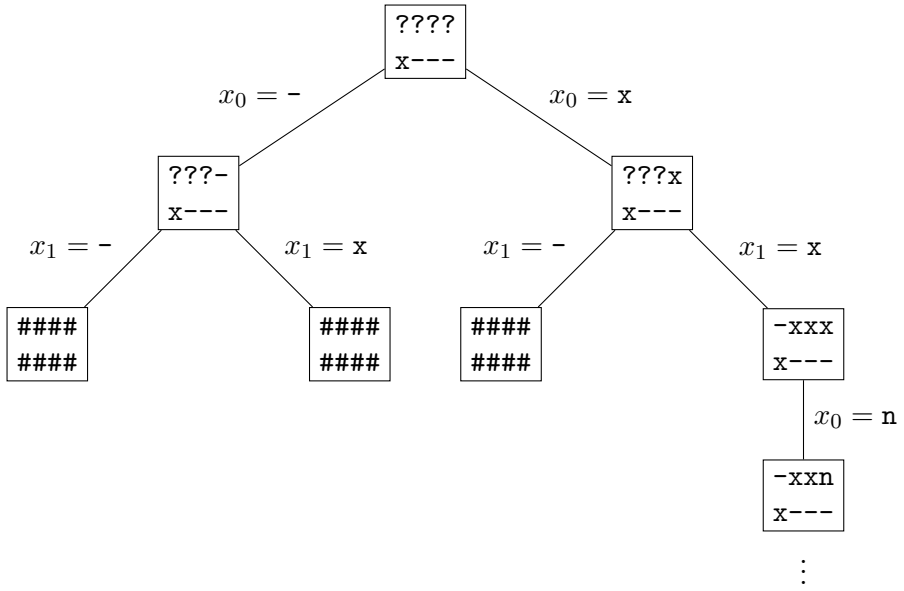


Figure 5.1: Guessing tree for Σ.

Example 5.1. We return to the Σ function with initial characteristic

$$\Delta(x, x^*) = \texttt{????},$$
$$\Delta(y, y^*) = \texttt{x---}.$$

A partial guessing tree for the first phase is given in Figure 5.1. The guessing algorithm will follow the current (rightmost) branch until it encounters a contradiction, and then return to one of the earlier nodes on the way back to the root node. Bitsliced propagation is used to propagate guesses.

To be able to return to earlier stages of the guess, i.e. to higher levels of the search tree later, we build a search stack of these previous stages. After every guess, the current state is saved on top of the stack with a certain probability. Not every single tree level is saved, as it is more efficient to backtrack in larger steps This is further discussed at the very end of this chapter, in Section 5.6.

5.2.3 Integration of linear propagation

There are a number of different possibilities to integrate linear propagation into the existing method. The straightforward idea would be to replace any existing (bitsliced) propagation steps by linear propagation. However, on the one hand, this is not always possible. Specifically, nonlinear steps are ill suited for linear propagation. On the other hand, bitsliced propagation may still be superior to linear propagation in some cases. The consequence is to replace only some propagation processes, or to only supplement propagation with linear methods to some degree.

The tool defines hash functions as a composition of steps. Each step links various intermediate variables, and a propagation method is associated with it. These steps do not need not describe disjoint operations during the hash computation. For example, it is possible to declare one step twice, and associate each instance with a different propagation method. The tool will then try to propagate using both methods, which increases the runtime, but may improve propagation. Also, it is possible to declare a large (linear) step that covers operations already represented by several smaller (bitsliced) steps.

An additional question is how to handle nonlinear conditions. While linear conditions can be translated directly for the linear propagation method, this is not a priori the case for nonlinear conditions. Section 4.5 presented a linearization method to describe nonlinear conditions, but in practice, other ways to handle nonlinear conditions proved more effective.

Below, several integration methods are briefly described. They are not mutually exclusive since they describe the interaction with bitsliced propagation on the one hand, and the treatment of nonlinear conditions and other implementation issues on the other hand. More detailed performance results can be found in Chapter 6.

- Bitsliced-only: Do not use linear propagation at all, but stick to bitsliced propagation like previously. This provides the fastest propagation (per propagated step), but the propagation quality is often poor.
- Bitsliced and linear: Perform all propagations twice, using bitsliced and linear propagation in parallel whenever possible, i.e. for all linear steps. This approach provides the best quality propagation, but is also the slowest of the proposed usage variants.
- Linear-only: Use only linear propagation whenever possible. Bitsliced propagation is only used for nonlinear steps. This is faster than parallel propagation, and the propagation quality is often not worse in practice. This version may perform worse for other search strategies where nonlinear conditions occur more frequently.
- Bitsliced and occasionally linear: Use bitsliced propagation for all regular propagation steps. Apply linear propagation only occasionally, for example when the bitsliced method fails to propagate any further, or even more rarely every few guesses only.
- Large linear layer: Use different step sizes for bitsliced and linear propagation. Larger linear steps support the target of less local propagation, but are also more costly to evaluate. Linear steps could even span multiple rounds of the function.
- Linearize parts of nonlinear steps: To integrate also larger nonlinear steps into linear propagation, split them into smaller nonlinear parts (with additional intermediate variables) that are in turn linked in a linear way.
- Linearize complete nonlinear steps: Use additional product variables to linearize

first-order nonlinear functions, then use these linearized versions for linear propagation. Due to the large number of necessary variables, this is not recommended.

- Linearize nonlinear conditions: Introduce product variables and possibly remainder variables to linearize nonlinear conditions.
- Drop or avoid nonlinear conditions: Since nonlinear conditions are relatively rare in practical search paths, simply treating them like `?` in linear propagation steps yields usable results. The rarity of these nonlinear conditions is a consequence of the searching and guessing strategy. Instead of generalizing them to `?`, it is also possible to specialize nonlinear conditions (with 3 solutions) to stronger linear conditions (with 2 solutions) by guessing them preferredly.

Details on the models of SHA-2 used for linear propagation can be found in Subsection 5.4.4.

5.3 Existing propagation methods

Since the prototype also implements existing propagation methods, and for better comparison with linear propagation, this section briefly describes the implementation of perfect and bitsliced propagation. The two methods were already introduced in Section 3.5. Perfect propagation is (locally) optimal within the limits of the corresponding step, but not globally optimal for the whole hash function consisting of several steps. For larger step (input) sizes, it quickly becomes infeasible. A practically very effective implementation of perfect propagation for very small step sizes is bitsliced propagation, which combines perfect propagation with table lookups of previously computed results.

5.3.1 Perfect propagation

This method relies on exhaustive enumeration of all possible inputs to a step. These inputs are limited by the initial generalized input conditions. Since generalized conditions for a bit limit which of the four possible bit pair values $(0,0),(0,1),(1,0),(1,1)$ are permissible, a convenient representation is the binary notation for subsets. A condition is encoded as a 4-bit value, where each bit position is assigned to one pair value. The bit positions that correspond to allowed pairs are set to 1, the others are 0. For example, the basic conditions $\mathtt{0},\mathtt{u},\mathtt{n},\mathtt{1}$ can be encoded as 0001, 0010, 0100, 1000. The exact correspondence between bit position and pair is not relevant for the implementation, as long as all conditions are translated consistently.

Using this encoding, checking whether a condition c_i is a refinement of c_j, $c_i \trianglelefteq c_j$, is a simple binary operation,

$$c_i \trianglelefteq c_j \quad \text{iff} \quad c_i \,\&\, c_j = c_i.$$

This operator can be generalized to multi-bit generalized differences (with the same number of bits) trivially,

$$[c_{n-1} \cdots c_1 c_0] \trianglelefteq [c'_{n-1} \cdots c'_1 c'_0] \quad \text{iff} \quad \forall\, i = 0, \ldots, n-1 : c_i \trianglelefteq c'_i.$$

The union of two conditions is the set of values permitted by at least one of the two conditions. This is also a simple binary operation,

$$c_i \cup c_j \leftarrow c_i \,|\, c_j.$$

For this operator, the extension to multiple bits is not as clear. The true union of the solution sets of two generalized differences is not necessarily writable as another generalized difference, as in Example 5.2. For this reason, we define the closed union $\overline{\cup}$ as the set closure of the union with respect to the generalized difference property. That is, the closed union is the finest generalized difference that contains the union of two generalized differences. Naturally, if the difference size is only 1 bit, the closed union equals the true union, $c_i \cup c_j = c_i \,\overline{\cup}\, c_j$. For multi-bit differences, the closed union is usually a strict superset of the true union. The computation of the closed union is again achieved with a simple logical or operation, since

$$[c_{n-1} \cdots c_1 c_0] \,\overline{\cup} [c'_{n-1} \cdots c'_1 c'_0] \leftarrow [(c_{n-1} \cup c'_{n-1}) \cdots (c_1 \cup c'_1)(c_0 \cup c'_0)].$$

Example 5.2. Consider the 4-bit differences $c = [\texttt{??--}]$ and $c' = [\texttt{??xx}]$. Their binary encoding is $c = (1111, 1111, 1001, 1001)$ and $c' = (1111, 1111, 0110, 0110)$. They describe the solution sets S, S' defined by

$$S = \{((x_3 x_2 x_1 x_0), (x_3^*, x_2^*, x_1^*, x_0^*)) : x_0 = x_0^* \wedge x_1 = x_1^*\},$$
$$S' = \{((x_3 x_2 x_1 x_0), (x_3^*, x_2^*, x_1^*, x_0^*)) : x_0 \neq x_0^* \wedge x_1 \neq x_1^*\}.$$

The true union of the two conditions, $c \cup c'$, is the set

$$S^{\cup} = \{((x_3 x_2 x_1 x_0), (x_3^*, x_2^*, x_1^*, x_0^*)) : (x_0 = x_0^* \wedge x_1 = x_1^*) \vee (x_0 \neq x_0^* \wedge x_1 \neq x_1^*)\}.$$

There is no corresponding generalized difference since solutions for bit positions 0 and 1 depend on each other. The closed union of the two is $c \,\overline{\cup}\, c' = [\texttt{????}]$, or binary $(1111, 1111, 1111, 1111)$, that allows any message pairs, since the condition without dependencies is the tautologic

$$(x_0 = x_0^* \vee x_0 \neq x_0^*) \wedge (x_1 = x_1^* \vee x_1 \neq x_1^*).$$

Using these two operators, the perfect propagation algorithm for a step function f with n-bit input conditions $c = [c_{n-1} \cdots c_1 c_0]$ and m-bit output conditions $a = [a_{m-1} \cdots a_1 a_0]$ is given in Algorithm 4. The n nested for loops are responsible for the exponential running time (in the number of input bits).

We only used perfect propagation for toy examples with up to 4 input bits and 4 output bits, to compare its performance with non-optimal methods. Also, it implicitly appears in bitsliced propagation implementations.

Algorithm 4 Perfect propagation

Input: step function f, input/output conditions $c = [c_{n-1} \cdots c_1 c_0]$, $a = [a_{m-1} \cdots a_1 a_0]$
Output: perfect input/output propagation $\overline{c}$, $\overline{a}$

```
c̄ ← #···## (n bits)
ā ← #···## (m bits)
for g_{n-1} ∈ {0, u, n, 1} such that g_{n-1} ⊴ c_{n-1} do
  for g_{n-2} ∈ {0, u, n, 1} such that g_{n-2} ⊴ c_{n-2} do
    for ... do
      for g_1 ∈ {0, u, n, 1} such that g_1 ⊴ c_1 do
        for g_0 ∈ {0, u, n, 1} such that g_0 ⊴ c_0 do
          x ← unique 1st message solution to g
          x* ← unique 2nd message solution to g
          y ← f(x)
          y* ← f(x*)
          h ← base generalized condition for message pair y, y*
          if h < a then
            c̄ ← c̄ ⊔ g
            ā ← ā ⊔ h
return c̄, ā
```

5.3.2 Bitsliced propagation

Bitsliced propagation further subdivides each step into substeps, or bitsliced steps, typically with only one output bit each. For each such bitsliced step, perfect propagation is applied. To avoid repetitive computations, each propagation is computed only once. Later calls with the same initial difference use stored values from previous calls.

