Formal Algebraic Reasoning About Compression Function Security

by

Zahra Javar
B.Sc., Shahid Beheshti University, 2014
M.Sc., Sharif University of Technology, 2017

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Engineering and Computer Science

Formal Algebraic Reasoning About Compression Function Security

by

Zahra Javar
B.Sc., Shahid Beheshti University, 2014
M.Sc., Sharif University of Technology, 2017

Supervisory Committee

Dr. Bruce M. Kapron, Supervisor
(Department of Computer Science)

Dr. Yun Lu, Departmental Member
(Department of Computer Science)

Dr. Riham AlTawy, Outside Member
(Department of Electrical and Computer Engineering)

Dr.Rei Safavi-Naini, External Examiner
(Department of Computer Science, University of Calgary)

ABSTRACT

Cryptographic hash functions are fundamental in cryptographic constructions, as they transform variable-length input into fixed-length output while maintaining essential security properties like collision resistance, preimage resistance, and indifferentiability from a random oracle (RO). Creating efficient hash functions with provable security has long posed a challenge. Security proofs for hash functions usually fall under the random oracle model or the ideal cipher model, which assumes access to an ideal primitive like a truly random function or permutations.

This research endeavors to establish a systematic approach for analyzing the security of hash functions suitable for automated verification and function generation within both ideal models. Building upon prior work [25], which employed an algebraic framework known as Linicrypt[8], primarily for analyzing collision-resistant hash functions in the random oracle model, we extend our efforts in two key directions.

We first introduce a simple and easily verifiable property of Linicrypt programs that characterizes preimage awareness, a security property introduced by Dodis, Ristenpart, and Shrimpton [13] who also demonstrate its utility in the construction of indifferentiable hash functions. We also illustrate how this characterization can be efficiently automated and provide an example by enumerating preimage-aware compression functions that employ two random oracle calls. This includes several functions that Dodis et al. previously proved to be preimage aware through manual methods.

Next, we broaden the Linicrypt framework, originally proposed in the random oracle setting, to encompass hash function security in the ideal cipher model. Within this context, we delineate collision- and second-preimage-resistance properties using linear-algebraic conditions on Linicrypt programs. We also introduce an efficient algorithm for determining program compliance with these conditions. As an application, we delve into the case of block cipher-based hash functions as proposed by Preneel, Govaerts, and Vandewall [32] and establish that our characterization encapsulates the semantic analysis of PGV presented by Black et al.[5].

Additionally, our research further extends into the ideal cipher model to analyze group-2 compression functions, a category introduced in the well-known work[4]. These are compression functions which are not collision-resistant themselves, but produce collision-resistant hash functions when iterated by the Merkle-Damgard transformation. We also provide a comprehensive characterization of collision-resistant

double block length compression functions within the ideal cipher model.

Contents

List of Figures

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Professor Bruce Kapron, whose unwavering support, vast knowledge, and understanding have been instrumental throughout my doctoral journey. Professor Kapron's exceptional guidance, insightful comments, and exemplary leadership have played a pivotal role in the successful completion of this thesis.

I would also like to extend my sincere thanks to Dr. Yun Lu, Dr. Riham Altway, and Dr. Rei Safavi-Naini, who graciously accepted the role of examiners for my oral examination. Their expertise and contributions enriched the examination process, and I am truly grateful for their time and insights.

My family and close friends have been a constant source of strength and encouragement during the challenging times of this academic pursuit. I owe a special debt of gratitude to them for their unwavering belief in my abilities and their unwavering support.

I would like to acknowledge the invaluable assistance provided by the staff in the Computer Science department. Wendy Beggs, Kath Milinazzo, Nancy Chan, and Erin Robinson have been instrumental in navigating the administrative aspects of my academic journey. A special thanks goes to Glen McCloskey for his technical assistance in fixing my laptop, which proved to be a great relief during critical moments of my research.

I would also like to express my profound gratitude to the U Vic Fellowship Program and the NSERC Discovery Grant Program for providing funding, which has been a significant financial support throughout my academic journey.

Lastly, I express my appreciation to the CARSA squash court, which provided a welcome respite and a sense of well-being during times of overwhelming stress.

DEDICATION

This thesis is dedicated to the memory of my father, a guiding light in my journey, whose influence continues to shape my aspirations and belief in myself.

Chapter 1

Introduction

Hash functions are a fundamental cryptographic primitive used as a building block in a large number of cryptographic constructions [31]. Applications of cryptographic hash functions include domain extension for message-authentication codes [2] and digital signatures [10], as well as hash-based signatures [6, 15, 23, 27]. Hash functions are also an essential component in many public-key primitives built using the random-oracle methodology [3], in which an ideal construction using a random oracle is instantiated with a concrete hash function in place of the oracle. A hash function is usually built by iterating a fixed-input-length compression function over the blocks of a variable-length message. Hash functions are designed to be computationally efficient and to satisfy some security properties like collision resistance, preimage resistance, and second preimage resistance depending on the particular application. While collision- and preimage-resistance are sufficient for many basic applications, more advanced applications require stronger notions of security, the strongest being that a hash function "behaves like" a random oracle. Building efficient hash functions with proven security can be challenging, and most approaches tend to be *ad hoc*, and not amenable to automated techniques for analysis and generation.

In this research, we use an algebraic formal system called *Linicrypt* [8] in order to characterize security properties of compression functions in both the random oracle model and the ideal cipher model. This algebraic characterization supports automated verification of security as well as the automated generation of compression functions. This is in contrast to existing approaches to proving security properties [13, 4], which are *ad hoc* and not obviously amenable to automation.

This research aims to broaden existing work [25], which uses Linicrypt, a uniform and automatable framework for security proofs in the random oracle model, to

characterize collision-resistance properties for compression functions. We extend this work, both to capture stronger security properties and to work with the ideal cipher model. The stronger notion of security we consider is called preimage awareness (PrA) [13] which in terms of security, sits between preimage resistance and indifferentiability. Unlike preimage resistance, in the PrA security game, the adversary picks the hash function image. PrA is the strongest existence security notion that is preserved by *Merkle-Damgård (MD) transform*. We also offer a systematic and potentially automatable approach for constructing indifferentiable hash functions, aligning with the methodology outlined in [13]. We provide a simple characterization of PrA for compression functions in the Random Oracle model, which can be checked in polynomial time and we use it to generate all the compression functions with two inputs and two calls to the random oracle.

Many compression functions such as Davies–Meyer, Matyas–Meyer–Oseas, Miyaguchi–Preneel [4], and MDC-2, MDC-4, Hirose [30][19], are constructed using block ciphers [4, 19]. It is therefore a natural and recommended extension, as suggested by [25], to enhance the Linicrypt framework by incorporating the ideal cipher model. This extension is aimed at enabling the modeling of compression functions that rely on ideal ciphers.

Within the extended Linicrypt framework in the ideal cipher model, we focus on characterizing the collision resistance of such compression functions. Moreover, the 64 PGV compression functions are block-cipher-based and seamlessly integrated into the Linicrypt in the ideal cipher model. In Section 3.3, we employ Linicrypt to model the PGV compression functions, demonstrating that the characteristics proposed in [5] can be specifically derived as a particular instance within our broader characterization outlined in 3.2.5.

Furthermore, this research extends its scope to encompass the characterization of compression functions that may not initially possess collision resistance but still result in collision-resistant hash functions after the application of Merkle-Damgård transform. As a notable outcome of these characterizations, we are able to verify and generate the Group 1 and Group 2 [4] PGV compression functions [32]. This component of our research thus provides valuable insights into the automated verification and generation of collision and 2nd-preimage resistant compression functions in the ideal cipher model.

1.1 Background

This section aims to introduce the basic concepts and definitions needed to model compression functions utilizing ideal primitives in Linicrypt and review their security properties, as well as a systematic approach — the *Merkle-Damgård (MD) transform* — for constructing variable-length input hash functions by iterating compression functions, and finally consider security properties that are preserved by this transformation.

In general, hash functions are designed to process variable-length input strings and transform them into fixed-length output strings. A hash function that only accepts fixed-length inputs is referred to as a *compression* function. In order to formally model hash function security, we must consider families of functions indexed by a key s; in particular, H^s is a hash function from this family of functions. [1]

Definition 1.1.1. *[21] A hash function is a pair of probabilistic polynomial-time algorithms* (Gen, H) *fulfilling the following:*

1. *Gen is a probabilistic algorithm that takes as input a security parameter* 1^λ *and outputs a key* s *we assume that* 1^λ *is included in* s*.*

2. *There exists a polynomial* l *such that* H *is a deterministic polynomial time algorithm that takes as input a key* s *and any string* $x \in \{0,1\}^*$ *and outputs a string* $H^s(x) \in \{0,1\}^{l(\lambda)}$*.*

If for every λ *and* s*,* H^s *is defined only over inputs of length* $l'(\lambda)$ *where* $l'(\lambda) > l(\lambda)$*, then we say that* (Gen, H) *is a* fixed-length hash function *or* compression function *with length parameter* l'*.*

Some fundamental security properties of cryptographic hash functions which are necessary to ensure security for cryptographic applications are *collision resistance*, *preimage resistance*, and *2nd-preimage resistance*. These properties can be defined through an experiment involving a hash function $\Pi(Gen, H)$ and an adversary $\mathcal{A}$. In these definitions, λ denotes a *security parameter*. Note that the adversary is given the key s as input (so s is *not* secret.)

Definition 1.1.2. *A hash function* $\Pi = (Gen, H)$ *is* collision resistant *if for every probabilistic polynomial-time adversary* $\mathcal{A}$ *there exists a negligible function negl such*

[1] Unlike the setting of encryption, the key here is not secret — its purpose is to prevent an adversary from simply hard-coding a collision into its algorithm.

that $\mathcal{A}$'s probability of winning the following game is at most $negl(\lambda)$.

$$s \leftarrow Gen(1^\lambda);\ (x, x') \leftarrow \mathcal{A}^H(s);\ return (x \neq x') \wedge (H^s(x) = H^s(x'))$$

Definition 1.1.3. *A hash function $\Pi = (Gen, H)$ called* 2nd-preimage resistant *if for any probabilistic polynomial-time adversary $\mathcal{A}$ there exists a negligible function $negl$ such that $\mathcal{A}$'s probability of winning the following game is at most $negl(\lambda)$.*

$$s \leftarrow Gen(1^\lambda);\ x \leftarrow \{0,1\}^*;\ x' \leftarrow \mathcal{A}^H(s, x);\ return\ (x \neq x')\ \wedge\ H^s(x) = H^s(x')$$

Definition 1.1.4. *A hash function $\Pi = (Gen, H)$ is* preimage resistant *if for any probabilistic polynomial-time adversary $\mathcal{A}$ there exists a negligible function $negl$ such that $\mathcal{A}$'s probability of winning the following game is at most $negl(\lambda)$.*

$$s \leftarrow Gen(1^\lambda);\ y \leftarrow \{0,1\}^{l'(\lambda)};\ x \leftarrow \mathcal{A}^H(s, y);\ return\ y = H^s(x)$$

The above definitions are in decreasing order of strength — collision resistance implies 2nd-preimage resistance, while for "reasonable" hash functions 2nd-preimage resistance implies preimage resistance.

A collision-resistant hash function H is 2nd-preimage resistant because, if when given any preimage x the adversary can find a 2nd preimage $x' \neq x$ where $H(x) = H(x')$ then the adversary also can find a collision by choosing some x and finding a 2nd preimage x'. Similarly, a 2nd-preimage-resistant hash function H is preimage resistant because if, when given an image y the adversary can find a preimage x where $H(x) = y$ then when given a preimage x can compute $H(x) = y$ and having y finds the preimage x'. Assuming that the domain of H is much larger than the range and that it is not "pathological" then with good probability $x \neq x'$. As we do not deal with preimage resistance in this thesis we won't consider this last point in more detail, but refer to [33] for a full discussion.

A generic attack for finding a collision in a hash function is known as the birthday attack; this says that if the output length of a hash function is l then a collision can be found in $\mathcal{O}(2^{l/2})$ queries to the hash function with probability more than half. This attack provides an absolute lower bound on security based only on the output length. More details are given in Section A.1.1.