If the number of bits for one bitsliced step allows it, it is practical to keep a table of all possible initial differences and the corresponding perfectly propagated resulting differences. The binary encoding of the initial difference can be used as a table index, and the resulting difference as the corresponding table entry. This table can be completely precomputed prior to the actual guess-and-determine attack. A simple array or standard library vector of differences is a suitable data structure. The propagation of the complete step is then reduced to a trivial table lookup for each output bit. A separate table is necessary for each different bitsliced step function.

Let f again be a step function with n bits input size and m bits output size. Then f consists of m bitsliced steps, and each output bit y_j can be written as $y_j = g(x_{i_1}, x_{i_2}, \ldots, x_{i_s})$. Here, the bitsliced step function g and the number s of the input indices $i_1, i_2, \ldots, i_s$ also depend on the output bit position j. For the function g, we need a table with 16^{s+1} entries of $4 \cdot (s+1)$ bits each. The precomputation of each entry using perfect propagation takes up to 4^s evaluations of g. Algorithm 5 illustrates this. Examples without separate precomputation can be found in Examples 3.6 and 3.7.

Algorithm 5 Bitsliced propagation, basic version

Precomputation:

for all bitsliced step functions g with s input bits **do**
  create table $T_g[0 \dots 16^{s+1} - 1]$
  for initial difference $c = [c_s \cdots c_0]$ with binary representation $0, \dots, 16^{s+1} - 1$ **do**
    $T_g[c] \leftarrow$ perfect propagation of input $[c_{s-1} \cdots c_0]$ and output $[c_s]$ for g

Propagation:

Input: step function f, input/output conditions $c = [c_{n-1} \cdots c_1 c_0]$, $a = [a_{m-1} \cdots a_1 a_0]$
Output: bitsliced input/output propagation $\overline{c}$, $\overline{a}$

$\overline{c} \leftarrow c$
$\overline{a} \leftarrow a$
repeat
  $\overline{c}^{\text{old}} \leftarrow \overline{c}$
  $\overline{a}^{\text{old}} \leftarrow \overline{a}$
  for $j \leftarrow 0, \dots, m-1$ **do**
    let g be the bitsliced step function for output bit j and $i_1, \dots, i_s$ the inputs for g
    $[\overline{a}_j \overline{c}_{i_1} \overline{c}_{i_2} \cdots \overline{c}_{i_s}] \leftarrow T_g[\overline{a}_j \overline{c}_{i_1} \overline{c}_{i_2} \cdots \overline{c}_{i_s}]$
until $\overline{c}^{\text{old}} = \overline{c}$ **and** $\overline{a}^{\text{old}} = \overline{a}$

The repeat-until loop is necessary in this implementation to account for situations where one bitslice updates a shared input variable and another, already propagated bitslice would propagate further with this refined input. Re-propagating all bitslices until nothing changes any longer is a very simple, but wasteful approach to handle such situations, especially for steps with many inputs. A more efficient, target-oriented implementation would only re-propagated bitslices whose input bits have changed. To achieve this, two additional data structures are useful. First, each bit needs a list of bitslices it is connected to. This list may also include bitslices from other, neighboring steps. Second, we need a list of bit positions or bitslices that still need to be updated. A queue, stack or set are all suitable data structures. Set operations are more expensive, but compared to queues and stacks, it avoids storing duplicates. Initially, this list contains all bitslices (or, in later iterations, all bitslices affected by the last guess). Iteratively, the first element of the list is chosen for propagation and deleted from the list. If any bits are refined during the propagation, all connected bitslices (except for the current one) are added to the list. Algorithm 6 summarizes the process. It only covers the propagation phase, since the precomputation phase does not change compared to the basic version.

For some input sizes s, it may be easily feasible to compute one table entry, but infeasible to precompute (or store) all entries in the table. For example, for $s = 16$, computing one entry takes up to $4^{16} = 2^{32}$ evaluations of g, which is still acceptable. However, there are $16^{17} = 2^{68}$ table entries in total, which is far too much to store. In such cases, it is better to use an incomplete, associative map from initial to resulting differences. Instead of precomputation, the map is filled whenever a new, not yet computed initial difference

Algorithm 6 Bitsliced propagation with update list

Input: step function f, input/output conditions $c = [c_{n-1} \cdots c_1 c_0]$, $a = [a_{m-1} \cdots a_1 a_0]$
Output: bitsliced input/output propagation $\overline{c}$, $\overline{a}$

```
c̄ ← c
ā ← a
create list L as queue, stack or set
add bitslices 0, ..., m − 1 to the list (by output bit index)
while L is not empty do
  c̄^old ← c̄
  ā^old ← ā
  select and delete an element j from L
  let g be the bitsliced step function for output bit j and i_1, ..., i_s the inputs for g
  [ā_j c̄_{i_1} c̄_{i_2} ··· c̄_{i_s}] ← T_g[ā_j c̄_{i_1} c̄_{i_2} ··· c̄_{i_s}]
  for bit positions b ∈ {ā_j, c̄_{i_1}, c̄_{i_2}, ..., c̄_{i_s}} do
    if b ≠ b^old then
      for all bitslices k connected to b, where k ≠ j do
        add k to list L
```

is encountered. Then, the perfect propagation is calculated and the result stored in the map. Theoretically, when adding a difference c and its propagation $\overline{c}$ to the map, it is also valid to add any differences c' such that $\overline{c} \trianglelefteq c' \trianglelefteq c$ to the map (with propagation $\overline{c}$). If the table becomes too full, old (or random) entries can be deleted. Such a lazy evaluation method was not necessary for the prototype, but is implemented similarly in the existing tool. Algorithm 7 sketches the method.

Algorithm 7 Bitsliced propagation with lazy evaluation

Precomputation:

for all bitsliced step functions g with s input bits **do**
create empty map M_g, mapping difference to difference

Propagation:

Input: step function f, input/output conditions $c = [c_{n-1} \cdots c_1 c_0]$, $a = [a_{m-1} \cdots a_1 a_0]$
Output: bitsliced input/output propagation $\overline{c}$, $\overline{a}$

$\overline{c} \leftarrow c$
$\overline{a} \leftarrow a$
repeat
$\overline{c}^{\text{old}} \leftarrow \overline{c}$
$\overline{a}^{\text{old}} \leftarrow \overline{a}$
for $j \leftarrow 0, \ldots, m-1$ **do**
let g be the bitsliced step function for output bit j and $i_1, \ldots, i_s$ the inputs for g
if map M_g contains key $[\overline{a}_j \overline{c}_{i_1} \overline{c}_{i_2} \cdots \overline{c}_{i_s}]$ **then**
$[\overline{a}_j \overline{c}_{i_1} \overline{c}_{i_2} \cdots \overline{c}_{i_s}] \leftarrow M_g[\overline{a}_j \overline{c}_{i_1} \overline{c}_{i_2} \cdots \overline{c}_{i_s}]$
else
$[\overline{a}_j \overline{c}_{i_1} \overline{c}_{i_2} \cdots \overline{c}_{i_s}] \leftarrow$ perfect propagation of input $[\overline{c}_{i_1} \overline{c}_{i_2} \cdots \overline{c}_{i_s}]$, output $[\overline{a}_j]$ for g
add $M_g[\overline{a}_j^{\text{old}} \overline{c}_{i_1}^{\text{old}} \overline{c}_{i_2}^{\text{old}} \cdots \overline{c}_{i_s}^{\text{old}}] \leftarrow [\overline{a}_j \overline{c}_{i_1} \overline{c}_{i_2} \cdots \overline{c}_{i_s}]$
until $\overline{c}^{\text{old}} = \overline{c}$ **and** $\overline{a}^{\text{old}} = \overline{a}$

5.4 Linear propagation

Linear propagation relies on linear algebra, in particular linear equations and matrix operations. The matrix implementation and the related algorithms are the core of this propagation method. There are several aspects to consider, such as:

- the matrix representation as a table of bits, e.g. in full array or sparse format.
- the Gaussian elimination process and other linear algebra operations.
- the translation process of generalized conditions and functions to binary equations and back.
- the integration into the guessing strategy, e.g. backtracking.

These aspects can be considered separately from each other for the most part. The following sections and subsections discuss different implementation ideas in approximately this bottom-up order. While this section covers a basic implementation for the simpler case of only linear functions and conditions, additional advanced and higher-level aspects are discussed in the following sections.

The central picture to keep in mind during the whole chapter is that we want to translate generalized conditions and step functions to binary linear equations, where each coeffi-

cient is either 0 or 1. Such an equation can be represented just by the list of coefficients, i.e. a binary string, and the whole equation system is a set of such binary strings, written in matrix form. Each row of the matrix is one equation, and each column contains all coefficients of one particular bit position. For propagation, this matrix is transformed to reduced row echelon form and then translated back to conditions. During the guessing process, additional rows are added to the matrix, and any consequences for the triangular form should be determined immediately and translated back.

5.4.1 Full matrix format

The most basic and natural data structure for matrices is an array of the values. In our case, a single bit is sufficient to represent each value. The C++ library offers `std::bitset` and `std::vector<bool>` for (one-dimensional) arrays of bits. It is also possible to manually build a similar structure, using `uint64[]` or `std::vector<uint64>` and bitwise access functions with bitmasks. This way, each array entry is used for 64 matrix entries. For n bits, $\lceil \frac{n}{64} \rceil$ array entries are necessary. While standard library implementations are usually preferable, the manual version offers speed advantages when operations are to be applied to all bits of a matrix row. Such row-wise operations are very frequent in our application. Examples include adding a row to another (using bitwise addition, i.e. xor), copying rows or clearing rows. Instead of operating on each bit separately, the manual version can operate on 64 bits at once, using native word operations. Both the prototype and the framework implementation use manual bit arrays with native wordsize values.

If `uint64[]` is used, this can also easily be extended to `uint64[][]` for not only one row, but a whole system of multiple rows. Alternatively, a single large `std::vector<uint64>` or `std::vector<bool>` can be used, and the offset for every new row can be calculated. However, this requires an explicit multiplication for each row access. Also, all these structures have one crucial disadvantage. In our application, rows are often exchanged, or new rows inserted between existing rows. It would be much more efficient if row exchanges would not require exchanging actual row contents. For this reason, we introduced an additional index vector that points to the actual memory location of each row. The row contents are all stored in one large `std::vector<uint64>`, in an order that does not necessarily reflect the current system order. Figure 5.2 illustrates the basic bitmatrix structure as implemented by the `bitmatrix` class in the tool. The row content is stored in one block per row in the `data` vector. The `matrix` vector of `Row`s serves as an index to the `data` vector. The `Row` data type contains a pointer or index to `data`, pointing to the start of the row's contents, as well as additional metadata (compare Subsection 5.4.3 below). To exchange two rows, it suffices to exchange the index entries.

Inserting new rows still requires to move all following index entries in `matrix`. This could be improved to constant complexity by using linked lists or similar instead of vectors for the index. However, linked lists perform far worse in more frequent operations, such as random access. A compromise between access and insertion performance (as well as a

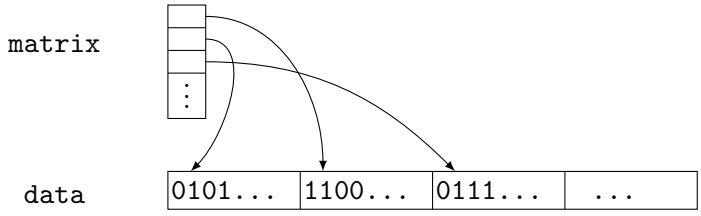


Figure 5.2: The `bitmatrix` data structure for equation systems

solution for the sorting task discussed in Subsection 5.4.3) is offered by various sorted trees, at the cost of significant data structure overhead. Also, none of the standard library tree implementations, such as `std::set`, supports random access in less than linear time. A useful tree implementation for our purpose would require both insertion and random access with logarithmic complexity. In practice, `std::vector<Row>` appears to be the best suitable choice, since insertions occur rarely compared to random access operations.

5.4.2 Sparse matrix format

One of the typical properties of the equations in our application is their sparsity. Especially for larger step sizes, only very few bits in every row of the matrix are set. There are a number of well-known data structures optimized for such sparse matrices since they are relatively well compressible. These structures require less memory than storing the full array and allow optimized multiplication and similar operations. In our case, reduced memory requirements would be especially useful since the matrix is often copied. In fact, copying matrix data shows up as one of the dominating runtime factors in profiling tools such as `cachegrind`. On the downside, sparse matrix formats are typically more complex and require far higher overhead per entry than a full format, even more so since each of our entries is only one bit in size.

Sparse matrix formats can be divided into two main groups. Formats of the first group allow easy construction and modification of the matrix, e.g. insertion and deletion of additional bits in some row. The second group is targeted for fixed matrices that are repeatedly involved in matrix operations such as addition or multiplication. In typical applications, sparse matrices are initially constructed using a format of the first type, and as soon as the matrix is complete and fixed, it is translated to a format of the second type for the actual operations. However, in our case, we do not intend to apply any typical matrix operations on fixed matrices. On the contrary, our matrices are modified unpredictably and element-wise throughout their lifetime. Thus, only the first group is relevant for our purposes. Also, some formats are optimized for numerical calculations, which is not useful for our binary matrices either. Generally, most widespread sparse matrix formats can be simplified since we do not need to store actual values for binary matrices, but only the information which bits are different from zero.

Typical formats of the "construction group" for sparse matrices include those offered for example by the Scipy Python library [40]. Using the library's terminology, three construction formats are the following:

- DOK, Dictionary of keys: this is a map from pairs of row and column index to the value at this index. Only nonzero elements are added to the map. For the binary case, the only relevant question for a given row and column index pair is whether this index was added to the map (bit is 1) or not (bit is 0). However, this format is not suitable for iterating over selected values in any order.
- COO, Coordinate list: this format stores a list of all nonzero values, and each entry contains a matrix value's row and column index and value. Compared to the DOK's map, it is easier to keep these entries sorted by row or column in order to improve iteration capabilities. Again, in our application, the value can be left out, so the structure contains only a list of row and column index pairs.
- LIL, List of lists: Similar to COO, this structure lists all nonzero entries, but instead of collapsing all entries into one list, the entries are separated by row (or column) index. Thus, the basic structure is a list with one element for each row. Each such element is again a list and contains the indices of all elements set in this row. Compared to COO, this improves the access time for elements of a specific row to constant. However, if many matrix rows are empty, LIL contains many unnecessary entries, and COO would be more memory-efficient.

Since our matrix contains only rows with at least one bit set (since empty rows are deleted or moved to the end of the matrix), LIL is clearly the most suitable of these structures.

Consequently, just like in the full bitmatrix format, our `sparsematrix` implementation contains a `std::vector<Row>` with one element for each row. Again, this is very useful to exchange rows. However, unlike the bitmatrix, `Row` no longer points to a full array of the row values. Instead, it contains a list of bits set in this row. We use a `std::set<int>` for this purpose, so that the bits of a row can be accessed in sorted order. Also, searching whether one bit is set works in logarithmic complexity (of the row size) this way.

Besides exchanging rows and setting single bits, there are two main operations we perform on our matrix. The first is to determine the first bit set in one row (and sort rows by this value). This is now trivially possible in constant time, without storing additional metadata like for the full bitmatrix. The second is to compute the xor addition of two rows. In terms of the bit index sets we store, this corresponds to the symmetric difference of the two sets. While this operation is not offered by the standard library, it is easy to implement a linear-time algorithm using the set's forward iterators since the sets are sorted. Like in the Mergesort merging algorithm, we need one iterator for each of the two sets. We always iterate the iterator with the smaller index so far, and add this index to the result if it is strictly smaller than the other. If the two are equal, the index is dropped. As soon as the index at the current iterator is larger, we change to

increase the other iterator. Since this is of linear complexity in terms of the number of bits in this row, it can theoretically be faster than the xor operations of the full matrix, but typically, it is slower since the involved basic constant operations are more complex and involve more conditional statements.

The main advantage of the sparse implementation should be its copy performance, since the compressed format usually needs less memory. However, runtime profiling showed that this is not the case. Part of the problem is the number of standard library structures used, and their organizational overhead. Since sets are not stored in contiguous memory blocks and use dynamic memory, copying sets performs worse than copying arrays, even if they are relatively small.

5.4.3 Gaussian elimination and ε-Gauss

To extract any information from our equation matrix, it is required to be in reduced row echelon form. The basic algorithm to transform a matrix to this form is the Gauss-Jordan elimination algorithm which was already listed in Algorithm 1. For the prototype elimination, this basic algorithm is sufficient and efficient. In the tool, only reduced and more specialized variants are implemented.

In a guess-and-determine attack, we typically work with a matrix changing over time, and we want to extract information at various times during its lifetime. Thus, we require our matrix to conform to reduced row echelon form almost always. For this purpose, we defined the `linearmatrix` class. It wraps a `bitmatrix` (or `sparsematrix`, since the two are derived from the same common matrix class and provide the same interface). After any modifications to this matrix, such as added rows, the class transforms the matrix to its reduced form.

The two defining properties of reduced row echelon form are the following:

- Row echelon: the rows are sorted by increasing pivot index, i.e. the index of the first bit set in each row. This implies that any bits in the quadrant to the lower left of each pivot element are zero.
- Reduced: no two pivot indices are equal, and more specific, if an index is the pivot index of one row, this column contains only zero elements in all other rows. This means each pivot element is the only nonzero element in its column.

Instead of always applying the full Gauss-Jordan elimination algorithm, we try to check as few rows as possible to restore this state after local modifications. For this purpose, we declare any modified rows as dirty (or touched, or updated) and add them to the dirty set. Treating a dirty row is basically the same as adding a new row to the system. This treatment consists of three phases:

1. Reduce new row: if any of the bits of the new row are also pivot elements of an existing row, these need to be eliminated by xor-adding the conflicting existing

row to the new row. Only existing rows with pivot larger than the new row need to be checked. The exact order in which the rows are checked makes no difference.

2. Reduce existing rows, backsubstitution: Now, the new row's pivot needs to be eliminated in all existing rows. For this, xor-add the new row to any existing rows in which its pivot is set. Only existing rows with pivot less than the new row need to be checked, in arbitrary order. If any existing row's pivot element is changed in this procedure, mark this row as dirty. If only non-pivot elements are changed, the row is not dirty.

3. Sort: Now, the new row is ready to be inserted into the system at the correct position, between the largest lesser and the least larger existing row pivot.

We refer to this basic procedure as ε-Gauss algorithm.

In practice, this can be further reduced since only very specific types of rows are added or marked dirty in our application. Rows marked dirty in the above procedure are already reduced (since no pivot-conflicting bits are ever added to the system), so the first phase can be dropped. The completely new rows added to the system, on the other hand, are very sparse, with only very few bits set. For this reason, it is more effective to selectively pick only rows from the existing whose pivot occurs in the new row and check those for conflicts in the first phase. Since searching these rows takes logarithmic time on average (because the rows are sorted by their pivot), it is faster to search a few elements than to iterate over all.

It is also possible to perform several modification steps at once and only then re-sort rows to return them to their pivot-sorted state. In this case, a standard sorting algorithm can be applied to re-sort rows. For this purpose, stable sorting algorithms with good best-case behaviour are preferable. In several versions of our implementation, this seemingly minor task was responsible for a respectable percentage of the total runtime. Switching from standard QuickSort to the usually inferior InsertionSort improved this behaviour considerable and cut the sorting runtime to a fraction of the previous time.

5.4.4 Linear function translation

The purpose of the matrix structure we described in the previous subsections is to hold equations from two sources: First, from the linear step function, and second, from the linear conditions. The function equations are fixed in number (depending on the output bitsize) and added only once during the initialization of each matrix. Choosing the exact step function definitions is a central performance factor. Smaller step functions imply a larger number of smaller matrices, which is beneficial for the performance of the local propagation steps. On the other hand, they deteriorate the global propagation effect of local guesses and propagations. If smaller substeps are combined into one larger step, the matrix grows quadratically, but we get one better global propagation result instead of several local ones.

In the case of SHA-2, suitable steps include the 32-bit or 64-bit $\Sigma_0, \Sigma_1, \sigma_0$ and σ_1 functions. The f_0 and f_1 functions and modular additions are not linear. However, they can also be partially described by linear equations as discussed in Section 4.6. Other operations, such as word permutations, do not need to be translated to equations, but should instead be mirrored in how the individual steps are connected. Alternatively, one whole round of SHA-2 (step update function), including all the above, can be (partially) translated to one large linear step.

For Keccak, the hash function definition also already suggests natural step functions, namely θ, ρ, π, χ and ι. Of these, only χ is nonlinear. The remaining steps form one very large linear layer that can be directly translated. In the standardized SHA-3 version, the internal state consists of 1600 bits, so the matrix is considerably larger than for SHA-2.

As soon as the step functions are defined, the translation to linear equations is straightforward, as described in Section 4.2. A function with input bitsize m, output bitsize n and no explicit internal variables requires a matrix with $2 \cdot (m+n)+1$ columns and $2 \cdot n$ rows. Any additional explicit internal variable increases the number of variables (and thus columns) by 2.

Instead of the intuitive variable order of Section 4.2, we recommend that the leftmost columns correspond to the output variables y, y^*, while the columns to the right are used for input and intermediate variables x, x^*. This way, it is easy to directly construct the matrix in reduced row echelon form instead of having to transform it. For the construction, it is not important whether the z, z^* variables are arranged as $z_0, z_0^*, z_1, z_1^*, \ldots$ or $z_0, z_1, \ldots, z_0^*, z_1^*, \ldots$. However, while the second version as used in Section 4.2 allows a convenient block matrix notation, the first version improves the performance of reading results from the matrix, see next subsection. For this reason, we chose the first version, so the final order of variables is

$$y_0, y_0^*, y_1, y_1^*, \ldots, y_{n-1}, y_{n-1}^*, x_0, x_0^*, x_1, x_1^*, \ldots, x_{m-1}, x_{m-1}^*.$$

Algorithm 8 describes how the matrix can then be constructed.

5.4.5 Linear condition translation

During the guess-and-determine attack, any generalized conditions on variables involved in one step are also added to the matrix as linear equations. The definition of these equations can be directly taken from Table 4.1. Adding the equations to the matrix increases its size and requires an adequately dynamic matrix structure. However, it is not always necessary to increase the matrix dimensions with new equations. Instead, it is sufficient to apply only the first two phases of the ε-Gauss algorithm, and skip the sorting phase. The reason for this is that implementations like the search tool also keep track of the generalized conditions explicitly outside the matrix. Combinations of two generalized conditions on the same bit are also evaluated externally.

Algorithm 8 Constructing the linear function equation matrix.