Ideal Primitives It is often challenging to design cryptographic primitives which provably provide a needed security property, even in the presence of computational

assumptions. One approach to deal with this problem is to assume the existence of an ideal primitive, such as an ideal block cipher or a random oracle which may be used to prove the security of new primitives. The new primitive is then implemented using a real-world instantiation of the ideal primitive. While this approach is not sound in general — there are primitives that can be proven secure when using a truly random oracle but not when using a non-ideal hash function [7] — it is widely used in practice and is considered to provide some formal assurance of security.

The idea of modeling a block cipher as a random permutation appears as early as the work of Shannon [34]. In the ideal cipher model, all the parties have access to a family of permutations E_k. For a given key k (depending on the security parameter λ), E_k is chosen uniformly from the set of all possible permutations on $\{0,1\}^{l(\lambda)}$. All parties have access to E_k and E_k^{-1} and the adversary is restricted based on the number of queries it makes.

A random oracle can be thought of as a black box that implements a random function f, which is uniformly selected from the set of all the functions $\{f \mid f : \{0,1\}^* \to \{0,1\}^{l(\lambda)}\}$. All parties have access to the random oracle and the adversary is restricted based on the number of queries it makes. This model was first adopted in [16] and was formalized in [3].

In studying the construction of compression functions in the random oracle model, the goal is to find constructions based on the existence of a simpler ideal primitive, so a random oracle with domain all of $\{0,1\}^*$ and a fixed image size is already too strong. In this case, it makes sense to consider a very simple form of random oracle which is just a length-preserving random function.[2]

Compression Functions Using an Ideal Primitive. As we mentioned, this research focuses mainly on the construction of compression functions utilizing ideal primitives, such as ideal ciphers or (length-preserving) random oracles. Here we review some of the existing compression functions that use an ideal primitive with varying levels of security. A well-studied group of ideal-cipher-based compression functions was proposed by Preneel, Govaerts, and Vandewalle (PGV) [32]. They proposed a systematic way to construct ideal-cipher-based compression functions with only a single call to the ideal cipher (rate-1) using only a simple algebraic operation (XOR). They introduced 64 rate-1 compression functions $h : \{0,1\}^l \times \{0,1\}^l \to \{0,1\}^l$

[2]Note that once we define compression and hash functions using an ideal primitive, we no longer need to also explicitly use a key s as part of the definition, as the ideal primitive itself parameterizes the construction and definition of security.

utilizing a single call to an block-cipher $E : \{0,1\}^l \times \{0,1\}^l \to \{0,1\}^l$ of the form $h^E(y,m) = E_a(b) \oplus c$ where y and m are the compression function inputs and $a, b, c \in \{y, m, y \oplus m, v\}$, and v is a fixed constant. In [4], Black, Rogaway, and Shrimpton prove that of the 64 PGV compression functions, 12 of them referred to as *group-1* are collision-resistant and preimage resistant up to the birthday bound, and 8 of them, which are called *group-2* are only collision resistant after some iteration. But 8 of the group-1 compression functions are vulnerable to a fixed point attack, which means it is easy for an adversary to find m where $h^E(y,m) = m$. One of the PGV compression functions, the *Davies-Meyer* function, is defined as $h^E(y,m) = E_y(m) \oplus m$. Real-word hash functions based on the Davies-Mayer compression function include the MD5 and SHA family. In the random oracle setting, Dodis et al. [12] introduced the collision-resistant compression function $h^{f_1,f_2}(x_1,x_2) = f_1(x_1) \oplus f_2(x_2)$ where f_1 and f_2 are two independent random oracles. Shrimpton and Stam [35] also proposed a rate-3 collision-resistant compression function $h^{f_1,f_2,f_3}(x_1,x_2) = f_3(f_1(x_1) \oplus f_2(x_2)) \oplus f_1(x_1)$.

Domain Extension. Given the utility of hash functions in cryptography, it is natural to investigate techniques for their systematic construction and validation. A well-known systematic approach involves the use of the *Merkle-Damgård (MD) transform* [11, 29], which allows the construction of variable-length hash functions from fixed-length hash functions (compression functions). MD and its variants are known to preserve collision resistance and so provide an approach based on focusing attention on the construction of compression functions. If we assume $h(y,m)$ is a compression function over two inputs where each input size is λ then in the original MD the message $m = m_1||m_2, \ldots, ||m_n$ length is a multiple of λ and may be defined as

$$y_1 = IV \quad H^h = y_n \quad y_i = h(y_{i-1}, m_{i-1})$$

Where IV is an initial value like 0^λ. While MD only handles messages with specific length, *strengthened* MD considers inputs with arbitrary length by adding some bits called *padding* to make the length of it a multiple of block size. Also, to make the construction collision resistant another block that represents the length of the original message is added after the padding. The following is the formal construction of the MD transform.

Definition 1.1.5. *Merkle-Damgård Transform [21]. Let (Gen_h, h) be a fixed-length hash function with input length $2l(\lambda)$. Construct a variable length hash function (Gen, H) as follows:*

- *$Gen(1^\lambda)$: upon input 1^λ, run the key generation algorithm Gen_h of the fixed-length hash function and output the key. That is, output $s \leftarrow Gen_h$*
- *$H^s(x)$: Upon input key s and message $x \in \{0,1\}^*$ of length at most $2^{l(\lambda)} - 1$, compute as follows:*
  1. *Let $L = |x|$ (the length of x) and let $B = \lceil \frac{L}{l} \rceil$. Pad x with zeroes so that its length is an exact multiple of l.*
  2. *Define $IV = y_0 := 0^l$ and then for every $i = 1, \cdots, B$, compute $y_i := h^s(y_{i-1} \,\|\, x_i)$, where h^s is the given fixed length hash function.*
  3. *Output $y = H^s(y_B \,\|\, L)$*

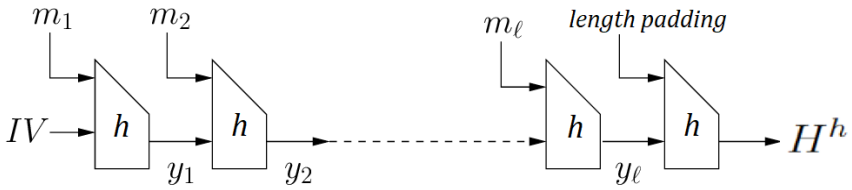

Figure 1.1: The Merkle-Damgård transform is a way of extending a fixed-length hash function (h) into a variable length that receives inputs of any length by iterating h over input blocks m_i.

Theorem 1.1.6. *[28, 11] If (Gen_h, h) is a collision-resistant compression function, then (Gen, H) is a collision-resistant hash function.*

The real-world hash functions MD5 and SHA-1 with 128 and 160-bit outputs are constructed using the MD transform. Even though MD preserves collision resistance, but it is still vulnerable to attacks. A well-known one is the *length-extension* attack. In particular, if an adversary has the value $H(m)$ then it may calculate $H^h(m||x) = h(H(m), x)$. This has practical implications, for example for a message-authentication code (MAC). MAC security requires that if an adversary learns a tag $t = \text{MAC}_k(m)$, for some m then it cannot infer anything about other messages' tags unless it knows the key. A MAC defined by $\text{MAC}_k(m) = H(k||m)$ will satisfy MAC security when H is a random oracle. However, if H is constructed from a compression function h using MD, security fails because once the adversary learns the tag t for a message m, then without knowing the key, it can deduce a valid tag for any message that has m as a prefix by calculating $\text{MAC}_k(m||x) = h(H(k||m), x) = h(t||m)$.

These weaknesses make MD-based hash functions unsuitable for many applications and demonstrate the need for notions of security stronger than collision resistance.

More broadly, as described above, the random oracle methodology involves using a constructed hash function in place of a random oracle in the implementation. The question of when this approach is sound led to the study of *indifferentiable* hash functions.

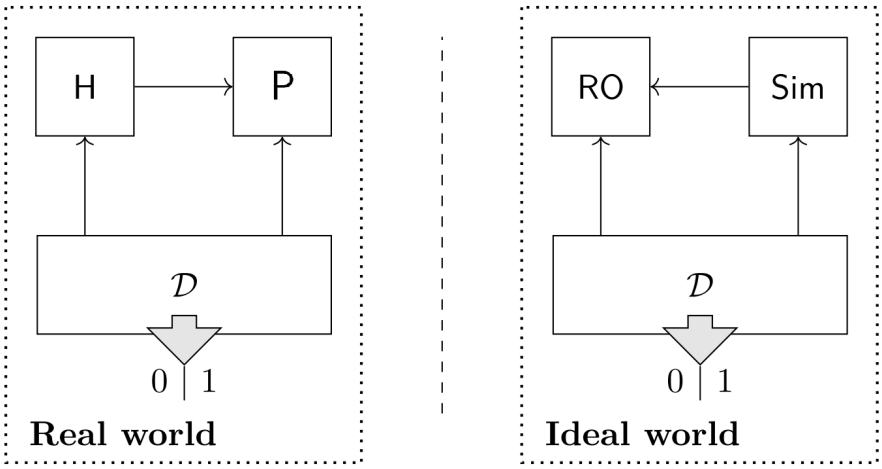

Figure 1.2: Indifferentiability setting [31]. H has oracle access to P and Sim has oracle access to RO, and distinguisher either interact with H and P or with RO and Sim.

1.1.1 Indifferentiability

Coron et al. [9] applied the indifferentiability framework introduced in [24] to define hash functions that are indifferentiable from random oracles.

Assume H is a variable length input hash function that utilizes a fixed length input ideal primitive P, and RO is a random oracle, so informally, if to any distinguisher $\mathcal{D}$, the hash function H^P and RO "look the same", then H^P called *indifferentiable* (from a random oracle). The main result of [9] is that if a cryptosystem is proven secure when H is a random oracle, it remains secure when it is instantiated with an indifferentiable hash function H^P. In this framework, $\mathcal{D}$ also has access to P and may interact either with H^P and P or RO and a *simulator* S which is trying to behave like the ideal primitive P while it also has access to RO, but it cannot see the distinguisher queries to RO. Figure 1.2 shows the indifferentiability setting. This can be formalized as follows:

Definition 1.1.7. *A construction H with oracle access to an ideal primitive P is* (t_D, t_S, q, ϵ)-indifferentiable *from a random oracle RO if there exists a simulator S, such that for any distinguisher $\mathcal{D}$ it holds that*

$$\Pr[\mathcal{D}^{H,P} = 1] - \Pr[\mathcal{D}^{RO,S} = 1]| < \epsilon$$

Where the simulator has access to RO and runs in time at most t_S. The distinguisher makes q queries and runs in time $t_{\mathcal{D}}$.

In the above definition, ϵ is a negligible function of λ and t_D and t_S are bounded by a polynomial function of λ.

To understand this security notion better we include the following example from [17]. This is an outline of a generic attack on compression functions of the form of $f(h, m) = P_{K_P}(E_{K_E}(X_E))$ where P and E are two ideal ciphers and the function that maps the inputs (h, m) to the keys (K_P, K_E) is non-bijective and $X_E \in \{h, m, h \oplus m\}$. We also write $\overline{f}$ when the adversary does not know if this is the compression function or the random oracle, we also use the same notation for the ideal cipher and simulator.

1. The adversary picks two inputs (h_1, m_1) and (h_2, m_2) where the corresponding keys are equal $(K_{P1}, K_{E1}) = (K_{P2}, K_{E2})$.

2. If $X_{E1} = X_{E2}$:

   - Then if the adversary is making a query to the compression function then the results are equal and if it is to the RO then they are not equal.

3. If $X_{E1} \neq X_{E2}$:

   - The adversary picks an arbitrary (h, m) from the domain.
   - Makes a query to $\overline{f}$ and receives $\overline{f}(h, m)$.
   - For (h, m) it calculates K_E, K_P and X_E.
   - Query for $\overline{P_{K_P}^{-1}}(\overline{f}(h, m))$ and receives X_P. Query for $\overline{E_{K_E}^{-1}}(X_P)$ and receives X'_E.
   - If $X'_E = X_E$ then $\overline{f}$ is the compression function otherwise is the RO.