Input: Linear function $f : \{0,1\}^m \to \{0,1\}^n$
Output: Linear equation matrix A for f with $2 \cdot (m+n)+1$ columns and $2 \cdot n$ rows
  Variable order $y_0, y_0^*, y_1, y_1^*, \ldots, y_{n-1}, y_{n-1}^*, x_0, x_0^*, x_1, x_1^*, \ldots, x_{m-1}, x_{m-1}^*$
  $A[0, \ldots, 2n-1][0, \ldots, 2n-1] \leftarrow I$, the identity matrix
  $A[0, \ldots, 2n-1][2n, \ldots, 2(n+m)] \leftarrow O$, the zero matrix
  $e \leftarrow f(00\cdots0)$
  for $i = 0, \ldots, m-1$ **do**
    $b_i = 0\cdots010\cdots0$ where only bit i is 1
    $A[0, 2, 4, \ldots, 2n-2][2n+2i] \leftarrow f(b_i) \oplus e$
    $A[1, 3, 5, \ldots, 2n-1][2n+2i+1] \leftarrow f(b_i) \oplus e$
  $A[0, 2, 4, \ldots, 2n-2][2n+2m] \leftarrow e$
  $A[1, 3, 5, \ldots, 2n-1][2n+2m] \leftarrow e$

Example 5.3. Consider the following small example for illustration. Assume our matrix contains the equation

$$z_i \oplus z_j = 0,$$

and we guess first the generalized conditions $z_i =$ `-`, then $z_j =$ `5`.

With a dynamically growing matrix, the propagation of these conditions would work as follows:

1. Propagate $z_i =$ `-` with equation $z_i \oplus {z_i}^* = 0$:
   (a) reduce new row to ${z_i}^* \oplus z_j = 0$ by adding the matrix row with pivot z_i
   (b) no backsubstitution possible
   (c) add reduced new row to matrix to obtain the new matrix

$$\begin{aligned} z_i \oplus z_j &= 0, \\ {z_i}^* \oplus z_j &= 0. \end{aligned}$$

2. Propagate $z_j =$ `5` with equation $z_j = 0$:
   (a) no reduction of new row possible
   (b) backsubstitute new row to obtain the matrix

$$\begin{aligned} z_i &= 0, \\ {z_i}^* &= 0. \end{aligned}$$

   (c) add reduced new row to obtain

$$\begin{aligned} z_i &= 0, \\ {z_i}^* &= 0, \\ z_j &= 0. \end{aligned}$$

3. Extract conditions $z_i =$ `0` and $z_j =$ `5`.

With a fixed-size matrix and externally stored generalized conditions, on the other hand, we have the following:

1. Propagate $z_i =$ `-` with equation $z_i \oplus z_i{}^* = 0$:
   (a) reduce new row to $z_i{}^* \oplus z_j = 0$ by adding the matrix row with pivot z_i
   (b) no backsubstitution possible, matrix unchanged
2. Propagate $z_j =$ `5` with equation $z_j = 0$:
   (a) no reduction of new row possible
   (b) backsubstitute new row to obtain the matrix

$$z_i = 0.$$

3. Extract condition $z_i =$ `5` and combine this with the stored condition $z_i =$ `-` to obtain $z_i =$ `0`. No changes for $z_j =$ `5`.

To extract conditions from an equation system, it is useful to order variable pairs z_i and z_i^* side by side in the matrix. Then, it is possible to iterate linearly through the list of equations. For each new equation, we peek ahead at the next row, and if its pivot is the pair partner of the current row's pivot, both rows together define one generalized condition. If an equation has only variables for one bit position set, i.e. if $z_i = 1$ and no other variable except z_i^* has coefficient 1 in the current row, then it can be translated back to generalized conditions. For a pair of rows, it is either possible that each can be translated back separated, or that their xor sum can be translated, or that they are not translatable at all.

An anonymous reviewer of [19] suggested to further split the equations for one step into three matrices. This idea is based on the observation that each initial equation, whether from the function definition or the conditions, contains either only z or only z^* variables, or the sum $z \oplus z^* = z^+$. Storing all z equations in one matrix (with only the corresponding variables), all z^* equations in a second and the z^+ equations in a third matrix would effectively reduce the required memory to half of the original. This is because the total number of equations stays the same, but each matrix has only $m + n + 1$ instead of $2(m + n) + 1$ columns. Of course, this calculation is only correct for the full bitmatrix representation. A sparse implementation hardly profits from a reduced number of columns (while keeping the same number of set bits per row, except for z^+). In spite of the doubtless improvement for matrix sizes, we expect no significant advantages for the overall process since such a matrix separation complicates the propagation process. After all, the z^+ variables are not independent from z and z^*, and the information from the three matrices still needs to be combined in some manner to detect contradictions and propagations. For this, we again need to construct equations with the full number of $2(m + n) + 1$ variables. We expect this would run into problems similar to the linearization of nonlinear conditions. It is nevertheless possible that the ε-Gauss algorithm could be improved for this separated matrix storage format. The special equation system structure with two separate submatrices for z and

z^*, coupled only by the simple z^+ equations, is certainly worth to examine for specialized Gaussian elimination implementations. However, we attempted no implementations in this direction.

Summarizing, the additional existing data structures of the tool framework allow to store only parts of the equation system and still retain perfect propagation for linear generalized conditions. Additionally, the special equation system structure deserves further investigation.

5.5 Nonlinearity

The method described in the previous section achieves perfect propagation of steps as long as initial conditions and the step function are (affine) linear. To handle also nonlinear conditions and nonlinear functions to some extent, additional extensions are necessary.

5.5.1 Nonlinear functions

Since our propagation method is inherently based on linearity, it is a priori not possible to apply it to other functions. However, there are two (closely related) possibilities to handle such nonlinear functions. The first is to linearize the nonlinearity by introducing new variables for all nonlinear combinations of existing variables. For example, many hash functions only involve nonlinear functions of degree 2. Then, they can be described as linear functions of the variables x_i and $x_i \cdot x_j$. However, since this requires a huge (quadratic) amount of additional variables (and thus matrix columns), we refrained from implementing such linearized variants.

Instead, we modified the step descriptions of hash functions to achieve the maximum possible linear steps. For example, instead of classifying the complete modular addition block as a nonlinear function, we considered it a linear function of the actual inputs and additional inputs, the carry variables. This is possible since only the carry variables depend directly nonlinearly on the actual inputs. The carry variables are also defined as outputs of the bitsliced step description. Thus, the bitsliced propagation results are used to improve the linear propagation of the complete function.

5.5.2 Nonlinear conditions

Handling nonlinear conditions is not as essential to the usefulness of linear propagation as nonlinear functions. While nonlinear functions are integral parts of hash functions and must always be included in the propagation process in some way, nonlinear conditions only provide a slightly refined description of generalized conditions compared to the linear `?` condition. Thus, completely ignoring nonlinear conditions and treating

them like ? only leads to slightly less effective, but still very useful propagation. Additionally, nonlinear conditions appear relatively rarely compared to linear conditions due to the guessing strategy. Nevertheless, it is certainly worth using the information from nonlinear conditions whenever easily possible.

An easy measure is to translate nonlinear conditions as ? for the matrix, but iterate the propagation process if necessary. If the input difference contained any nonlinear conditions with three solutions, the result of the linear propagation process is not necessarily a refinement of the initial difference since it may allow the fourth solution as well. If the extracted linear propagation result allows the fourth solution and at most one additional solution for the nonlinear input condition, combining this information narrows the solutions for this bit down to zero or one solution. This combined result is then again linear and finer than the current matrix contents. Thus, feeding it back to the matrix and repeating the elimination process may cause additional propagation.

A more thorough approach is the introduction of additional variables for all nonlinear terms $x_i x_j^*$, as suggested in Section 4.5. The quadratic number of additionally necessary variables makes this method unpractical. In addition to the usual $2(m+n)$ variables, we would need another $(m+n)^2$ variables for a full linearization of all mixed $x_i x_j^*$ terms.

Instead, we implemented the remainder variable strategy also suggested in Section 4.5 in the prototype. We refrained from employing it in the tool since a combination with bitsliced propagation seems more promising, both in terms of runtime performance and quality of the propagation. This idea requires only $m+n$ linearization variables and n remainder variables $\dot{x}_i$ in addition to the basic $2(m+n)$ variables, but introduces the additional challenge of evaluating these remainder variables. Basically, we want to deduce information on the remainder variables from information on the normal bits. This is very similar to a one-directional propagation with all normal bits as inputs and the remainder variables as outputs. However, each remainder variable depends on both parts of the input message pair, but does not define a pair itself. Note that in spite of this additional insecurity, this approach never propagates worse than normal linear propagation. The matrix contains all lines of the original matrix, plus some addition lines that include the remainder variables. Even if we derive no information about the remainder variables at all, the worst possible propagation result is the same that we would get from normal linear propagation.

There is one remainder variable for each output bit of the linear function. Assume that one output bit is a linear combination of several input bits. For simplicity of notation, we will collect both input and output variables into one variable pool, indexed as z_i. Let z_k be the output bit, and I_k the set of all variables involved in the equation for z_k, i.e.

$$z_k = \sum_{i \in I_k \setminus \{k\}} z_i.$$

Then, we have $|I_k|$ different, equivalent definitions for the remainder variable r_k, where

each definition contains all but one bit of I_k:

$$\forall i \in I_k : \qquad r_k = \sum_{\substack{j \in I_k \\ j \neq i}} \sum_{\substack{j' \in I_k \\ j' \neq j,i}} z_j z_{j'}^*.$$

For example, for our toy 4-bit Σ function, we have 4 remainder variables. Each of them involves 4 variables, so each has 4 representations, each representation containing 3 of the 4 involved variables.

These remainder variable definitions are not part of the equation system, since they are nonlinear with respect to the other variables. Instead, they must be encoded elsewhere. Since each remainder variable and all its representations are uniquely defined by the set of involved variables I_k, this is a first idea how to store them. Each remainder variable is stored as an $m + n$-bit string, where each bit represents one variable that can be involved. The string serves as a member list of the set I_k and is set to zero except for the bit positions corresponding to members of I_k. If we obtain information about the variables in I_k, we may be able to determine r_k and thus simplify the matrix equations that contain r_k. For example, if all except one variable of I_k are fully determined, the value of the remainder variable can be deduced by a few basic logical operations on the I_k string.

Unfortunately, this simple representation is only useful for a small number of cases, and only rarely better than linear propagation. For better results, a representation closer to the full linearization is necessary. We suggest to supplement the list of involved variables with a table of the exact terms. To represent terms z_i, z_i^* and $z_i z_j^*$, a square bitmatrix can be used:

$\cdot$	1	z_0	z_1	$\cdots$
1				
z_0^*				
z_1^*				
$\vdots$				

For all terms occurring in one possible representation for r_k, the corresponding cell in the square is set to 1. Any of the actual representations of r_k can be obtained by dropping one variable column and the row of the same index.

Example 5.4. We return to our Σ example, with input bits $z_0, \ldots, z_3$ and output bits $z_4, \ldots, z_7$. Consider the output bit

$$z_7 = z_2 \oplus z_1 \oplus z_0.$$

The associated remainder variable r_3 has four possible representations,

$$\begin{aligned} r_3 &= z_2 z_1^* \oplus z_2 z_0^* \oplus z_1 z_2^* \oplus z_1 z_0^* \oplus z_0 z_2^* \oplus z_0 z_1^* \\ &= z_7 z_1^* \oplus z_7 z_0^* \oplus z_1 z_7^* \oplus z_1 z_0^* \oplus z_0 z_7^* \oplus z_0 z_1^* \\ &= z_7 z_2^* \oplus z_7 z_0^* \oplus z_2 z_7^* \oplus z_2 z_0^* \oplus z_0 z_7^* \oplus z_0 z_2^* \end{aligned}$$

The involved variables are $I_3 = \{7, 2, 1, 0\}$. The square representation is

$\cdot$	1	z_7	z_2	z_1	z_0
1	–	–	–	–	–
z_7^*	–	–	1	1	1
z_2^*	–	1	–	1	1
z_1^*	–	1	1	–	1
z_0^*	–	1	1	1	–

Generalized conditions known for the involved variables can be used to simplify this square. If only one row and the corresponding column are left, the remainder variable is known to evaluate to 0 (or to 1 if the entry for $1 \cdot 1$ is set).

Square update rules include the following:

- `0`: if $z_i = z_i^* = 0$, delete the row and column for z_i, z_i^*. In the simpler I_k notation, delete i from I_k.
- `1`: if $z_i = z_i^* = 1$, delete the row and column for z_i, z_i^*. For any previous entry $z_i \cdot z_j^*$, toggle the entry $1 \cdot z_j^*$; and vice versa for $z_j \cdot z_i^*$.
- `n`: if $z_i = 0$ and $z_i^* = 1$, delete corresponding row and column. For any deleted $z_j \cdot z_i^*$, toggle $z_j \cdot 1$.
- `u`: if $z_i = 1$ and $z_i^* = 0$, delete corresponding row and column. For any deleted $z_i \cdot z_j^*$, toggle $1 \cdot z_j^*$.
- `5`: if $z_i = 0$, delete column.
- `A`: if $z_i = 1$, delete column. For any deleted $z_i \cdot z_j^*$, toggle $1 \cdot z_j^*$.
- `3`: if $z_i^* = 0$, delete row.
- `C`: if $z_i^* = 1$, delete row. For any deleted $z_j \cdot z_i^*$, toggle $z_j \cdot 1$.
- `-` and `-`: if $\Delta(z_i, {z_i}^*) =$ `-`, i.e. $z_i \oplus z_i^* = 0$, and if additionally for some $j \neq i$, $\Delta(z_j, {z_j}^*) =$ `-`, i.e. $z_j \oplus z_j^* = 0$, delete entries $z_i \cdot z_j^*$ and $z_j \cdot z_i^*$ (only if both are set).
- `x` and `-`: if $\Delta(z_i, {z_i}^*) =$ `x`, i.e. $z_i \oplus z_i^* = 1$, and if additionally for some $j \neq i$, $\Delta(z_j, {z_j}^*) =$ `-`, i.e. $z_j \oplus z_j^* = 0$, delete entries $z_i \cdot z_j^*$ and $z_j \cdot z_i^*$ (only if both are set) and toggle either z_j or z_j^*.

Even if the remainder variable cannot be explicitly evaluated, it may be possible to eliminated it. For example, if only terms $1 \cdot z_i^*$ and $z_i \cdot 1$ are left, it is possible to simply choose one of the representations and add all terms to all equations in the matrix that contain r_k, since all remaining terms are linear. At a certain point, square update rules also become more cumbersome to evaluate than simply brute-forcing all involved variables (constrained by their generalized conditions) and checking if by chance all inputs result in the same remainder variable value.

The synchronization of the square table with the matrix equations is a tradeoff between the extremes of complete linearization (with a larger number of variables, where all remainder variable terms are directly included in the matrix) and no remainder variable evaluation at all (which is very similar to basic linear propagation). However, note that even complete linearization does not guarantee perfect propagation of nonlinear conditions. This is because the nonlinear coupling of basic terms z_j and nonlinear product variables is not part of the equation system and needs to be handled explicitly externally. For example, if we discover $z_j = 0$, we can also set all product variables $z_j z_i^*$ to 0. A set of useful toy examples to compare different square update rules and synchronization rules is obtained from the Σ function, combined with initial generalized differences $*7*?$ for the input bits, and `????` for the output bits, where each $*$ is one of $\{0, 1, n, u\}$.

We repeat at this point that linearization and remainder variables are in our opinion not a promising extension of basic linear propagation. We think that the combination of bitsliced and basic linear propagation is a more effective and efficient method. Furthermore, nonlinear conditions do not play an important role in practical guess-and-determine search trees. The uniformly distributed inputs we use for some experiments over-represent nonlinear conditions in comparison with practical search trees.

5.6 Guessing strategy

On a higher level, two additional improvements were implemented. The first concerns the choice of variables for guessing. The existing strategy favors variables that are likely to propagate well. This estimation of propagation effects can be supplemented with information from the linear equation system. The second attempts to improve performance by avoiding many matrix copies. These copies are necessary to return to earlier stages of the guessing tree. Instead of full copies, we introduce incremental backups that can be undone and re-applied.

5.6.1 Two-bit conditions

In the existing tool, the choice of variables to be guessed is based on the expected propagation effects of this guess. Naturally, we want to first guess crucial, central variables with large propagation effects. With this choice, we hope to detect contradictions much earlier than with a random choice. If a variable is linked closely to several other variables, for example in equations like $z_i = z_j$, guessing this variable will also effect the linked variables. This idea was already discussed in Section 4.4. In a nutshell, we want to determine for each variable in how many "intimate relationships" with only one additional variable it is. The equations in question are often, but not always explicitly contained in the reduced row echelon matrix. These relationships, or two-bit conditions, are transitive in nature, as described in Section 4.4. If z_i is linked with z_j, and z_j is linked

with z_k, then z_i and z_k are also as closely related, even if the corresponding equation is not explicitly part of the system.

In graph terminology, let each bit position i (or variable pair z_i, z_i^*) be represented by a node in a graph. We scan through the matrix for any two-bit equations, and link the two variables in question by an edge. The resulting connected graph components define sets where each node is closely connected to each other node in the same component, but to no node of any other component. To reflect this, we extend every connected component to a full clique by interconnecting any pair of nodes in the component. Then, the number of neighbors of each node characterizes its expected propagation effect. The more neighbors, the better the expected propagation, and the earlier this variable should be guessed.

It is not necessary to explicitly add all additional edges to complete the clique. Instead, it is sufficient to determine the size of each node's connected component. This can be achieved more efficiently by using a Union-Find data structure, such as a Fibonacci heap. We implemented this approach using the `boost::disjoint_sets` structure. We start by assigning each variable to its own single-element set. Then we scan the full matrix for equations of the type $z_i \oplus z_j \in \{0, 1\}$ or $z_i^* \oplus z_j^* \in \{0, 1\}$. For each such equation, we merge the disjoint sets that currently contain bit positions i and j. Afterwards, the propagation coefficient of each variable can be found by efficiently searching the component that contains the variable, and reading its cardinality.

5.6.2 Backtracking

One of the dominating runtime factors of our standard linear propagation implementation is matrix copying. Such copies are made before guessing each variable, in order to be able to undo the guess later by reverting to the previous, stored matrix. To avoid copying the complete matrix so often, we implemented a backtracking mechanism that applies and undoes changes in one persistent matrix. Instead of the complete matrix, only the list of changes needs to be stored. Only occasionally, we store a complete copy of the matrix as a reference in the search stack.

For this purpose, each possible modifying operation is encapsulated in a dedicated class, similar to the command pattern. All operations are derived from the same base class `Change`, but have their own parameters as members. Required methods are `Apply` and `Undo`. The three central operations are the following:

- `XorBit`: a single bit is flipped. Parameters are the row and column index of the target bit. This operation is an involution function, so applying it is equivalent to undoing it.
- `XorRow`: one row is xor-added to another. Parameters are the two row indices, and this modification is also an involution.

- `Sort`: a permutation is applied to the rows of the matrix. Parameter to this modification is the permutation itself, stored in one-line notation in an array. To undo the modification, the inverse permutation needs to be determined and applied.

A list of applied modifications is collected on a stack of `Change`s. Any modifying (non-`const`) methods of the matrix class need to add the corresponding items to the stack when called. To return the matrix to the original values, or any previous state, the `Change` items on the stack need to be undone in inverse order (from the top of the stack), and deleted from the stack.

However, this simple, linear history is not sufficient for our situation. When searching through the guessing tree, the decision process for each target variable works as in Algorithm 9.

Algorithm 9 Guessing procedure for one variable, without incremental backtracking

Input: matrix for current state of conditions, variable to guess with two alternatives
Output: matrix for refined conditions, with variable guessed

```
create backup of current state, including the matrix
guess alternative 1 and propagate
if alternative 1 caused no immediate contradictions then
  decide randomly whether to stack alternative 2 in case the branch fails later
  if yes then
    guess alternative 2 in backup state and propagate in backup
    if alternative 2 caused no immediate contradictions then
      push the modified backup to the search stack
  done, continue with next variable
else
  guess alternative 2 in backup state and propagate in backup
  if alternative 2 caused no immediate contradictions then
    overwrite the current state with the modified backup and continue from there
    done, continue with next variable
  else
    mark the current variable as problematic
    return to an earlier alternative branch from the search stack
```

The matrix copy we want to avoid is the backup created in the very beginning. The occasional matrix copies for the search stack are comparably harmless for the performance, so we are not concerned with them. The backup is relevant for two cases:

- If alternative 1 is bad, we want to return to the original state to try alternative 2. In this case, we forget the changes made for alternative 1.
- If alternative 1 is fine, we sometimes want to quickly try alternative 2 from the original state as well before pursuing alternative 1 any further.

The first case can be handled with the previously described stack of recent operations and the `Undo` command. If the first alternative fails, we undo all changes caused during the propagation of alternative 1 and continue with alternative 2. The undone changes can be dropped and need not be remembered, since they only lead to contradictions.

The second case, however, requires to remember the changes caused by alternative 1 while executing alternative 2. If alternative 2 looks fine, too, we add a complete copy of it to the search stack and continue with a restored version of alternative 1. In a way, this is similar to the situation of using `git stash` for version control: we start from a clean, committed state (the original) and work a little (alternative 1 propagation). Suddenly, we feel the urge to return to another committed state (the original) and branch out in a different direction to test some things. We do not want to lose any work already done for the first branch, but do not have the time right now to finish it into a presentable, committable state (finish the branch). So we lay it aside into our temporary stash, return to the clean state and perform the necessary tasks there (try alternative 2). After we are done, we return to our original clean state (undo alternative 2) and re-apply the stashed changes. From there, we can continue normally with alternative 1.

Such an extra stash for temporarily undone, but relevant work is also implemented in our incremental update system. The `Stash` command saves a backup of the current stack of recent `Change`s to the stash stack. Then, the recent stack is undone. The `UndoAndApplyStash` command also undoes the recent stack to return to the original data, then re-applies the commands stored on the stash. This should return the state to exactly the version it had before the last `Stash` command.

With this incremental backtracking and stashing method, the guessing procedure for variables can be implemented like in Algorithm 10.

Algorithm 10 Guessing procedure for one variable, with incremental backtracking

Input: matrix for current state of conditions, variable to guess with two alternatives
Output: matrix for refined conditions, with variable guessed

```
create ShallowCopy of current state, without copying the matrix
guess alternative 1 and propagate
if alternative 1 caused no immediate contradictions then
  decide randomly whether to stack alternative 2 in case the branch fails later
  if yes then
    create Stash of alternative 1 and propagate alternative 2 in clean original state
    if alternative 2 caused no immediate contradictions then
      push the modified state to the search stack
    UndoAndApplyStash to continue with alternative 1
  done, continue with next variable
else
  Undo recent changes to return to original
  guess and propagate alternative 2
  if alternative 2 caused no immediate contradictions then
    done, continue with next variable
  else
    mark the current variable as problematic
    return to an earlier alternative branch from the search stack
```

Chapter 6

Evaluation

In this chapter, we compare the linear propagation method described in Chapter 4 to existing methods from Section 3.5, including perfect propagation (exhaustive search) as a reference value. We show that the propagation quality is superior compared to bitsliced propagation in the majority of cases. In particular, the new method benefits from the non-uniform distribution of inputs that occur in practical applications. The best results for larger examples are obtained with a combination of the linear and bitsliced approach.

Section 6.1 defines the figure of merit we use to evaluate the propagation quality, and explains the corresponding diagrams we use in the remaining sections. The following sections show our results for a number of different target functions with increasing size. Section 6.2 analyzes the 4-bit Σ function from various toy examples in previous chapters. Section 6.3 extends the bitsize to 32 and 64 bits and discusses the Σ_i, σ_i functions defined for SHA-2. The focus is on σ_0. Finally, Section 6.4 presents our results on the SHA-3 competition winner Keccak, where linear propagation performs particularly well.

6.1 Figures of merit

To compare and evaluate different propagation methods, we need to measure how well they propagate. Propagation corresponds to narrowing down the solution space, or gaining information about the solution. A second criterion is the runtime performance of each algorithm. We first analyze the runtime of calculating a single propagation. However, this is not sufficient to judge the overall performance. Better propagation and especially earlier detection of contradictions significantly reduce the search process and thus the number of necessary propagation calculations. This section defines the different figures of merit and graphical diagrams we used for our evaluation.