1.1.2 Preimage Awareness

In this section we will briefly review the definition and some important facts regarding preimage awareness as defined in [13]. Suppose H is a hash function built from an ideal primitive P (e.g., a random oracle or ideal cipher.) Preimage awareness formalizes the notion which states that an adversary who knows a "later useful" output z of H^P must "already know" (be aware of) a particular corresponding preimage x. This is formalized using an auxiliary function called an *extractor* and the following experiment: After interacting with P, the adversary $\mathcal{A}$ produces z in the range space of H. This z and the sequence of (query, response) pairs made by $\mathcal{A}$ to P called α are passed to the extractor $\mathcal{E}$, which then produces a value x in the domain of H (or $\perp$.) Note that $\mathcal{E}$ does not have access to P. Then $\mathcal{A}$ runs again and attempts to output a preimage x' such that $H^P(x') = z$ but $x \neq x'$. If any adversary (from some class) has only a small chance of winning in this experiment, H is preimage aware. In the general definition of preimage awareness, the adversary is allowed multiple adaptive rounds against the extractor. A schematic depiction of the general game is given in Figure 1.2. The more restrictive version we have described is called 1-PrA. We note that by [13], Theorem D.1, we may restrict attention to 1-PrA, up to a loss in adversary advantage which is linear in the number of calls to the extractor. So below we use the term PrA synonymously with 1-PrA. We recall two important facts (stated informally) which underlie the utility of PrA in the construction of indifferentiable hash functions:

1. Suppose $H^P : Dom \to Rng$ is a PrA hash function defined using ideal primitive P, and $R : Rng \to Rng$ is an independent fixed-input length random oracle. If we define G using P and R by $G^{P,R}(m) = R(H^P(m))$, then G is indifferentiable from a random oracle ([13], Theorem 4.1)

2. Suppose $h^P : \{0,1\}^{n+d} \to \{0,1\}^n$ is a PrA compression function defined using ideal primitive P, and H is a hash function defined by applying the strengthened Merkle-Damgård construction to h^P. Then H is PrA. ([13], Theorem 4.2)

As mentioned above, these facts explain the importance of PrA as part of a methodology for constructing hash functions with strong security properties. In particular, in order to construct indifferentiable hash functions we only need to be concerned with constructing PrA compression functions. Given this methodology for constructing indifferentiable hash functions, an important goal is the design and anal-

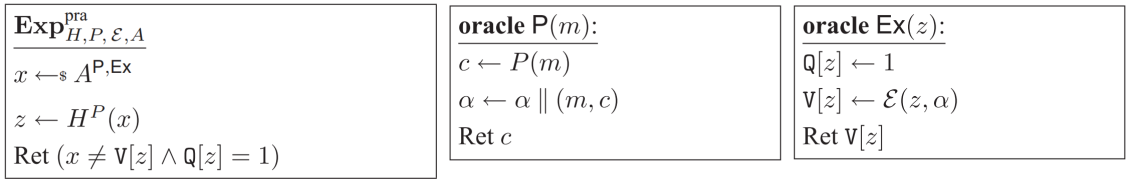

Figure 1.3: Preimage awareness game from [13]. Oracle P mediates access to the ideal primitive while recording all queries made, while Ex mediates access to the extractor while using Q, V to record the image/preimage pairs involved in each call.

ysis of PrA compression functions, *especially via methods which support automated verification, or even automated generation of such functions.*

In the following, we review the Linicrypt framework [8] as a tool to be employed to model hash functions via an algebraic characterization and to analyze security properties in a systematic and efficient way.

1.1.3 Linicrypt Programs

The Linicrypt formalism, introduced in [8], provides a model for the specification of cryptographic primitives via straight-line programs over a finite field $\mathbb{F}$, with access to a random oracle. In [8], it was shown that via an algebraic condition on programs, it is possible to efficiently decide whether two programs induce computationally indistinguishable distributions. The practical application of the approach was demonstrated through the use of a SAT solver to automatically synthesize programs used in a provably correct construction of garbled circuits.[3] While [8] focused on indistinguishability-based security for inputless Linicrypt programs, subsequent work [25] demonstrated the utility of Linicrypt for characterizing collision resistance and second-preimage resistance for fixed input-length hash functions constructed using a random oracle. This latter work is the starting point for the research presented in this thesis. Our presentation uses the model of [25], with some restrictions and slight variations which are more suited to our proof techniques. We denote elements of $\mathbb{F}$ by lower-case non-bold letters[4], vectors over $\mathbb{F}$ by lowercase bold letters and matrices over $\mathbb{F}$ by uppercase bold letters. We treat vectors as column vectors and write $\boldsymbol{a} \cdot \boldsymbol{b}$ or $\boldsymbol{a}^\top \boldsymbol{b}$ for the inner product and $\boldsymbol{M} \times \boldsymbol{a}$ or $\boldsymbol{M}\boldsymbol{a}$ for the matrix-vector product. Note

[3] More recently ([20]), Linicrypt was also used to characterize the security of block cipher modes of operation.

[4] We usually use Roman letters, but may also use Greek or script letters depending on the setting.

that we will sometimes think of matrices as a column vector of row vectors (so we may write $\boldsymbol{M} = (\boldsymbol{m}_1, \ldots, \boldsymbol{m}_k)^\top$.)

A Linicrypt program is a straight-line program over a fixed vector $(v_1, \ldots, v_m)$ of *program variables*, where the first k are designated as *inputs*. A program is a sequence of lines specifying assignments to (non-input) program variables where the right-hand side of each assignment is either[5]

1. A call to the random oracle on a previously assigned variable or input
2. A $\mathbb{F}$-linear combination of previously assigned variables and inputs

The program also specifies a vector of output variables. For a given random oracle H, such a program computes a function $\mathbb{F}^k \to \mathbb{F}^r$ for some k, r.

The following is an example of a two-input Linicrypt program with random oracle H,

$$\begin{aligned}
&\underline{\mathcal{P}^H(v_1, v_2):}\\
&\quad v_3 := H(t_1, v_1)\\
&\quad v_4 := H(t_2, v_2)\\
&\quad v_5 := v_3 + v_4\\
&\quad \mathit{return}\ v_5
\end{aligned}$$

We allow the random oracle to take an extra input from a designated set of strings or *nonces* (in this example, these are $t_1 \neq t_2$.) When $\mathbb{F} = \mathbb{GF}(2^\lambda)$, this program computes the function $f(x, y) = f_1(x) \oplus f_2(y)$ where f_1 and f_2 are two independent random oracles mapping $\mathbb{F}$ to $\mathbb{F}$. This is the Dodis-Pietrzak-Puniya function [12], which is one of the compression functions shown to be PrA in [13]. We will return to this example after giving our characterization of PrA and verifying that it satisfies the critical query condition in Chapter 2.4. Other examples of Linicrypt programs may be found in Section 2.4.1. We have noted that a program has access to a random oracle H, that is, a random element of $\{F \mid F : \mathsf{nonce} \times \mathbb{F} \to \mathbb{F}\}$, where $\mathsf{nonce} = \{t_i \mid i \in \mathbb{N}\}$. As in [25], we will require that for $i \neq j$, $t_i \neq t_j$, *so that all calls to the random oracle in a program $\mathcal{P}$ are effectively to an independent random function.* We note that this restriction is equivalent to assuming that functions are defined using a fixed

[5]The Linicrypt model, introduced in [8], also allows the assignment of a random field element to a variable. Here, as in [25], we consider only deterministic programs.

collection of independent random oracles, where each may be called exactly once. This is adequate for defining a number of the compression functions considered by [13]. While it is not a general restriction for the Linicrypt model, we will only consider random oracles that take a single input from $\mathbb{F}$. This will simplify the presentation and seems adequate for the application to PrA hash functions. See [25], Section 5.1, for a discussion of extending the model to random oracles with multiple field inputs.

As noted in [25], the cryptographic power of this model derives from the random oracle, and so we require a field size $|\mathbb{F}|$ that is exponential in the security parameter λ. Furthermore, since a program is specified by linear combinations of elements with coefficients from $\mathbb{F}$, programs may also depend on the security parameter. On the other hand, we could either consider a family of programs parameterized by λ, or fix a subfield of $\mathbb{F}$ for the coefficients (e.g., if the field is $\mathbb{GF}(p^k)$, we take coefficients from $\mathbb{GF}(p)$) and work with a single program that can be instantiated over any field. Again as noted in [25], in the concrete setting of their work (and ours), this choice is not significant. In all the examples given, we work over $\mathbb{GF}(2^\lambda)$ and assume that programs work with coefficients from $\{0, 1\}$.

An important observation of [8] is that it is possible to present programs in a purely algebraic fashion. In this view, Linicrypt program $\mathcal{P}$ over a field $\mathbb{F}$ is given by a set of *base variables*, corresponding to program inputs and results of oracle queries, a sequence of *oracle constraints* specifying the input and output of each query, and an *output matrix*. The base vectors are represented as canonical basis vectors, so for a $\mathcal{P}$ with k inputs and n oracle queries the base vectors are $\mathbf{e}_1, \ldots, \mathbf{e}_k, \mathbf{e}_{k+1}, \ldots, \mathbf{e}_{k+n}$ where, $\mathbf{e}_i$ denotes the ith canonical basis vector over $\mathbb{F}^{k+n}$ without loss of generality, $\mathbf{e}_1, \ldots, \mathbf{e}_k$ correspond to the inputs and $\mathbf{e}_{k+1}, \ldots, \mathbf{e}_{k+n}$ correspond to the results of oracle queries. As program variables are linear combinations of base variables with coefficients from $\mathbb{F}$, they are represented as elements of $\mathbb{F}^{n+k}$. In fact, by a process of successive in-lining, we may eliminate the notion of non-base variables altogether. So we need only consider assignments of the first sort described above, but where the second argument is given by a $\mathbb{F}$-linear combination of inputs and results of previous calls. With this perspective, each oracle call is represented by an *oracle constraint* $c = (t, \boldsymbol{q}, \boldsymbol{a})$ where $t \in \mathsf{nonce}$ and $\boldsymbol{q}, \boldsymbol{a} \in \mathbb{F}^{k+n}$ (note that $\boldsymbol{a}$ here will in fact be some $\mathbf{e}_j$ where $k < j \leq k + n$). Similarly, the output may be given as a vector of $\mathbb{F}$-linear combinations of program variables, and this may be specified by a matrix $\boldsymbol{M} = (\boldsymbol{m}_1^\top, \ldots, \boldsymbol{m}_r^\top)^\top$, where $\boldsymbol{m}_i \in \mathbb{F}^{k+n}$. In [8], the underlying sequential structure of $\mathcal{P}$ is essentially forgotten, and a program is represented by its output

matrix and (multi)set of oracle constraints. In order to simplify some of our proofs, we will deviate slightly and remember the order of oracle calls via a sequence $\mathcal{C} = \langle c_i \mid 1 \leq i \leq n \rangle$ of constraints, where $c_i = (t_i, \boldsymbol{q}_i, \boldsymbol{a}_i)$. In particular, this implies that $\boldsymbol{q}_i \in \mathsf{span}(\mathbf{e}_1, \ldots, \mathbf{e}_k, \boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1})$, i.e., the input to a query depends only on the program inputs and answers to previous queries. We note that retaining the ordering of queries is not essential to our results, but we do so to simplify the presentation.

We introduce the following notation (not used by [25]) for matrices related to the base variables and queries of $\mathcal{P}$: $\boldsymbol{A}$ for $(\boldsymbol{a}_1^\top, \ldots, \boldsymbol{a}_n^\top)^\top$, $\boldsymbol{Q}$ for $(\boldsymbol{q}_1^\top, \ldots, \boldsymbol{q}_n^\top)^\top$, and $\boldsymbol{X}$ for $(\mathbf{e}_1^\top, \ldots, \mathbf{e}_k^\top)^\top$. For any matrix $\boldsymbol{M}$, we write $\mathsf{rows}(\boldsymbol{M})$ for the multiset of vectors corresponding to the rows of $\boldsymbol{M}$.

Viewing $\mathcal{P}$ as a straight-line program provides a simple semantics defining $\mathcal{P}^H : \mathbb{F}^k \to \mathbb{F}^r$. For an oracle H and input $\boldsymbol{x} = (x_1, \ldots, x_k)^\top$, a line whose right-hand side is a linear expression evaluates to an element of $\mathbb{F}$ in a natural way, using previously defined elements. For an oracle call $H(t_i, v_j)$, v_j has already been assigned a value from $\mathbb{F}$ and so the call returns a value in $\mathbb{F}$. In both cases, the resulting value is assigned to the left-hand side variable. The final line outputs the vector of field elements assigned to the corresponding variables, which is the value of $\mathcal{P}^H(\boldsymbol{x})$. Algebraically, for any input $\boldsymbol{x} = (x_1, \ldots, x_k)^\top$ and oracle H there is a corresponding vector $\boldsymbol{v}_{base} = (v_1, \ldots, v_{k+n}) \in \mathbb{F}^{k+n}$ which satisfies $\mathbf{e}_i \cdot \boldsymbol{v}_{base} = x_i$ for $1 \leq i \leq k$ and $\mathbf{e}_{k+i} \cdot \boldsymbol{v}_{base} = \boldsymbol{a}_i \cdot \boldsymbol{v}_{base} = H(t_i, \boldsymbol{q}_i \cdot \boldsymbol{v}_{base})$. In this view, $\mathcal{P}^H(\boldsymbol{x}) = \boldsymbol{M} \times \boldsymbol{v}_{base}$. It is not hard to see that both views give equivalent semantics.

In the algebraic presentation, the program given above is specified by:

$$\boldsymbol{v}_{base} = (v_1, v_2, v_3, v_4)^\top$$
$$\boldsymbol{M} = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$$
$$\mathcal{C} = \langle (t_1, \boldsymbol{q}_1, \boldsymbol{a}_1), (t_2, \boldsymbol{q}_2, \boldsymbol{a}_2) \rangle$$

where

$$\boldsymbol{q}_1 = (1, 0, 0, 0)^\top \quad \boldsymbol{q}_2 = (0, 1, 0, 0)^\top$$
$$\boldsymbol{a}_1 = (0, 0, 1, 0)^\top \quad \boldsymbol{a}_2 = (0, 0, 0, 1)^\top$$

1.2 Previous Work

The Linicrypt formalism, as introduced in reference [8], offers a framework for specifying cryptographic primitives through linear programs operating within a finite field $\mathbb{F}$, while also incorporating a random oracle. The initial work [8] revealed that, by imposing an algebraic condition on these programs, it becomes feasible to efficiently determine whether two programs generate distributions that are computationally indistinguishable.

Later McQuoid et al. in [25] characterize collision resistance and 2nd-preimage resistance in terms of the algebraic representation for the class of Linicrypt programs. In particular, they define an algebraic property of Linicrypt programs referred to as a collision structure, the absence of which characterizes both second-preimage resistance and collision resistance. The detection of a collision structure within a program $\mathcal{P}$ can be accomplished in polynomial time, taking into consideration the size of $\mathcal{P}$'s algebraic representation. As discussed above, this work assumes a specific class of Linicrypt programs where each random call has the form of $H(t, x)$ and the first input is a unique nonce and the second input is the oracle input, these unique nonces make all the queries independent. The next example gives such a program, along with its algebraic representation.

Example 1.2.1. *A Linicrypt program and its matrix representation:*

$$
\begin{aligned}
&\mathcal{P}^H(v_1, v_2, v_3, v_4) : \\
&v_5 = H(t_1, v_1) \\
&v_6 = H(t_2, v_2) \\
&v_7 = H(t_3, v_3) \\
&return\ (v_5 + v_4, v_6)
\end{aligned}
$$

$$\begin{array}{c} \boldsymbol{q}_1 \\ \boldsymbol{q}_2 \\ \boldsymbol{q}_3 \\ \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \boldsymbol{a}_3 \\ \boldsymbol{m}_1 \\ \boldsymbol{m}_2 \end{array} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ l_1 \\ l_2 \end{bmatrix}$$

Thus:

$$\boldsymbol{M} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \ \boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \ \boldsymbol{Q} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathcal{C} = \begin{Bmatrix} (t_1, [1,0,0,0,0,0,0], [0,0,0,0,1,0,0]) \\ (t_2, [0,1,0,0,0,0,0], [0,0,0,0,0,1,0]) \\ (t_3, [0,0,1,0,0,0,0], [0,0,0,0,0,0,1]) \end{Bmatrix}$$

The next example from [25] is a motivating illustration, elucidating the concept of the collision structure as defined in Definition 1.2.3.

Example 1.2.2.

$$\begin{aligned} &\mathcal{P}^H(v_1, v_2, v_3): \\ &v_5 = H(t_1, v_1) \\ &v_6 = H(t_2, v_3) \\ &v_7 = v_5 + v_6 + v_2 \\ &v_8 = H(t_3, v_7) \\ &return\ (v_8 + v_1, v_6) \end{aligned}$$

Here we go over a possible attack by a 2nd preimage adversary $\mathcal{A}$. Given (x, y, z) as the input of the program, the adversary returns a 2nd-preimage (x', y', z'):

1. *$\mathcal{A}$ runs the program on (x, y, z) and gets (l_1, l_2) as the programs output.*

2. *The second component of $\mathcal{P}$'s output is $H(t_2, z)$ and it is not possible to find a different input for the random oracle that returns the same output. Thus, the adversary picks $z' = z$.*

3. *$\mathcal{A}$ picks $v_7' \neq v_7$ and makes the query $v_8' = H(t_3, v_7')$ and finds $x' = l_1 + v_8'$.*

4. *Now, $H(t_1, x')$ and $H(t_3, z')$ and $v_7' = H(t_1, x') + H(t_3, z') + y'$ are fixed and the adversary picks y' so the equality holds, and it returns (x', y', z').*

In this example, we can see the adversary had to pick the same query value for $H(t_2, v_3)$ in both computations. The adversary identifies the first query which assigns a different value and makes the followup oracle queries to find the intermediate values by solving linear equations.

The next definition, from [25], formalizes the algebraic conditions for a collision structure

Definition 1.2.3. *Let $\mathcal{P} = (\boldsymbol{M}, \mathcal{C})$ be a Linicrypt program. A* collision structure *for $\mathcal{P}$ is a tuple $(i^*, c_1, \cdots, c_n)$ where:*

1. *$c_1, \cdots, c_n$ is an ordering of $\mathcal{C}$, and we write $c_i = (t_i, \boldsymbol{q}_i, \boldsymbol{a}_i)$.*

2. *$\boldsymbol{q}_{i^*} \notin \mathsf{span}(\{\boldsymbol{q}_1, \cdots, \boldsymbol{q}_{i^*-1}\} \cup \{\boldsymbol{a}_1, \cdots, \boldsymbol{a}_{i^*-1}\} \cup rows(\boldsymbol{M}))$*

3. *For $j \geq i^* : \boldsymbol{a}_j \notin \mathsf{span}(\{\boldsymbol{q}_1, \cdots, \boldsymbol{q}_j\} \cup \{\boldsymbol{a}_1, \cdots, \boldsymbol{a}_j\} \cup rows(\boldsymbol{M}))$*

The main result in [25] is the following theorem.

Theorem 1.2.4. *(Main Theorem) Let $\mathcal{P}$ be a deterministic Linicrypt program with distinct nonces, making n oracle queries. Let $\mathbb{F}$ be the underlying field (and range of the random oracle). Then the following are equivalent:*

1. *There is an adversary $\mathcal{A}$ making q oracle queries that finds collisions with probability more than $(q/n)^{2n}/|\mathbb{F}|$.*

2. *There is an adversary $\mathcal{A}$ making q oracle queries that finds second preimages with probability more than $(q/n)^{n}/|\mathbb{F}|$.*

3. *There is an adversary $\mathcal{A}$ making at most $2n$ oracle queries that finds second preimages with probability 1.*

4. *$\mathcal{P}$ either has a collision structure or is degenerate.*

1.3 Contributions

In this thesis, we extend the results of [25] by considering stronger notions of hash function security and also adapting their framework to the ideal cipher model. We also show that the approach of [25] leads directly to automated methods for verification and generation of secure compression functions. The following are our main contributions:

1. A simple and efficiently-checkable property of Linicrypt programs that characterizes preimage awareness is given.

2. A demonstration of the usefulness of (1) by showing that the characterization may be efficiently automated and, as a special case, using this implementation to enumerate all preimage-aware compression functions that use two calls to the random oracle.

3. An extension of the Linicrypt framework to support constructions in the ideal cipher model.

4. A characterization of collision- and 2nd-preimage-resistance using an adaptation of the definition of collision structure to the ideal cipher model.

5. An efficient algorithm for determining whether a program in the ideal cipher model satisfies the condition of being collision-resistant.

6. As of the application of (4), we consider the case of the block cipher- based hash functions proposed by Preneel, Govaerts, and Vandewall [32] and show that the semantic analysis of PGV given by Black et al. [5] can be captured as a special case of the characterization.

7. Examining constructions in Linicrypt that are not initially collision-resistant, but become collision-resistant after applying the MD transformation. The group-2 PGV compression functions fall under this category of construction.

8. Characterizing collision resistance for double-block-length compression functions.

These results demonstrate the utility of [25] as a general approach to modeling, verification, and generation of compression functions built using linear operations in both

the random oracle model and the ideal cipher model. In the case of PrA compression functions in the random oracle model, and collision-resistant double-block-length functions in the ideal cipher model, the approach we present identified many functions that were not previously known in the literature, suggesting candidates for further study and, potentially, for implementation.

1.4 Thesis Overview

The outline of this dissertation is as follows.

- **Chapter 2** Characterizes preimage awareness security notion in Linicrypt and generates PrA compression function with two inputs and two random oracle calls.
- **Chapter 3** Extends Linicrypt to support constructions in the ideal cipher model, and characterizes collision-resistant programs in this setting.
- **Chapter 4** Explores two additional categories of Linicrypt programs in the ideal cipher model

  4.1 Gives a characterization of programs that define compression functions for which MD produces a collision-resistant hash function. This includes the group-2 functions of [4]

  4.2 Investigates double-block-length (DBL) programs and gives a characterization of collision-resistant DBL programs.
- **Chapter 5** Concludes the dissertation and discusses potential future work.
- **Appendix** Appendix A.1, includes additional information and theorems. In Appendix A.2, you can explore the implementations of algorithms for verifying and generating PrA programs in the random oracle model, and collision-resistant programs (single block length and DBL) in the ideal cipher model.

The results of Chapter 2 appear in

> Zahra Javar and Bruce M. Kapron, "Preimage Awareness in Linicrypt", *36th IEEE Computer Security Foundations Symposium, CSF 2023*, pp. 33-42.

while those of Chapter 3 appear in

Zahra Javar and Bruce M. Kapron, "Linicrypt in the Ideal Cipher Model", *Provable and Practical Security - 17th International Conference, ProvSec 2023* , pp. 91-111.

Chapter 2

Preimage Awareness in Linicrypt

This chapter presents an approach using the Linicrypt framework [8], which allows constructions specified by straight-line programs over a finite field, with access to a random oracle to characterize preimage awareness (PrA) compression functions. The approach is built on the methodology of [25] in order to characterize PrA compression functions defined by Linicrypt programs. The following paragraphs delve into the historical context and background of hash functions and PrA security notion, elucidating the significance of PrA and their characterization within the Linicrypt framework.

Given the utility of hash functions in cryptography, it is natural to investigate techniques for their systematic construction and validation. A well-known systematic approach involves the use of the *Merkle-Damgård (MD) transform* [11, 29], which allows the construction of variable-length hash functions from fixed-length hash functions (compression functions). MD and its variants are known to preserve collision resistance and so provide an approach based on focusing attention on the construction of compression functions. On the other hand, well-known attacks such as length-extension make MD-based hash functions unsuitable for many applications, and demonstrate the need for notions of security stronger than collision-resistance apparent.

More broadly, following the work of [3], a large number of constructions make heuristic use of hash functions under the assumption that they behave as a random oracle. Although this methodology is not sound in general [7], it has proven useful as a means of providing evidence that construction does not have certain structural flaws.

At this point, a natural question is whether it is possible to detect flaws in the

design of a concrete hash function which renders it unsuitable as an instantiation of a random oracle. For hash functions constructed via the MD transform [9] propose an approach based on constructions which, under the assumption that the underlying compression function is a random oracle (or ideal cipher), produce a hash function which is *indifferentiable* from a random function in the sense of [24]. They were able to show that, while strengthened MD did not satisfy this criterion, straightforward modifications did. However, this approach would not apply to the real-world scenario of hash functions constructed using MD and therefore did not provide an explanation for why such constructions appear to work in practice, or a justification for the soundness of the random oracle model in such settings.