6.1.1 Propagation quality

Our most important criterion is the propagation quality achieved by each method. This quantifies the information we gain from applying the propagation method in question. A special case of this is the detection of contradictions, which has the most immediate consequences in the search algorithm. We measure the propagation quality for one particular input by comparing the number of solutions permitted by the generalized difference before and after propagation. Roughly speaking, the logarithmic scale transforms the results from number of solutions to degrees of freedom (or number of bits).

Denote by $|\Delta(z, z^*)| \in \{0, \ldots, 2^{2(m+n)}\}$ the number of solutions (message pairs with $m + n$ bits each) admitted by the generalized condition $\Delta(z, z^*)$. If $\Delta(z, z^*)$ consists of the generalized difference $\Delta(x, x^*)$ on the inputs of a function and the generalized difference $\Delta(y, y^*)$ on the outputs, then $|\Delta(z, z^*)| = |\Delta(x, x^*)| \cdot |\Delta(y, y^*)|$.

Denote by $\Delta(z, z^*)'$ the generalized condition obtained by propagating $\Delta(z, z^*)$ by means of propagation method M. Then we measure the propagation quality of M based on the figure of merit $I_M(z)$, where

$$I_M(z) = \log_2 |\Delta(z, z^*)| - \log_2 |\Delta(z, z^*)'| = \log_2 \frac{|\Delta(z, z^*)|}{|\Delta(z, z^*)'|}.$$

If $\Delta(z, z^*)$ is a contradiction, then $|\Delta(z, z^*)| = |\Delta(z, z^*)'| = 0$ and we set $I_M(z) = 0$. If $|\Delta(z, z^*)'| = 0$ but $|\Delta(z, z^*)| \neq 0$, then $I_M(z)$ is undefined, which we denote by $I_M(z) = \#$. We also refer to $I_M(z)$ as the information gain of method M for input z in bits.

To compare two propagation methods and measure the gain of method M_2 over M_1 for one specific condition $\Delta(z, z^*)$, we compute the difference of the two methods' respective figures of merit:

$$I_{\text{diff}}(z) = I_{M_1}(z) - I_{M_2}(z).$$

If $I_{M_1}(z) = \#$, then we set $I_{\text{diff}}(z) = \#$, except if also $I_{M_2}(z) = \#$; then we set $I_{\text{diff}}(z) = 0$. The case $I_{M_2}(z) = \#$ while $I_{M_1}(z) \neq \#$, meaning that method M_2 detected a contradiction but M_1 did not, is marked by $I_{\text{diff}}(z) = -\#$. Note that negative values indicate that M_2 performs better.

We refer to the unit of both I_M and I_{diff} as "bits", since they both intuitively measure the difference in number of bits necessary to enumerate a set with the size of the remaining solution space.

Example 6.1. We revisit Example 4.1 and the 4-bit Σ function. In this example, 64 pairs (x, x^*) fulfill the input difference $\Delta(x, x^*)$, and 4 pairs (y, y^*) fulfill the output difference $\Delta(y, y^*)$:

$$|\Delta(x, x^*)| = |\texttt{[1???]}| = 1 \cdot 4 \cdot 4 \cdot 4 = 64$$
$$|\Delta(y, y^*)| = |\texttt{[--00]}| = 2 \cdot 2 \cdot 1 \cdot 1 = 4.$$

If we apply the bitslice method, the generalized condition remains unchanged and

$$I_{\text{bitslice}}(z) = 0.$$

However, after applying the linear propagation method, only 8 pairs are still possible at the input and 2 pairs are possible at the output:

$$|\Delta(x, x^*)| = |\texttt{[1---]}| = 1 \cdot 2 \cdot 2 \cdot 2 = 8$$
$$|\Delta(y, y^*)| = |\texttt{[-100]}| = 2 \cdot 1 \cdot 1 \cdot 1 = 2.$$

Hence,

$$I_{\text{linear}}(x, y) = \log_2 \frac{|\Delta(x, x^*)| \cdot |\Delta(y, y^*)|}{|\Delta(x, x^*)'| \cdot |\Delta(y, y^*)'|} = \log_2 \frac{64 \cdot 4}{8 \cdot 2} = 4.$$

The gain of linear propagation over bitsliced propagation is

$$I_{\text{diff}}(x, y) = I_{\text{bitslice}}(x, y) - I_{\text{linear}}(x, y) = 0 - 4 = -4.$$

6.1.2 Diagrams for propagation quality

The previously defined figure of merit refers only to a fixed difference. We evaluate the propagation methods for different functions f by computing $I_{\text{diff}}(z)$ for a large number of randomly drawn sample differences $\Delta(z, z^*)$. Then, the random variable $I_{\text{diff}}(z)$ can be analyzed for different methods M_1 and M_2. The results depend on the distribution of the input difference samples. We chose two different distributions:

- Uniform distribution over all generalized conditions. Since explicitly contradictory initial conditions (containing `#`) certainly contribute nothing and only dilute the remaining results, we excluded them.
- Practical distribution as it occurs during a complete search. Compared to the uniform distribution, this is biased in favor of `?`, `-`, `x` and one-solution conditions `0`, `1`, `u` and `n`, and contains relatively few nonlinear conditions. In addition, there are fewer contradictions since typically only very few bits are guessed or changed between propagation steps. This means that most bits of the input were already tested for contradiction, and found consistent.

The propagation quality diagram compares the distribution of the gain $I_{\text{diff}}(z)$ of different methods to one reference method, usually bitsliced propagation. For this purpose, the figures show the empiric distribution function of $I_{\text{diff}}(z) = I_{\text{bitslice}}(z) - I_{\text{method}}(z)$. The second method can be bitsliced, linear or perfect propagation, as well as a combination of bitsliced and linear (bit+line) or bitsliced propagation with additional bitslices for the inverse function as described in Section 3.5 (bit+inv).

Higher values in this plot correspond to better propagation. For reference, the diagrams also contain the threshold-shaped distribution function for the reference function itself. The vertical line divides the plot into negative and positive values. Values above the

lower left reference line represent cases where alternative methods propagated better than the reference method, while values below the upper right reference line mean worse propagation of the method in question. The (signed) area between the reference line and any second propagation method is an indicator for the average behavior: the larger the area, the better the alternative method.

6.1.3 Runtime performance

There are two different ways to judge the runtime performance of a particular implementation of a propagation method. The first is by measuring the runtime of each single propagation process. This is achieved by either measuring the time explicitly (in the prototype), or by considering the number of iterations per second (in the search tool). The latter value varies during the search process. Both versions are basically easy to measure and relatively easy to compare. However, both are not well suited to compare completely different propagation methods. On the one hand, measuring isolated propagation processes leads to runtimes that are dominated by the unrepresentative initialization case, for example building up the matrix and adding a large number of equations for linear propagation. On the other hand, using averaged results from an actual search process is only truly meaningful if the implementations under comparison are functionally equivalent, i.e. produce the same results. Otherwise, if one implementation produces slower, but better propagation results, its overall performance may still be superior. For this reason, we mostly used these figures to evaluate implementation variations within one method. These values are the basis for performance comments in Chapter 5.

To compare, say, bitsliced and linear propagation in terms of runtime, it is necessary to consider the complete search time with one specific method employed (where possible). Unfortunately, this leads to statistically somewhat unsatisfactory results, especially for larger examples. This random variable displays a very high variance, and the behavior changes wildly for different starting points and, in particular, different hash functions. In addition, measurements take relatively long and it is harder to collect a statistically suitable number of samples. For these reasons, we sorted the analysis results by hash function and focused more on the propagation quality. Section 6.4 contains some runtime results for SHA-3.

6.2 Experiments for 4-bit Σ

Our first target function is the 4-bit Σ function introduced in Example 3.4. This function is a variant of a SHA-2 building block with reduced bit size, and thus structurally representative for practical applications. At the same time, it is small enough to compare the performance of bitsliced and linear propagation with perfect propagation. We also included bitsliced propagation with additional bitslices for the inverse function. As a

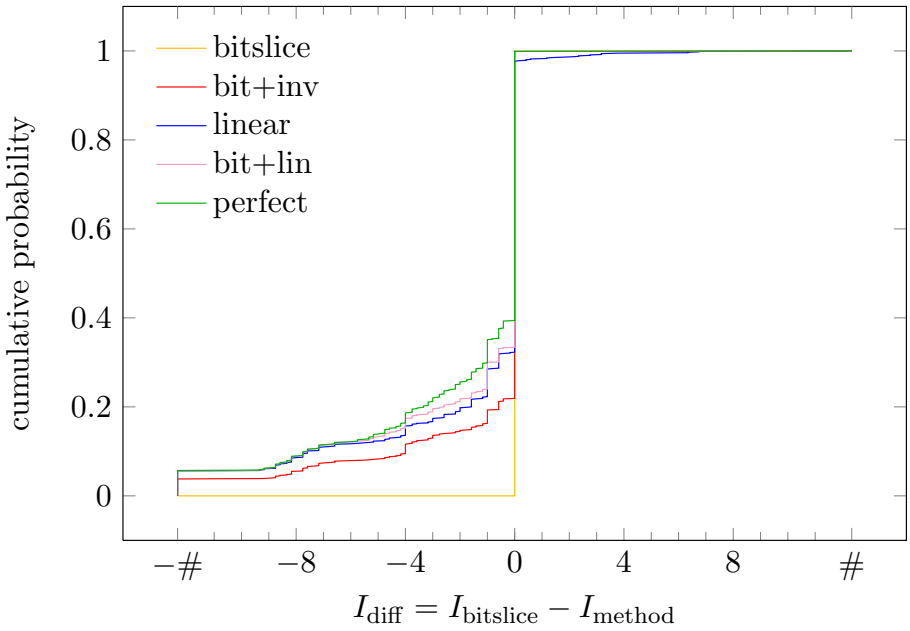


Figure 6.1: Comparison of propagation methods for 4-bit Σ, with bitsliced propagation as reference method and uniformly distributed inputs.

fifth propagation method, we apply a combination of bitsliced and linear propagation where both methods are iterated until neither can propagate any further.

As an additional advantage, the number of 4-to-4-bit generalized differences is sufficiently small to allow the complete analysis of all possible values instead of selecting a set of sample values. We only leave out the useless contradictory inputs containing # and test all remaining $15^8 < 2^{32}$ 8-bit generalized differences. The resulting values of I_{diff} can be found in Figure 6.1, with bitsliced propagation as reference method. The experiments were conducted with the prototype implementation. Higher values correspond to better propagation.

The main information from these experiments is that linear propagation performs at least as good as bitsliced propagation in almost all cases, corresponding to $I_{\text{diff}} < 0$ in the left half. Only for very few samples, bitslicing works better, represented by the short blue line in the upper corner to the right of the threshold. In virtually no cases, bitsliced propagation detected a contradiction where linear propagation did not. Conversely, roughly 5 % of cases were contradictory and only bitslicing failed to detect it. In around 30 % of all cases, linear propagation performed strictly better than bitslicing.

It is interesting to note that the difference between perfect and linear propagation is quite small and the combination bitslice + linear does not improve much on the linear propagation. This is especially noteworthy since more than 90 % of all samples contained nonlinear conditions, significantly more than typically occur during a practical search.