This problem was addressed by [13], who define a weaker property – *preimage awareness (PrA)* – that is preserved by MD. Furthermore, they show that composing a PrA function with a fixed-input-length (independent) random oracle produces a function that is indifferentiable from a random oracle. Informally, preimage awareness means that if an adversary knows an image and later learns a preimage for that image, it is presumed to already know that preimage — a formal definition is given in the background section in Chapter 1.

This suggests the following methodology for the construction of indifferentiable hash functions: (1) Define a compression function h using a random oracle and prove that it is PrA. (2) Apply strengthened MD to this compression function to produce a variable-input-length function H^h which is PrA. (3) Define a variable-input-length indifferentiable function by composing the result of (2) with an independent fixed-input-length random oracle R.

Given this methodology for constructing indifferentiable hash functions, an important goal is the design and analysis of PrA compression functions, *especially via methods that support automated verification, or even automated generation of such functions.* This is the main problem addressed in this chapter.

The intuition behind the PrA characterization is the following: if the result of at least one call to the random oracle is linearly independent of the other calls as well as the output (in a sense which will be made precise,) an adversary can carry out an attack that breaks PrA as follows: produce an output without making the independent call, and later, once the call is made, produce an input consistent with the previously produced output. Formalization of this concept is achieved through the notion of *PrA critical query* as defined in Definition 2.4.1. The complete characterization of preimage awareness is demonstrated in Theorem 4.2.4 via this syntactic

condition. Additionally, solving the problem of identifying critical queries is simplified to the extent that it becomes a matter of solving a system of linear equations, ensuring its polynomial time solvability. To illustrate this characterization's practical application, an example is provided, showcasing how it can be used to enumerate all PrA compression functions with two inputs and the utilization of two independent random oracle calls.

Contributions The main contributions of this chapter are summarized as follows:

1. Section 2.4 gives an algebraic characterization of PrA for hash functions modeled as Linicrypt programs which have access to a random oracle that is sound, complete, and efficiently checkable.

2. Using this characterization, all PrA compression functions with two inputs and two random oracle calls are automatically generated, as listed in Section 2.4.2.

3. Section 2.1 explores the modeling of constant values within Linicrypt, and in Section 2.4.2, PrA compression functions that utilize a constant value are presented.

In general, the existence of a PrA critical query may be automatically verified (in polynomial time.) This contrasts with known proofs of PrA for the Shrimpton-Stam and Dodis-Pietrzak-Puniya compression functions [13], which are quite involved and specific to these particular definitions. Practically, we can automatically generate low-rate PrA compression functions, which are then suitable as a component in the above-described pipeline for constructing indifferentiable hash functions.

2.1 Dealing with Constant Values in Linicrypt

In practice, the definition of a hash function may depend on the use of a constant value from the underlying field or domain, typically referred to as an *initialization vector (IV)*. In their classification of rate-1 compression functions in the ideal cipher model, Black et al. [5] include definitions that use a constant value. Such definitions involve affine expressions and so, strictly speaking, are beyond the model provided by Linicrypt. One approach to dealing with this problem is to utilize some underlying algebraic property of operations involving constant values. This is the approach taken in the algebraic analysis of [36], where it is noted that translation by a constant

preserves bijectivity. In this section, a more general approach is undertaken, treating constants *parametrically*. Namely, a constant c used in a program $\mathcal{P}$ is treated as an additional input and also as an output of the program, making it a fixed parameter. In particular, $\mathcal{P}$ has base variables $x_1, \ldots, x_{k+n}$ and inputs $x_1, \ldots, x_k$ the modified program has base variables $x_1, \ldots, x_{k+n+1}$ where $\mathcal{P}$'s base variables $x_{k+1}, \ldots, x_{k+n}$ are (respectively) renamed $x_{k+2}, \ldots, x_{k+n+1}$, input x_{k+1} is used to represent the constant c, and $\boldsymbol{M}$ and $\mathcal{C}$ are updated appropriately. Finally, the single row $\mathbf{e}_{k+1}^{\top}$ is appended to $\boldsymbol{M}$ (indicating that c is an output.) With this convention, we can analyze the security properties of hash functions defined by Linicrypt programs using constants without any changes to the definitions or proofs. Moreover, any property which does not depend on a particular property (e.g., the bit-level representation) of a constant value used in a program will be preserved by this convention. While this does not capture implementation-level details, it provides a level of analysis consistent with works such as [5]. The following example shows how a compression function with a constant value c can be modeled in Linicrypt.

Example 2.1.1. *Consider compression function $f(x_1, x_2) = f_1(x_1 + c) + f_2(x_2) + c$, this can be presented as the following Linicrypt program,*

$$
\begin{aligned}
&\underline{\mathcal{P}^H(v_1, v_2, v_3):} \\
&\quad v_4 := v_1 + v_3 \\
&\quad v_5 := H(t_1, v_4) \\
&\quad v_6 := H(t_2, v_2) \\
&\quad v_7 := v_5 + v_6 + v_3 \\
&\quad \textit{return } v_7
\end{aligned}
$$

$$
\boldsymbol{v}_{base} = (x_1, x_2, c, v_5, v_6)^{\top}
$$

$$
\boldsymbol{M} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}
$$

$$
\mathcal{C} = \langle (t_1, \boldsymbol{q}_1, \boldsymbol{a}_1), (t_2, \boldsymbol{q}_2, \boldsymbol{a}_2) \rangle
$$

where

$$
\boldsymbol{q}_1 = (1, 0, 1, 0, 0)^{\top} \quad \boldsymbol{q}_2 = (0, 1, 0, 0, 0)^{\top}
$$

$$
\boldsymbol{a}_1 = (0, 0, 0, 1, 0)^{\top} \quad \boldsymbol{a}_2 = (0, 0, 0, 0, 1)^{\top}
$$

2.2 Defining Security Properties of Programs

Any $\mathcal{P}$ with k inputs and r outputs defines a distribution on the set of functions from $\mathbb{F}^k$ to $\mathbb{F}^r$ which is sampled by returning $\mathcal{P}^H$ for a uniformly sampled H, and so we can consider the security properties of $\mathcal{P}$ by considering the property for the (randomized) function it defines. Here we will give explicit definitions for standard collision-resistance properties of $\mathcal{P}$ (originally defined in [25],) as well as for preimage awareness, using the 1-PrA formulation as discussed in Section 2.4.

In general, if we are working in a finite field $\mathbb{F}_\lambda$, the field size will be exponential in λ and we may take λ as our security parameter. The adversary $\mathcal{A}$ is defined with respect to λ; in particular, although we do not bound the running time of $\mathcal{A}$, there must be a polynomial p such that $\mathcal{A}$ makes at most $p(\lambda)$ queries. A program could be viewed as a uniform definition of a family of functions indexed by λ. As our characterization below is in the concrete setting (with respect to the advantage probability and the number of queries) we do not need to worry about these considerations.

Definition 2.2.1 ([25] Definition 2)**.** *Program $\mathcal{P}$ is* (q, ϵ)-collision resistant *if any oracle adversary $\mathcal{A}$ making at most q queries has probability of success at most ϵ in the following game:*

$$(\boldsymbol{x}, \boldsymbol{x}') \leftarrow \mathcal{A}^H(\lambda);\ \textit{return}\ (\boldsymbol{x} \neq \boldsymbol{x}')\ \textit{and}\ \mathcal{P}^H(\boldsymbol{x}) = \mathcal{P}^H(\boldsymbol{x}')$$

Definition 2.2.2 ([25] Definition 3)**.** *Program $\mathcal{P}$ is* (q, ϵ)-2nd-preimage resistant *if any oracle adversary $\mathcal{A}$ making at most q queries has probability of success at most ϵ in the following game:*

$$\boldsymbol{x} \leftarrow \mathbb{F}^k; \boldsymbol{x}' \leftarrow \mathcal{A}^H(\boldsymbol{x});\ \textit{return}\ (\boldsymbol{x} \neq \boldsymbol{x}')\ \textit{and}\ \mathcal{P}^H(\boldsymbol{x}) = \mathcal{P}^H(\boldsymbol{x}')$$

To define preimage awareness we first need the notion of *extractor*, which we will take to be a (deterministic) function $\mathcal{E} : (\mathbb{F}^2)^* \times \mathbb{F}^r \to \mathbb{F}^k \cup \{\perp\}$.

In the current setting (as in [25]) where we only consider the query complexity of adversaries, we do not restrict extractors to be computationally efficient. What is important is that $\mathcal{E}$ does not have access to the random oracle H. Note that we allow $\mathcal{E}$ to return the "undefined" value $\perp$.

Definition 2.2.3. *Program $\mathcal{P}$ is* (q, ϵ)-PrA *if there is an extractor $\mathcal{E}$ such that any oracle adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ making at most q queries has probability of success at*

most ϵ in the following game:

$$(\boldsymbol{qs}, \boldsymbol{\ell}, st) \leftarrow \mathcal{A}_1^H(\lambda);\ \boldsymbol{x} := \mathcal{E}(\boldsymbol{qs}, \boldsymbol{\ell});\ \boldsymbol{x}' \leftarrow \mathcal{A}_2^H(\boldsymbol{x}, \boldsymbol{qs}, \boldsymbol{\ell}, st);$$
$$\textit{return } (\boldsymbol{x} \neq \boldsymbol{x}') \textit{ and } \mathcal{P}^H(\boldsymbol{x}') = \boldsymbol{\ell}$$

where $\boldsymbol{qs} = (\alpha_1, \beta_1), \ldots, (\alpha_s, \beta_s)$ is the sequence of queries made by $\mathcal{A}_1$ to H and their corresponding responses, $\boldsymbol{\ell}$ is a purported output of $\mathcal{P}^H$ and st is a state variable *(hidden from $\mathcal{E}$).*

In each definition, the probability is over the random choices of $\mathcal{A}$ and random choice of $H : \mathsf{nonce} \times \mathbb{F} \to \mathbb{F}$.

Intuitively, the definition of 1-PrA captures an experiment which has three phases. In the first, the adversary $\mathcal{A}$ first makes a sequence $\boldsymbol{qs}$ of queries to H and produces a purported output $\boldsymbol{\ell}$ of $\mathcal{P}$. This output, as well as $\boldsymbol{qs}$ is then passed to the extractor $\mathcal{E}$, which returns a pre-image $\boldsymbol{x}$, or $\perp$, indicating failure. At this point, $\mathcal{A}$ is given $\boldsymbol{x}$, its previous outputs, and encoding of its previous state st, resumes execution. $\mathcal{A}$ succeeds if it can return a pre-image $\boldsymbol{x}'$ of $\boldsymbol{\ell}$ such that $\boldsymbol{x}' \neq \boldsymbol{x}$. When $\boldsymbol{x} = \perp$ this reduces to $\mathcal{A}$ returning some pre-image of $\boldsymbol{\ell}$.

2.3 Normalized Programs

On the way to characterizing programs that define PrA hash functions, we first note that some programs fail to be PrA trivially, due to easily checked conditions, and for the sake of a more straightforward characterization, we begin by ruling out such programs.

For any matrix $\boldsymbol{M} = (\boldsymbol{m}_1, \ldots, \boldsymbol{m}_k)^\top$ and $1 \leq i \leq k$, $\boldsymbol{M}_{-i}$ denotes $\boldsymbol{M}$ with its ith row removed, i.e.. $(\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{i-1}, \boldsymbol{m}_{i+1}, \ldots, \boldsymbol{m}_k)^\top$. This generalizes in the natural way to $\boldsymbol{M}_{-\mathcal{I}}$ for any $\mathcal{I} \subseteq [k]$. In particular $\boldsymbol{M}_{-\emptyset} = \boldsymbol{M}$.

A query is *useful* if its result is used as part of a query to another useful query, or as part of the final output of the program. We formalize the notion of *useless* query as follows:

Definition 2.3.1. *Define the inductive operator* $\mathsf{useless} : 2^{[n]} \to 2^{[n]}$ *by*

$$\mathsf{useless}(\mathcal{I}) = \{i \mid \boldsymbol{a}_i \notin \mathsf{span}(\mathsf{rows}(\boldsymbol{Q}_{-\mathcal{I}}) \cup \mathsf{rows}(\boldsymbol{A}_{-\mathcal{I}'}) \cup \mathsf{rows}(\boldsymbol{M}) \cup \mathsf{rows}(\boldsymbol{X}))\}$$

where $\mathcal{I}' = \mathcal{I} \cup \{i\}$. *Query* i *in program* $\mathcal{P}$ *is* useless *if* $i \in \bigcup_{j=1}^{n} \mathsf{useless}^j(\emptyset)$.

The definition of useless query corresponds to that of [8] but extends to the case of programs with inputs.

Lemma 2.3.2. *Let* $\mathcal{P}'$ *be obtained by removing all useless queries from* $\mathcal{P}$. *Then* $\mathcal{P}'$ *is* (q, ϵ)*-PrA iff* $\mathcal{P}$ *is* (q, ϵ)*-PrA.*

Proof. It suffices to show the result for the case that $\mathcal{P}'$ is obtained by removing a single query i for some $i \in \mathsf{useless}(\emptyset)$. To begin we note that for any H and any $\boldsymbol{x}$, $\mathcal{P}^H(\boldsymbol{x}) = \mathcal{P}'^H(\boldsymbol{x})$. But then any adversary $\mathcal{A}$ has the same PrA-advantage against $\mathcal{P}$ as it does against $\mathcal{P}'$. □

We also recall the following definition.[1]

Definition 2.3.3. *Program* $\mathcal{P}$ *is* degenerate *if*

$$\mathsf{span}(\{\mathbf{e}_1, \ldots, \mathbf{e}_{k+n}\}) \not\subseteq \mathsf{span}(\mathsf{rows}(\boldsymbol{Q}) \cup \mathsf{rows}(\boldsymbol{A}) \cup \mathsf{rows}(\boldsymbol{M}))$$

It is the case that if $\mathcal{P}$ is degenerate, a second preimage may be found with probability 1 [25], Lemma 5 and so such programs trivially fail to be PrA. We can also prove this directly:

Lemma 2.3.4. *If* $\mathcal{P}$ *is degenerate, then there is a PrA adversary* $\mathcal{A}$ *against* $\mathcal{P}$ *that succeeds with probability 1.*

Proof. The adversary $\mathcal{A}$ first picks an arbitrary input $\boldsymbol{x}$ and runs the program $\mathcal{P}$ on it, computing the corresponding base vector $\boldsymbol{v}_{base}$ and output value $\boldsymbol{\ell}$ and returns $(\boldsymbol{qs}, \boldsymbol{\ell}, st)$, where $\boldsymbol{qs}$ is the sequence of queries made by $\mathcal{P}$ and $st = \boldsymbol{v}_{base}$. Assume that on input $(\boldsymbol{qs}, \boldsymbol{\ell})$ the extractor $\mathcal{E}$ returns x — if it returns $\perp$ or $\boldsymbol{x}' \neq \boldsymbol{x}$, the adversary will return $\boldsymbol{x}$ and win. So the adversary must return a valid preimage $\boldsymbol{x}' \neq \boldsymbol{x}$.

Define the matrix $\boldsymbol{P} = \begin{bmatrix} \boldsymbol{Q} \\ \hdashline \boldsymbol{A} \\ \hdashline \boldsymbol{M} \end{bmatrix}$ where $\boldsymbol{Q}, \boldsymbol{A}, \boldsymbol{M}$ are the matrices defining $\mathcal{P}$. Then $\boldsymbol{P} \times \boldsymbol{v}_{base}$ is a vector consisting of $\mathcal{P}$'s queries and their answers and its final output. Suppose $\mathcal{A}$ can find a base vector $\boldsymbol{v}'_{base} \neq \boldsymbol{v}_{base}$ where $\boldsymbol{P} \times \boldsymbol{v}'_{base} = \boldsymbol{P} \times \boldsymbol{v}_{base}$. Then,

[1] Our definition corresponds to the version given in revision 2 of [?], correcting an earlier version (which is the version appearing in [25])

if $\boldsymbol{x}' = \boldsymbol{X} \times \boldsymbol{v}'_{base}$, it must be the case that $\boldsymbol{x}' \neq \boldsymbol{x}$, since the values of the remaining base variables are fixed for a given input. Since program $\mathcal{P}$ is degenerate, the rows of $\boldsymbol{P}$ cannot span all $k+n$ basis vectors which means $\text{rank}(\boldsymbol{P}) < k+n$, and thus, for some $\boldsymbol{v} \neq \boldsymbol{0}$, $\boldsymbol{P} \times \boldsymbol{v} = \boldsymbol{0}$. The adversary can solve for this $\boldsymbol{v}$ and set $\boldsymbol{v}'_{base} = \boldsymbol{v}_{base} + \boldsymbol{v}$. Then $\boldsymbol{P} \times (\boldsymbol{v}'_{base} - \boldsymbol{v}_{base}) = \boldsymbol{0}$, so $\boldsymbol{v}'_{base} \neq \boldsymbol{v}_{base}$ and $\boldsymbol{P} \times \boldsymbol{v}'_{base} = \boldsymbol{P} \times \boldsymbol{v}_{base}$. In particular, this allows $\mathcal{A}$ to compute $\boldsymbol{x}' \neq \boldsymbol{x}$ such that $\mathcal{P}(\boldsymbol{x}') = \mathcal{P}(\boldsymbol{x})$ □

Henceforth, we will assume (without loss of generality) that programs have no useless queries. Note that for any $\mathcal{P}$, useless queries may be removed in polynomial time (in the size of $\mathcal{P}$), using a standard reachability algorithm. We will also assume that programs are non-degenerate, a condition which can also be checked in polynomial time.

2.4 Characterizing Preimage Awareness

The main result of [25] gives an algebraic condition on Linicrypt programs which can be used to characterize collision and 2nd-preimage resistance. We will do the same for PrA: in Definition 2.4.1 we define the notion of *PrA-critical* query, and in Theorem 4.2.4 we use this to characterize preimage awareness. We begin by presenting a motivating example.

$$
\begin{array}{l}
\underline{\mathcal{P}^H(v_1, v_2):} \\
\quad v_3 := H(t_1, v_1) \\
\quad v_4 := H(t_2, v_1) \\
\quad \textit{return } v_3 + v_4 + v_2
\end{array}
$$

(this defines the function $f(x,y) = f_1(x) + f_2(x) + y$ where $f_1, f_2 : \mathbb{F} \to \mathbb{F}$ are independent random oracles.) Consider a PrA adversary $(\mathcal{A}_1, \mathcal{A}_2)$ that does the following: $\mathcal{A}_1$ chooses α_1 and ℓ at random from $\mathbb{F}$, calls $H(t_1, \alpha_1)$ to obtain β_1, and outputs $st = \alpha_1$, $\boldsymbol{qs} = (\alpha_1, \beta_1)$ and ℓ. Let $\boldsymbol{x}' = (\gamma_1, \gamma_2)$, where $\gamma_1 = \alpha_1$ and $\gamma_2 = \beta_1 + H(t_2, \alpha_1) + \ell$.

Then

$$\begin{aligned}\mathcal{P}^H(\boldsymbol{x}') &= H(t_1,\gamma_1) + H(t_2,\gamma_1) + \beta_1 + H(t_2,\alpha_1) + \ell \\ &= H(t_1,\alpha_1) + H(t_2,\alpha_1) + \beta_1 + H(t_2,\alpha_1) + \ell \\ &= \beta_1 + H(t_2,\alpha_1) + \beta_1 + H(t_2,\alpha_1) + \ell \\ &= \ell\end{aligned}$$

Moreover, given $st = \alpha_1$, $\mathcal{A}_2$ can call $H(t_2, \alpha_1)$ to obtain β_2 and so is able to return $(\alpha_1, \beta_1+\beta_2+\ell) = (\gamma_1, \gamma_2)$. Thus, *unless* the extractor can return $\boldsymbol{x}'$, $\mathcal{A}$ has a successful PrA attack. Suppose the extractor can compute $\boldsymbol{x}'$. Since $\mathcal{E}$ is given $(\alpha_1, \beta_1), \ell$, this means the $\mathcal{E}$ can compute $\gamma_2+\beta_1+\ell$, which is $H(t_2, \alpha_1)$. Since $\mathcal{E}$ does not have access to H, its probability of success is at most $1/|\mathbb{F}|$.

This attack succeeds because $\boldsymbol{a}_2 = (0,0,0,1)^\top$ is independent of $\{\boldsymbol{a}_1, \boldsymbol{m}_1\}$ where $\boldsymbol{a}_1 = (0,0,1,0)^\top$, $\boldsymbol{m}_1 = (0,1,1,1)^\top$ so the adversary may return an image without making this query. On the other hand, under the assumption that all queries are useful, a preimage cannot be determined without knowing the value of this query. Thus the extractor, without access to the random oracle, could at best guess its value. On the other hand, $\mathcal{A}_2$ is given $st = \alpha_1$ and so is able to make the query, and by non-degeneracy is guaranteed to obtain a preimage.

This suggests the following condition on program queries.

Definition 2.4.1. *Let $1 \le i^* \le n$. We say that i^* is* PrA-critical *(or just* critical*) for $\mathcal{P}$*

$$\boldsymbol{a}_{i^*} \notin \mathsf{span}(\mathsf{rows}(\boldsymbol{Q}) \cup \mathsf{rows}(\boldsymbol{A}_{-i^*}) \cup \mathsf{rows}(\boldsymbol{M})) \qquad (\dagger)$$

Theorem 4.2.4 characterizes PrA for non-degenerate Linicrypt programs using this simple algebraic condition. Before proceeding to this result, we give the algebraic characterization of collision and 2nd-preimage resistance given by [25]. Recall that in the presentation of [25], constraints are treated as an (unordered) set. We then have:

Definition 2.4.2 ([25], Definition 6)**.** *A* collision structure *for $\mathcal{P}$ is an ordering $c'_1, \cdots, c'_n$ of the (multiset of) constraints of $\mathcal{P}$ and $1 \le i^* \le n$ such that*

1. $\boldsymbol{q}'_{i^*} \notin \mathsf{span}(\{\boldsymbol{q}'_1, \ldots, \boldsymbol{q}'_{i^*-1}\} \cup \{\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_{i^*-1}\} \cup \mathsf{rows}(\boldsymbol{M}))$
2. *For $j \ge i^*$, $a'_j \notin \mathsf{span}(\{\boldsymbol{q}'_1, \ldots, \boldsymbol{q}'_j\} \cup \{\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_{j-1}\} \cup \mathsf{rows}(\boldsymbol{M}))$*

We have denoted the ith constraint in the collision structure ordering by $c'_i = (t'_i, \boldsymbol{q}'_i, \boldsymbol{a}'_i)$ to avoid confusion with our notation in which $c_i = (t_i, \boldsymbol{q}_i, \boldsymbol{a}_i)$ is the ith constraint in the ordering given by program $\mathcal{P}$. Note that we may view $(\boldsymbol{q}'_1, \ldots, \boldsymbol{q}'_n)$ and $(\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_n)$ respectively as permutations of the rows of $\boldsymbol{Q}$ and of $\boldsymbol{A}$. In particular, viewed as multisets, $\{\boldsymbol{q}'_1, \ldots, \boldsymbol{q}'_n\} = \mathsf{rows}(\boldsymbol{Q})$ and $\{\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_n\} = \mathsf{rows}(\boldsymbol{Q})$ The Main Theorem of [25] implies that $\mathcal{P}$ is $(q, (q/n)^{2n}/|\mathbb{F}|)$-collision resistant iff it is $(q, (q/n)^n/|\mathbb{F}|)$-2nd-preimage resistant iff it does not have a collision structure. We also note the following connection between collision structures and critical queries:

Lemma 2.4.3. *If program $\mathcal{P}$ has a collision structure, then it has a critical query.*

Proof. Suppose $(i^*, c'_1, \ldots, c'_n)$ is a collision structure for $\mathcal{P}$. Then, according to Definition 2.4.2 for $j \geq i^*$,

$$a'_j \notin \mathsf{span}(\{\boldsymbol{q}'_1, \ldots, \boldsymbol{q}'_j\} \cup \{\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_{j-1}\} \cup \mathsf{rows}(\boldsymbol{M})).$$