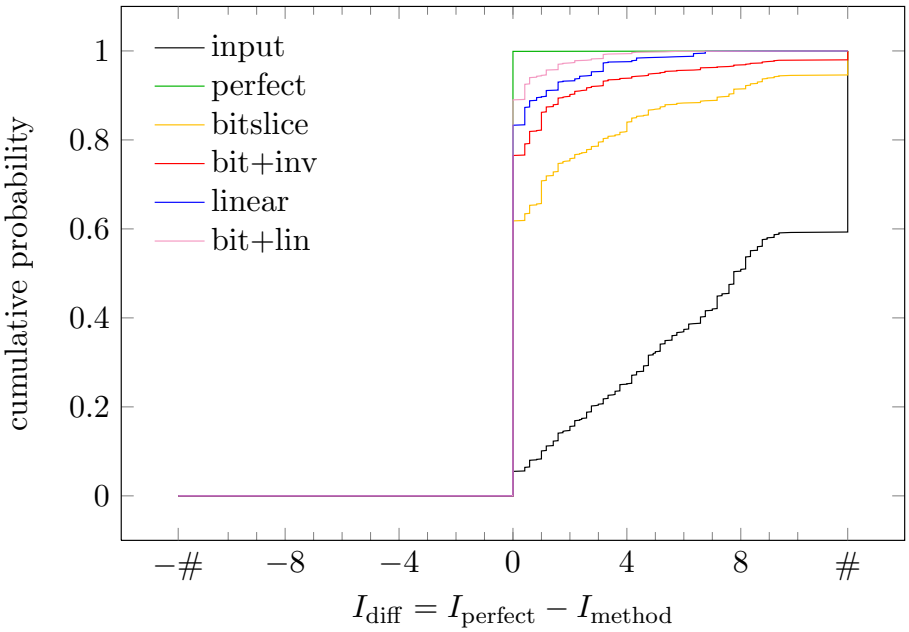


Figure 6.2: Comparison of propagation methods for 4-bit Σ, with perfect propagation as reference method and uniformly distributed inputs.

Figure 6.2 is based on the same experiments, but with perfect propagation as the reference method for the diagram. Additionally, an additional graph is added for the original input difference, i.e. no propagation. In contrast to the previous plot, this figure explicitly quantifies the missed contradictions for each propagation method. Linear propagation missed virtually no contradictions. Around 2 % of all samples were contradictory without inverse-bitsliced propagation detecting it, and for 5.5 %, the simple bitsliced method failed to find the contradiction. The input graph shows that 40 % of all samples were implicitly contradictory.

Surprisingly, only 5.5 % of all input differences were already stable and did not propagate. For the remaining 55 %, the degree of propagation is relatively uniformly distributed between 0 and 9.5 bits (out of up to 32 possible bits, or 24 without counting contradictions). In contrast, other propagation methods typically perform only a few bits worse than perfect propagation, and cases with high deviation are relatively rare.

The runtime results for these experiments are not representative for linear propagation. While the precomputation for the bitsliced method is performed only once initially, precomputations for linear propagation such as setting up the matrix are repeated for each propagation request. Because of this, the measured runtime for linear propagation is far too high an estimation for actual propagation steps during a search.

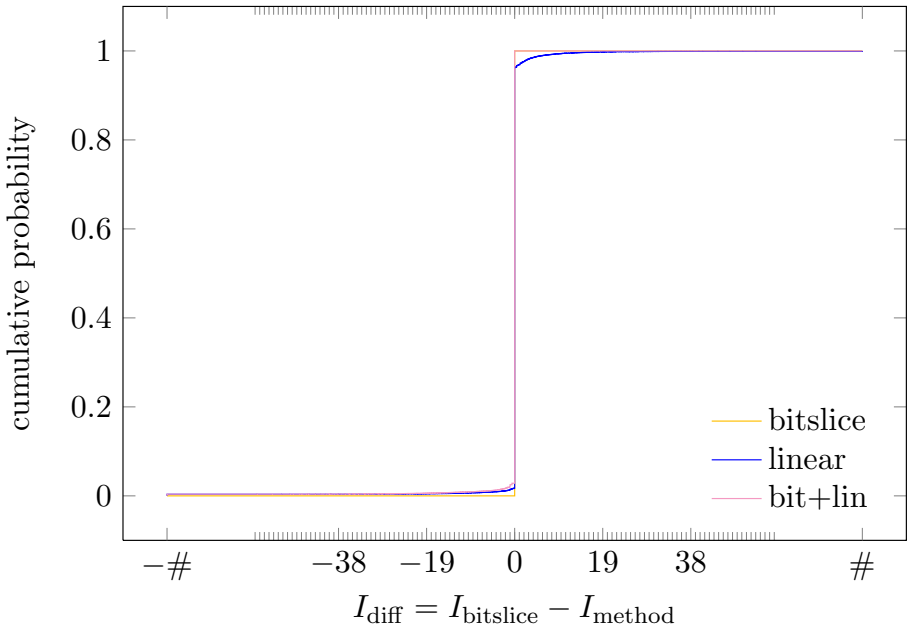


Figure 6.3: Comparison of propagation methods for 32-bit σ_0, with bitsliced propagation as reference method and uniformly distributed inputs.

6.3 Experiments for 32-bit Σ_0 and σ_0

Similar experiments, but without perfect propagation and only for a uniformly distributed selection of random samples, were also conducted for the 32-bit variant Σ_0 of Σ as it occurs in the SHA-2 standard, see Section 2.4. There is hardly any difference between bitsliced and linear propagation.

Results are slightly more interesting for the slightly different σ_0 function, also from the SHA-2 definition. Unlike Σ_0, where each output bit is the xor of three input bits, some output bits of σ_0 depend only on two input bits. This simplifies and improves propagation for both the linear and the bitsliced approach. The corresponding diagram can be found in Figure 6.3. Surprisingly, linear propagation performs relatively poor for this function. The results are not significantly worse or better than those from bitsliced propagation overall, but the calculation is much slower.

In our samples, linear was better than bitsliced propagation for about 2.3 % of all cases. In the diagram, this corresponds to the left half of the blue graph. Conversely, bitslicing was better than linear propagation in roughly 4.8 % of our samples, as can be observed in the right half. The hybrid method is naturally always at least as good as bitsliced propagation, but only rarely better.

This result seems a bit counterintuitive since linear propagation performed well for the 4-bit example, and its advantage over bitsliced propagation is its global behavior especially

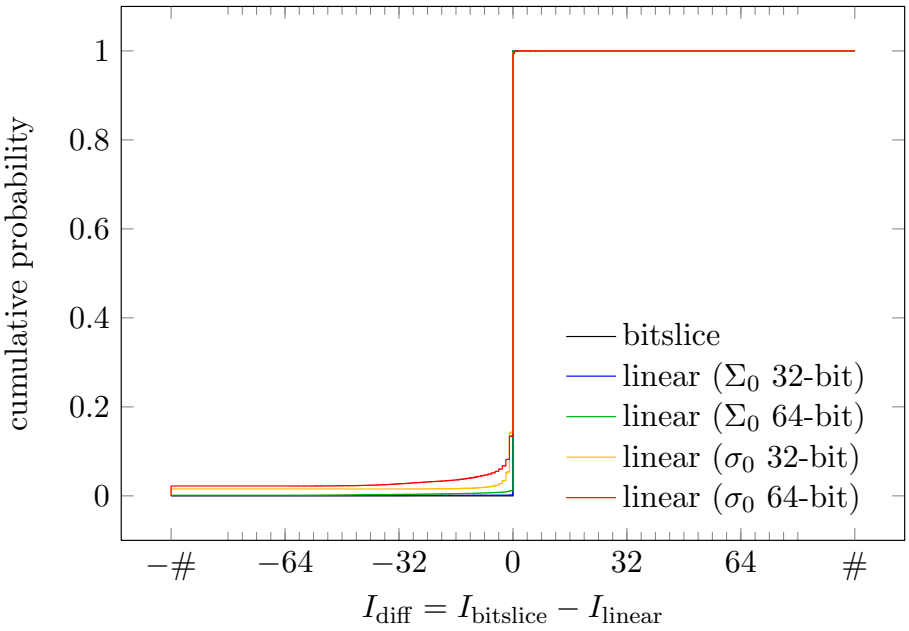


Figure 6.4: Comparison of linear and bitsliced propagation for 32-bit and 64-bit Σ_0 and σ_0, with bitsliced propagation as reference method and samples drawn from a search process.

for larger systems. One reason may be the dominating role of nonlinear conditions for propagation: the probability for an initial characteristic to contain only linear conditions is just $2 \cdot 10^{-7}$ %.

The situation looks better if the samples are not uniformly distributed, but drawn from a search process in the search tool. The reason for this is that some generalized conditions are much more likely to occur during the search process than others. In particular, the simple linear conditions `?`, `x`, `-`, `n`, `u`, `0` and `1` occur very frequently since they are explicitly guessed, while the four nonlinear conditions `E`, `B`, `D` and `7` are relatively rare. During large parts of the search, `?` is the dominating condition. As expected, this distribution of conditions favors linear propagation, which performs clearly better.

Figure 6.4 demonstrates the differences between Σ_0 and σ_0, as well as between the 32-bit (SHA-256) and 64-bit (SHA-512) version. Each linear graph is plotted in reference to its own bitsliced graph, for different random input samples from a search process. The plots clearly show that linear propagation offers more improvement over bitslicing for σ_0 than it does for Σ_0. Part of the reason for this is that σ_0 is used in the message expansion, while Σ_0 is part of the regular round function. This causes slightly different distributions of the inputs to the two functions. In addition, the σ_0 definition is slightly sparser and contains some bitslices with two instead of three inputs. While this improves the propagation of single bitslices, the linking between those bitslices also becomes slightly looser, which may be a disadvantage for bisliced propagation. It can also be observed that

larger functions favor linear propagation. This is in accordance with our assumption that bitsliced propagation is too local, and that the linear approach improves the propagation quality on a global level. However, compared to Keccak, the overall advantage of linear propagation is limited.

6.4 Experiments for SHA-3

The final target for our evaluation is Keccak, the new SHA-3 standard, already introduced in Section 2.4. Its core transformation is particularly interesting for linear propagation. The large state size of 1600 bits for the default lane size makes it very hard to tackle with bitsliced propagation. On the other hand, 4 of its 5 steps θ, ρ, π, χ and ι are linear functions, which makes it attractive to model all linear functions of each round as one large linear layer for linear propagation. The resulting matrix has 50 times as many rows and columns as for the Σ_i, σ_i functions of SHA-256 and is thus 2500 times as large.

We define the linear layer as $\pi \circ \rho \circ \theta$. The definition of χ is nonlinear, so this part of each round is propagated using bitslicing only. The final operation, ι, is only the addition of a round constant, which can be trivially propagated without any specific method.

For random inputs, almost all samples describe impossible characteristics. For this reason, we only used practical samples drawn from actual search processes. Compared to uniformly distributed inputs, they are much more likely to be consistent.

We also analyze the non-standard lane sizes of 32, 16 and 8 bits to demonstrate that the larger the linear function, the better linear propagation performs in comparison with bitslicing. The resulting graphs for linear propagation, with bitsliced propagation as reference method, are collected in Figure 6.5. Note that of course, the values of I_{linear} for each lane size were plotted against I_{bitslice} for the same lane size, i.e. each graph is relative to its own reference method. This explains that the values of I_{diff} are generally smaller for smaller lane sizes.

The figure shows that linear propagation is generally much better than bitsliced propagation at detecting contradictions. Larger lane sizes increase this advantage. Most notably, almost half of all samples for 64-bit lanes were contradictions that linear propagation detected, but bitsliced propagation did not. For 97 % of all 64-bit samples, linear propagation performed better than bitslicing. This also implies that almost all guesses and changes directly cause significant propagation.