In particular, for $j = n$,

$$\boldsymbol{a}'_n \notin \mathsf{span}(\{\boldsymbol{q}'_1, \ldots, \boldsymbol{q}'_n\} \cup \{\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_{n-1}\} \cup \mathsf{rows}(\boldsymbol{M})).$$

Suppose $\boldsymbol{a}'_n$ corresponds to $\boldsymbol{a}_{i^\dagger}$ (i.e., the $i^\dagger$th row of $\boldsymbol{A}$.) We then have $\mathsf{rows}(\boldsymbol{Q}) = \{\boldsymbol{q}'_1, \ldots, \boldsymbol{q}'_n\}$ and $\mathsf{rows}(\boldsymbol{A}_{-i^\dagger}) = \{\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_{n-1}\}$, so

$$\boldsymbol{a}_{i^\dagger} \notin \mathsf{span}(\mathsf{rows}(\boldsymbol{Q}) \cup \mathsf{rows}(\boldsymbol{A}_{-i^\dagger}) \cup \mathsf{rows}(\boldsymbol{M})),$$

and so $i^\dagger$ is a critical query. □

This connection becomes significant in combination with the characterization of second-preimage resistance given by [25], in particular, the following

Lemma 2.4.4 ([25], Lemma 10)**.** *Let $\mathcal{P}$ be a Linicrypt program making n oracle queries, and $\mathcal{A}$ an adversary that makes at most N oracle queries. If $\mathcal{A}$ finds second-preimages with probability at least $(\frac{N}{n})^n/|\mathbb{F}|$ then $\mathcal{P}$ has a collsion structure.*

We use these two results as part of the proof of Lemma 2.4.7 below. While a direct proof would be possible, the use of Lemmas 2.4.3 and 2.4.4 makes things considerably simpler.

We note (not surprisingly) that the notion of critical query is conceptually (and computationally) simpler than the corresponding notion of collision structure, and in

particular, does not require re-ordering of constraints as they appear in the underlying program.

Lemma 2.4.5. *Suppose i^* is critical for $\mathcal{P}$. Then there is a randomized $\mathcal{A}_1$ that, given H, generates values*

$$\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_{i^*-1}, \beta_{i^*+1}, \ldots, \beta_n, \ell_1, \ldots, \ell_r$$

such that

1. *The only queries made by $\mathcal{A}_1$ are $H(t_i, \alpha_i)$, $1 \leq i \leq n$, $i \neq i^*$.*
2. *$\beta_i = H(t_i, \alpha_i)$, $1 \leq i \leq n$, $i \neq i^*$.*
3. *If $\beta_{i^*} = H(t_{i^*}, \alpha_{i^*})$, then the system*

$$\begin{bmatrix} \boldsymbol{Q} \\ \hdashline \boldsymbol{A} \\ \hdashline \boldsymbol{M} \end{bmatrix} \times \boldsymbol{v} = \begin{bmatrix} \boldsymbol{\alpha} \\ \hdashline \boldsymbol{\beta} \\ \hdashline \boldsymbol{\ell} \end{bmatrix}$$

   where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)^\top$, $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_n)^\top$, $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_r)^\top$, has a unique solution $(\gamma_1, \ldots, \gamma_{k+n})^\top \in \mathbb{F}^{k+n}$.

4. *$\mathcal{P}^H(\gamma_1, \ldots, \gamma_k) = (\ell_1, \ldots, \ell_r)$*
5. *If $\boldsymbol{q}_{i^*} \notin \mathsf{span}(\mathsf{rows}(\boldsymbol{Q}_{-i^*}) \cup \mathsf{rows}(\boldsymbol{A}_{-i^*}) \cup \mathsf{rows}(\boldsymbol{M}))$, then α_{i^*} is a random element of $\mathbb{F}$.*

Proof. For $1 \leq i \leq n$, suppose $\mathcal{A}_1$ has determined

$$\alpha_1, \ldots, \alpha_{i-1}, \beta_1, \ldots, \beta_{i^*-1}, \beta_{i^*+1}, \ldots, \beta_{i-1}.$$

It then does the following: determine whether

$$q_i \in \mathsf{span}(\{q_1, \ldots, q_{i-1}\} \cup \{a_1, \ldots, a_{i^*-1}, a_{i^*+1}, \ldots, a_{i-1}\}).$$

If it is, use the values determined by previous queries to determine α_i, otherwise, it chooses α_i uniformly from $\mathbb{F}$. If $i \neq i^*$, it then sets $\beta_i = H_i(t_i, \alpha_i)$. Finally, for $1 \leq j \leq r$, if $\boldsymbol{m}_j \in \mathsf{span}(\mathsf{rows}(\boldsymbol{Q}) \cup \mathsf{rows}(\boldsymbol{A}))$, $\mathcal{A}_1$ uses the values obtained in the

preceding steps to determine ℓ_j; otherwise it chooses ℓ_j uniformly from $\mathbb{F}$. Note that by (†), no $\boldsymbol{q}_i$ or $\boldsymbol{m}_j$ depends on $\boldsymbol{a}_{i^*}$, so α_i and ℓ_j may be determined as described.

It is immediate that (1), (2) and (5) are satisfied by this construction. Letting $\boldsymbol{P}$ denote the matrix in (3) first note that the values $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\ell}$ are chosen to respect any dependencies in $\boldsymbol{P}$, so that a solution exists. Non-degeneracy ensures that $\boldsymbol{P}$ is full rank, so the solution is unique. Once (3) is established, (4) follows from the fact that the β_i's are chosen using the corresponding calls to H. □

Lemma 2.4.6. *Suppose that $\mathcal{P}$ is a program with n query constraints. If there is a critical i^* for $\mathcal{P}$ then for any $\mathcal{E}$ there is a PrA adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ that makes n queries and succeeds with probability at least $1 - 1/|\mathbb{F}|$.*

Proof. The adversary $\mathcal{A}$ proceeds as follows:

1. $\mathcal{A}_1$ generates the values specified in Lemma 2.4.5 and outputs $st = \alpha_{i^*}$, $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_r)$ and

$$\boldsymbol{qs} = \langle (\alpha_1, \beta_1), \ldots, (\alpha_{i^*-1}, \beta_{i^*-1}), \\ (\alpha_{i^*+1}, \beta_{i^*+1}), \ldots, (\alpha_n, \beta_n) \rangle.$$

2. When $\mathcal{E}$ returns, $\mathcal{A}_2$ computes $\beta_{i^*} := H(t_{i^*}, \alpha_{i^*})$, solve the system given in condition (3) of the Lemma, and returns $(\gamma_1, \ldots, \gamma_k)$.

Unless $\mathcal{E}$ returns $(\gamma_1, \ldots, \gamma_k)$, $\mathcal{A}$ will win the PrA-game (Definition 2.2.3) with probability 1. So the only way that $\mathcal{E}$ can defeat $\mathcal{A}$ is by returning $(\gamma_1, \ldots, \gamma_k)$. We will show that $\mathcal{E}$ can do this with probability at most $1/|\mathbb{F}|$.

Write the system in Lemma 2.4.5 (3) as $\boldsymbol{Pv} = \boldsymbol{b}$. Since all the queries in $\mathcal{P}$ are useful, $\boldsymbol{P}$ must have at least one row other than a_{i^*} which is nonzero in its $(k + i^*)$th entry. We note that, for any Linicrypt program, this cannot be row q_{i^*}, because in general, the input to a query cannot depend on its output. Supposing that this row is the jth row, there exists j, $0 \leq j \leq 2n + r, j \neq n + i^*, i^*$ such that

$$\gamma_{k+i^*} + \sum_{1 \leq i \leq k+n, i \neq k+i^*} \nu_i \gamma_i = b_j$$

where $b_j \in \mathbb{F}$, $\nu_i \in \{0, 1\}$ and γ_{k+i}, $1 \leq i \leq n$, $i \neq i^*$, are all known to $\mathcal{E}$ (in particular, for $i \neq i^*$, $\gamma_{k+i} = \beta_i$.) Thus, if $\mathcal{E}$ can determine $(\gamma_1, \ldots, \gamma_k)$, it can solve the above equation and determine the only unknown, namely $\gamma_{k+i^*} = H_{i^*}(t_{i^*}, \alpha_{i^*})$. Since $\mathcal{E}$ does

not have access to H, and the fact that i^* satisfies (†), the probability of determining the correct $H_i^*(t_{i^*}, \alpha_{i^*})$ is at most $1/|\mathbb{F}|$. □

Lemma 2.4.7. *If for any $\mathcal{E}$ there is a PrA adversary $\mathcal{A}$ for Linicrypt program $\mathcal{P}$ making at most N oracle queries with success probability $> (\frac{N^{n+1}}{n^n})/|\mathbb{F}|$ then there is an i^* which is critical for $\mathcal{P}$.*

Proof. We begin by making some standard assumptions about $\mathcal{A}$. Before returning a preimage $\boldsymbol{\gamma}$, $\mathcal{A}^H$ makes all the queries made by $\mathcal{P}^H(\boldsymbol{\gamma})$, and never repeats any queries made to H.

Based on these assumptions, there is a mapping $T : [n] \to [N]$ such that the $T(i)$th query made by $\mathcal{A}^H$ corresponds to constraint c_i in the computation of $\mathcal{P}^H$. Letting N_i denote the number of queries made by $\mathcal{A}^H$ using nonce t_i, $1 \leq i \leq n$, we have that there are $\prod_{i=1}^n N_i$ possible mappings. Since $\sum_{i=1}^n N_i \leq N$, the product is maximized when $N_i = N/n$, so that there are at most $(N/n)^n$ possible mappings. Furthermore, there are at most N choices for t such that $\mathcal{A}^H$ calls the extractor after making its tth query. So there must be a specific T and t such that the success probability of $\mathcal{A}^H$ conditioned on its use of T and t is greater than $1/|\mathbb{F}|$. We may fix T and t and assume the successful adversary $\mathcal{A}$ uses them by modifying $\mathcal{A}$ so that it fails if this is not the case.

Let $i_1, \ldots, i_n$ be a permutation of $[n]$ with the property that $1 \leq j < k \leq n$ implies $T(i_j) < T(i_k)$, and let $s \in [n]$ be such that $T(i_s) \leq t$ and $T(i_{s+1}) > t$.

In particular, before calling $\mathcal{E}$, $\mathcal{A}^H$ makes the queries corresponding to $c_{i_1}, \ldots, c_{i_s}$ in $\mathcal{P}$. If any of $i_1, \ldots, i_s$ are critical, we are done. Otherwise, if $\mathcal{E}$ successfully returns a preimage, then by the assumption that $\mathcal{A}$ is a successful PrA adversary, it returns a second preimage with the probability $(\frac{N^{n+1}}{n^n})/|\mathbb{F}| \geq (\frac{N}{n})^n/|\mathbb{F}|$, so it follows by Lemma 2.4.4 that $\mathcal{P}$ has a collision structure, which by Lemma 2.4.3 implies that it has a critical i^*.