Again, linear propagation is slower than bitslicing for single propagation steps. However, the far superior propagation quality of the linear approach clearly makes it the better choice here. To illustrate this, we attack two rounds of Keccak. Table 6.1 lists the number of iterations per second and the number of found solutions in a fixed amount of time for different lane sizes. The results for 8 and 16 bits were measured after a short

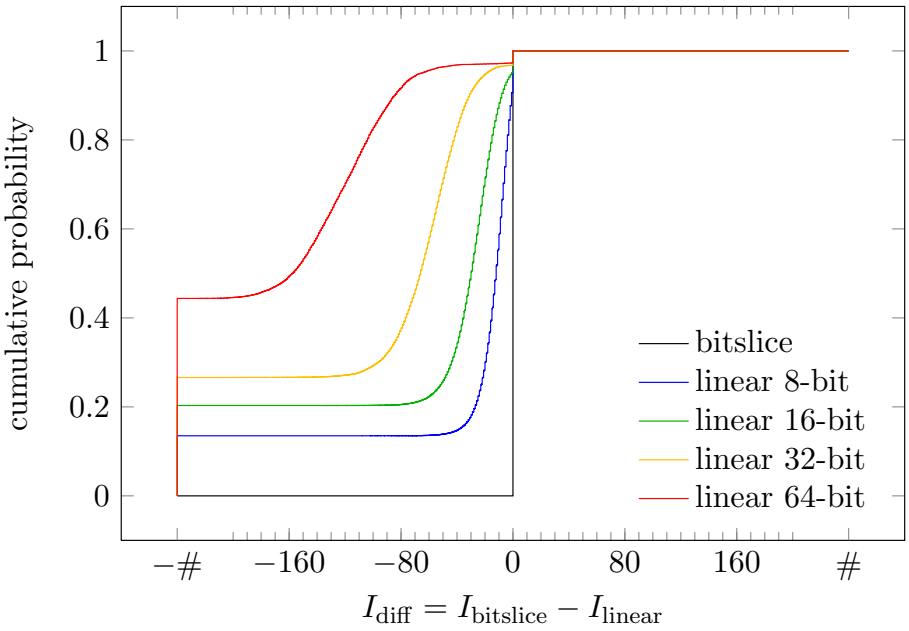


Figure 6.5: Comparison of linear and bitsliced propagation for Keccak's linear layer with different lane sizes, with bitsliced propagation as reference method and samples drawn from a search process.

		2h		16h	
lane size		8-bit	16-bit	32-bit	64-bit
iterations/sec	bitslice	21158	21542	14612	9566
	linear	4272	2102	769	222
solutions found	bitslice	333	4	0	0
	linear	1069	41	4	0

Table 6.1: Runtime results for Keccak with lane sizes of 8, 16, 32 and 64 bits

search of 2 hours, while those for 32 and 64 bits correspond to 16 hours of searching. The table clearly shows that with increasing step size, the advantage of linear over bitsliced propagation grows. While for 8 bit lanes, linear propagation finds 3 times as many results as bitsliced propagation, it finds already 10 times as many for 16 bit lanes. For 64-bit lanes, the time limit was too short for both propagation methods to find any solutions.

Summarizing, linear propagation is better than bitslicing for large linear functions like the Keccak linear layer, and in particular very good at detecting contradictions.

Chapter 7

Conclusion

In this thesis, we motivated the importance of collision-resistant hash functions in Chapter 2, and presented several construction schemes for such functions, as well as the standards SHA-2 and SHA-3. In Chapter 3, we introduced the concepts of differential collision attacks. In particular, we are focusing on guess-and-determine attacks for the difference model of generalized conditions. Such attacks require a propagation method to determine consequences of a guess and to detect contradictions. We demonstrated that the main propagation method in previous publications, bitsliced propagation, yields only locally perfect solutions, and the global results are often not satisfactory. We presented an alternative algorithm, based on linear algebra, with more globally oriented behaviour in Chapter 4.

If only linear conditions and linear functions are involved, this method achieves perfect propagation. While the approach can theoretically be generalized for nonlinear functions, we only recommend it for linear parts of the hash function. We suggested an extension for nonlinear conditions, but since nonlinear conditions play only a minor role in practical attacks, we recommend to just ignore nonlinear information instead.

We implemented this linear method and combined it with bitsliced propagation for practical attacks on reduced versions and building blocks of SHA-2 and SHA-3. It was integrated into an existing search tool for guess-and-determine attacks. While our method is slower than bitslicing, it usually provides better propagation quality. Especially for very large, linear functions such as the linear layer of SHA-3, the superior propagation quality outweighs the slower propagation and speeds up the search process. The most important effect is the better detection of contradictions, which curtails the search tree. The exact effect on the total runtime of the search is hard to measure reliably due to the large variance of individual results. However, our experiments with different reduced lane size versions of Keccak clearly show the overall runtime advantage of linear propagation. To quantify the propagation quality, we developed suitable characteristic variables and diagrams in Chapter 6.

To speed up the propagation process, we investigated useful data structures and algorithms for storing and updating the matrix in Chapter 5. Often, the simpler solutions practically performed better than theoretically superior approaches due to the overhead of more complex data structures.

Since the method is very generic and easy to extend to other hash functions, it would be interesting to compare its performance for linear functions of different size and structure. In particular, for SHA-2 and SHA-3, combining multiple rounds into one linear function might further improve on our results from Chapter 6. It is also possible to extend our approach by translating not only bitwise generalized conditions to linear equations, but more general constraints such as two-bit conditions.

Summarizing, our new linear method significantly improves the performance of guess-and-determine attacks for large, linear functions.

Bibliography

[1] M. Albrecht. *Algorithmic Algebraic Techniques and their Application to Block Cipher Cryptanalysis*. PhD thesis, Royal Holloway, University of London, 2010.

[2] M. R. Albrecht and C. Cid. Algebraic Techniques in Differential Cryptanalysis. In O. Dunkelman, editor, *FSE*, volume 5665 of *LNCS*, pages 193–208. Springer, 2009.

[3] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Sponge functions. Ecrypt Hash Workshop 2007, May 2007.

[4] E. Biham. How to make a difference: Early history of differential cryptanalysis. FSE 2006 in Graz, Austria, 2006.

[5] E. Biham and R. Chen. Near-collisions of SHA-0. In M. K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004, Proceedings*, volume 3152 of *LNCS*, pages 290–305. Springer, 2004.

[6] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. pages 3–72, 1991.

[7] N. Borisov, M. Chew, R. Johnson, and D. Wagner. Multiplicative differentials. In J. Daemen and V. Rijmen, editors, *FSE*, volume 2365 of *LNCS*, pages 17–33. Springer, 2002.

[8] R. P. Brent. An improved Monte Carlo factorization algorithm. *BIT, Nord. Tidskr. Inf.-behandl. 20*, pages 176–184, 1980.

[9] F. Chabaud and A. Joux. Differential collisions in SHA-0. In H. Krawczyk, editor, *CRYPTO*, volume 1462 of *LNCS*, pages 56–71. Springer, 1998.

[10] S.-J. Chang, R. Perlner, W. E. Burr, M. S. Turan, J. M. Kelsey, S. Paul, and L. E. Bassham. Third-round report of the SHA-3 cryptographic hash algorithm competition, 2012.

[11] N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In B. Preneel, editor, *EUROCRYPT*, volume 1807 of *LNCS*, pages 392–407. Springer, 2000.

[12] R. Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *LNCS*. Springer, 2005.

[13] I. B. Damgård. Collision free hash functions and public key signature schemes. In D. Chaum and W. L. Price, editors, *EUROCRYPT*, volume 304 of *LNCS*, pages 203–216. Springer, 1987.

[14] M. Daum. *Cryptanalysis of Hash functions of the MD4-family*. PhD thesis, 2005.

[15] C. De Cannière and C. Rechberger. Finding SHA-1 characteristics: General results and applications. In X. Lai and K. Chen, editors, *ASIACRYPT*, volume 4284 of *LNCS*, pages 1–20. Springer, 2006.

[16] H. Dobbertin. The status of MD5 after a recent attack. *CryptoBytes*, 2(2):1–6, 1996.

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

[18] H. Dobbertin. Cryptanalysis of MD4. *Journal of Cryptology*, 11(4):253–271, 1998.

[19] M. Eichlseder, F. Mendel, T. Nad, V. Rijmen, and M. Schläffer. Linear propagation in efficient guess-and-determine attacks. In *Proceedings of the International Workshop on Coding and Cryptography (WCC 2013), Electronic Notes in Discrete Mathematics*. Elsevier, 2013.

[20] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases Without Reduction to Zero (F_5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, pages 75–83 (electronic), New York, 2002. ACM.

[21] Information Technology Laboratory, National Institute of Standards and Technology. FIPS PUB 180-4, Secure Hash Standard (SHS), 2012.

[22] L. R. Knudsen. Truncated and higher order differentials. In B. Preneel, editor, *FSE*, volume 1008 of *LNCS*, pages 196–211. Springer, 1994.

[23] L. R. Knudsen and Universitetet i Bergen. Dept. of Informatics. *Deal: A 128-bit Block Cipher*. Reports in informatics. Department of Informatics, University of Bergen, 1998.

[24] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[25] X. Lai. Higher order derivatives and differential cryptanalysis. In R. E. Blahut, J. Costello, Daniel J., U. Maurer, and T. Mittelholzer, editors, *Communications and Cryptography*, volume 276 of *The Springer International Series in Engineering and Computer Science*, pages 227–233. Springer US, 1994.

[26] X. Lai and J. L. Massey. Hash Functions Based on Block Ciphers. In R. A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT '92, Proceedings*, volume 658 of *LNCS*, pages 55–70. Springer, 1993.

[27] G. Leurent. Analysis of differential attacks in ARX constructions. In *ASIACRYPT*, pages 226–243, 2012.

[28] F. Massacci and L. Marraro. Logical Cryptanalysis as a SAT Problem. *J. Autom. Reasoning*, 24(1/2):165–203, 2000.

[29] S. M. Matyas, C. H. Meyer, and J. Oseas. Generating strong one-way functions with crypographic algorithm. *IBM Technical Disclosure Bulletin*, 27(10A):5658–5659, 1985.

[30] F. Mendel, T. Nad, and M. Schläffer. Finding SHA-2 characteristics: Searching through a minefield of contradictions. In D. H. Lee and X. Wang, editors, *ASIACRYPT*, volume 7073 of *LNCS*, pages 288–307. Springer, 2011.

[31] F. Mendel, T. Nad, and M. Schläffer. Finding collisions for round-reduced SM3. In E. Dawson, editor, *Topics in Cryptology - CT-RSA 2013*, volume 7779 of *LNCS*, pages 174 – 188. Springer, 2013.

[32] F. Mendel, T. Nad, and M. Schläffer. Improving local collisions: New attacks on reduced sha-256. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013*. Springer, 2013. in press.

[33] A. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[34] R. C. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford, CA, USA, 1979. AAI8001972.

[35] G. Nivasch. Cycle detection using a stack. *Inf. Process. Lett.*, 90(3):135–140, 2004.

[36] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, 1993. René Govaerts and Joos Vandewalle (promotors).

[37] M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical report, Cambridge, MA, USA, 1979.

[38] V. Rijmen and E. Oswald. Update on SHA-1. In A. Menezes, editor, *Topics in Cryptology – CT-RSA 2005, Proceedings*, volume 3376 of *LNCS*, pages 58–71. Springer, 2005.

[39] M. Schläffer. *Cryptanalysis of AES-Based Hash Functions*. PhD thesis, 2011.

[40] SciPy.org Project. Scipy package documentation on sparse matrix formats.

[41] M. Stevens, A. K. Lenstra, and B. de Weger. Chosen-prefix collisions for MD5 and applications. *IJACT*, 2(4):322–359, 2012.

[42] D. Wagner. The boomerang attack. In L. R. Knudsen, editor, *FSE*, volume 1636 of *LNCS*, pages 156–170. Springer, 1999.

[43] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Cramer [12], pages 1–18.

[44] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In V. Shoup, editor, *CRYPTO*, volume 3621 of *LNCS*, pages 17–36. Springer, 2005.

[45] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In Cramer [12], pages 19–35.

[46] M. N. Wegman and L. Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.

[47] T. Xie and D. Feng. How to find weak input differences for MD5 collision attacks. *IACR Cryptology ePrint Archive*, 2009:223, 2009.