It remains to consider the case when $\mathcal{E}$ returns $\perp$. In this case, $\mathcal{A}^H$ must produce a preimage of the output $\ell_1, \ldots, \ell_r$ to which is committed in its call to $\mathcal{E}$. Assume that none of $i_{s+1}, \ldots, i_n$ are critical. In particular, i_n is not critical, so that

$$\boldsymbol{a}_{i_n} \in \mathsf{span}(\mathsf{rows}(\boldsymbol{Q}) \cup \mathsf{rows}(\boldsymbol{A}_{-i_n}) \cup \mathsf{rows}(\boldsymbol{M})). \qquad (*)$$

By the assumptions about $\mathcal{A}$ stated above, we may assume that a successful $\mathcal{A}^H$

returning a preimage $(\gamma_1, \ldots, \gamma_k)$ may also determine the vector

$$\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_k, \gamma_{k+1}, \ldots, \gamma_{k+n})^\top$$

corresponding to values of all the base variables for $\mathcal{P}^H$. It follows by $(*)$ that $\boldsymbol{a}_{i_n}$ must satisfy the equation

$$\boldsymbol{a}_{i_n} \cdot \boldsymbol{\gamma} = \sum_{1 \le i \le n} \rho_i \boldsymbol{q}_i \cdot \boldsymbol{\gamma} + \sum_{1 \le i \le n i \ne i_n} \sigma_i \boldsymbol{a}_i \cdot \boldsymbol{\gamma} + \sum_{1 \le j \le r} \tau_j \ell_j.$$

Note that all the values on the right-hand side of the equation are fixed, but $\boldsymbol{a}_{i_n} \cdot \boldsymbol{\gamma}$ is determined by choosing a random element of $\mathbb{F}$. This means that $\mathcal{A}^H$ succeeds with probability at most $1/|\mathbb{F}|$, a contradiction. □

Combining Lemmas 2.4.6 and 2.4.7 gives our main result:

Theorem 2.4.8. *Suppose $\mathcal{P}$ is a Linicrypt program making n queries, and $q \ge n$. For sufficiently large λ, the following are equivalent*

- *$\mathcal{P}$ is $(q, (q^{n+1}/n^n)/|\mathbb{F}|)$-PrA*
- *$\mathcal{P}$ does not have a critical query*
- *$\mathcal{P}$ is $(n, 1 - 1/|\mathbb{F}|)$-PrA*

Proof. Suppose that $\mathcal{P}$ is $(q, (q^{n+1}/n^n)/|\mathbb{F}|)$-PrA. Since $q \ge n$, this means that it is also $(n, (q^{n+1}/n^n)/|\mathbb{F}|)$-PrA. Choose λ so that $|\mathbb{F}| - 1 \ge q^{n+1}/n^n$. Then $\mathcal{P}$ is $(n, 1 - 1/|\mathbb{F}|)$-PrA. The remaining implications follow by Lemmas 2.4.6 and 2.4.7 □

2.4.1 Examples

Returning to the Dodis-Pietrzak-Puniya function given above, recall that $\boldsymbol{M} = \boldsymbol{m}_1^\top = (0, 0, 1, 1)$ and $\mathcal{C} = \langle (t_1, \boldsymbol{q}_1, \boldsymbol{a}_1), (t_2, \boldsymbol{q}_2, \boldsymbol{a}_2) \rangle$ where

$$\boldsymbol{q}_1 = (1, 0, 0, 0)^\top \quad \boldsymbol{q}_2 = (0, 1, 0, 0)^\top$$

$$\boldsymbol{a}_1 = (0, 0, 1, 0)^\top \quad \boldsymbol{a}_2 = (0, 0, 0, 1)^\top$$

Then $\boldsymbol{a}_1 = \boldsymbol{m}_1 + \boldsymbol{a}_2$ and $\boldsymbol{a}_2 = \boldsymbol{m}_1 + \boldsymbol{q}_2$, so that neither queries are critical.

As another example, we consider the Shrimpton-Stam hash function [35]. In [13] this function, defined for independent random oracles $f_1, f_2, f_3 : \mathbb{F} \to \mathbb{F}$ as $f(c, m) =$

$f_3(f_1(m) + f_2(c)) + f_1(m)$, is proven to be PrA. In Linicrypt we have the following program:

$$\begin{aligned}
&\underline{\mathcal{P}^H(v_1, v_2):}\\
&\quad v_3 := H(t_1, v_1)\\
&\quad v_4 := H(t_2, v_2)\\
&\quad v_5 := H(t_3, v_3 + v_4)\\
&\quad return\ v_3 + v_5
\end{aligned}$$

In the algebraic presentation, we have:

$$\boldsymbol{M} = \boldsymbol{m}_1^\top = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$
$$\mathcal{C} = \langle (t_1, \boldsymbol{q}_1, \boldsymbol{a}_1), (t_2, \boldsymbol{q}_2, \boldsymbol{a}_2), (t_3, \boldsymbol{q}_3, \boldsymbol{a}_3) \rangle$$

where

$$\boldsymbol{q}_1 = (1,0,0,0,0)^\top\ \boldsymbol{q}_2 = (0,1,0,0,0)^\top\ \boldsymbol{q}_3 = (0,0,1,1,0)^\top$$
$$\boldsymbol{a}_1 = (0,0,1,0,0)^\top\ \boldsymbol{a}_2 = (0,0,0,1,0)^\top\ \boldsymbol{a}_3 = (0,0,0,0,1)^\top$$

Here we have $\boldsymbol{a}_1 = \boldsymbol{m}_1 + \boldsymbol{a}_3$, $\boldsymbol{a}_2 = \boldsymbol{q}_3 + \boldsymbol{a}_1$, $\boldsymbol{a}_3 = \boldsymbol{m}_1 + \boldsymbol{a}_1$, so none of the queries are critical.

Finally, we give an example of a collision-resistant compression function which is not PrA. The program

$$\begin{aligned}
&\underline{\mathcal{P}^H(v_1, v_2):}\\
&\quad v_3 := H(t_1, v_1)\\
&\quad v_4 := H(t_2, v_1)\\
&\quad return\ (v_3 + v_2, v_4)
\end{aligned}$$

defines the function $f(x, y) = (f_1(x) + y, f_2(x))$. Here we have

$$\boldsymbol{q}_1 = (1,0,0,0)^\top \quad \boldsymbol{q}_2 = (1,0,0,0)^\top$$
$$\boldsymbol{a}_1 = (0,0,1,0)^\top \quad \boldsymbol{a}_2 = (0,0,0,1)^\top$$
$$\boldsymbol{m}_1 = (0,1,1,0)^\top \quad \boldsymbol{m}_2 = (0,0,0,1)^\top$$

It is clear by inspection that this function is collision-resistant due to the second component of the output. In terms of collision structures, we see that no such structure is possible as $\boldsymbol{a}_2 = \boldsymbol{m}_2$. On the other hand,

$$\boldsymbol{a}_1 \notin \mathsf{span}(\boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{a}_2, \boldsymbol{m}_1, \boldsymbol{m}_2),$$

so $i^* = 1$ is a critical query, giving rise to an attack where the adversary queries $\ell_2 = f_2(x)$, sets ℓ_1 arbitrarily, and outputs $(\ell_1, \ell_2) \in \mathsf{rng}(f)$. However, the extractor responds, in the second phase the adversary may compute $y = \ell_1 + f_1(x)$ and return (x, y), winning with probability at least $1 - 1/|\mathbb{F}|$.

Finding critical queries: Algorithmically, determining the existence of a critical query is straightforward: each $\boldsymbol{a}_i$ is checked by solving a $(2n - 1 + r) \times (n + k)$ linear system (or, more accurately determining the existence of a solution) – overall this is polynomial in the size of $\mathcal{P}$. We leave a more refined analysis of the algorithmic complexity of finding a critical query as future work.

2.4.2 Preimage Aware Compression Functions

In this section, we use our characterization to produce an enumeration of all preimage aware compression functions $f : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ with two inputs and two calls to a random oracle (where $\mathbb{F} = \mathbb{GF}(2^\lambda)$ and programs are restricted to having coefficients from $\{0, 1\}$.) To do this we generated all 2^{12} binary matrices with 3 rows and 4 columns, corresponding to constraint vectors $\boldsymbol{q}_1$, $\boldsymbol{q}_2$ and output vector $\boldsymbol{m}$ respectively ($\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ are fixed.) The first and second columns correspond to inputs x and y and the last two to the random oracle calls f_1 and f_2. After removing programs with useless queries and applying constraints that rule out programs that are degenerate or contain critical queries, we obtain 76 preimage-aware compression functions. This can be done very directly in a language that supports matrix operations.[2] Here we list 38 of them — the other 38 are obtained by switching the random oracle calls f_1 and f_2. These may be divided into three groups based on the general form of the function.

The first 12 functions are among those the form $f(x, y) = f_1(a) + f_2(b) + c$ or $f(x, y) = f_1(a) + f_1(b)$, where $a, b, c \in \{x, y, x + y\}$:

[2] A sample implementation in Octave is available at `https://github.com/zahrajavar/PrACompressionFunctions.git`

1. $f_1(x) + f_2(y) + y$
2. $f_1(x) + f_2(y) + x$
3. $f_1(x) + f_2(y) + x + y$
4. $f_1(x) + f_2(y)$
5. $f_1(x) + f_2(x + y) + y$
6. $f_1(x) + f_2(x + y) + x$
7. $f_1(x) + f_2(x + y) + x + y$
8. $f_1(x) + f_2(x + y)$
9. $f_1(y) + f_2(x + y) + y$
10. $f_1(y) + f_2(x + y) + x$
11. $f_1(y) + f_2(x + y) + x + y$
12. $f_1(y) + f_2(x + y)$

The next 14 PrA functions have the form $f(x, y) = f_1(a + f_2(b)) + f_2(b) + c$ or $f(x, y) = f_1(a + f_2(b)) + f_2(b)$ where a, b, c are as above:

13. $f_1(x + f_2(y)) + f_2(y)$
14. $f_1(x + f_2(y)) + f_2(y) + y$
15. $f_1(x + f_2(x + y)) + f_2(x + y)$
16. $f_1(x + f_2(x + y)) + f_2(x + y) + x$
17. $f_1(x + f_2(x + y)) + f_2(x + y) + x + y$
18. $f_1(y + f_2(x)) + f_2(x)$
19. $f_1(y + f_2(x)) + f_2(x) + x$
20. $f_1(y + f_2(x + y)) + f_2(x + y)$
21. $f_1(y + f_2(x + y)) + f_2(x + y) + x$

22. $f_1(y + f_2(x + y)) + f_2(x + y) + x + y$
23. $f_1(x + y + f_2(x)) + f_2(x)$
24. $f_1(x + y + f_2(x)) + f_2(x) + x$
25. $f_1(x + y + f_2(y)) + f_2(y)$
26. $f_1(x + y + f_2(y)) + f_2(y) + y$

The remaining 12 functions have the form $f(x, y) = f_1(a + f_2(b)) + c$ where a, b, c are as above:

27. $f_1(x + f_2(y)) + x$
28. $f_1(x + f_2(y)) + x + y$
29. $f_1(x + f_2(x + y)) + x$
30. $f_1(x + f_2(x + y)) + y$
31. $f_1(y + f_2(x)) + y$
32. $f_1(y + f_2(x)) + x + y$
33. $f_1(y + f_2(x + y)) + x$
34. $f_1(y + f_2(x + y)) + y$
35. $f_1(x + y + f_2(x)) + y$
36. $f_1(x + y + f_2(y)) + x$
37. $f_1(x + y + f_2(y)) + x + y$
38. $f_1(x + y + f_2(x)) + x + y$

We also considered compression functions which are constructed using a predefined constant, as discussed in Section 2.1. Without any additional structural restrictions, this results in an additional 532 possible PrA compression functions. we model a constant in Linicrypt as an extra input v which is also exposed as an output. To do this we add this value to the base vector as the last input variable, and also, in order to make the value a constant, we allocate an output vector for it. Hence, if we

assume the inputs are x and y the base vector would be $(x, y, c, f1, f2)$ and the added output vector would be $(0, 0, 1, 0, 0)$. Listed below are 9 PrA compression functions that utilize a fixed constant c.

1. $f_1(x) + f_2(y) + c$
2. $f_1(x) + f_2(x + y) + c$
3. $f_1(y) + f_2(x + y) + c$
4. $f_1(x + f_2(y)) + f_2(y) + c$
5. $f_1(x + f_2(x + y)) + f_2(x + y) + c$
6. $f_1(y + f_2(x)) + f_2(x) + c$
7. $f_1(y + f_2(x + y)) + f_2(x + y) + c$
8. $f_1(x + y + f_2(y)) + f_2(y) + c$
9. $f_1(x + y + f_2(x)) + f_2(x) + c$

Chapter 3

Linicrypt in the Ideal Cipher Model

This chapter extends the approach introduced by [25] to characterize the collision-resistance properties of compression functions constructed in the ideal cipher model. Moreover, it demonstrates the potential for automated validation and generation.

In [25] the *Linicrypt* formalism [8] is applied to give a characterization of collision- and 2nd-preimage resistance of hash functions constructed via straight-line algebraic programs with access to a random oracle. In accordance with the insights presented by [25], the characterization provided in their work is expanded to the ideal cipher model in this chapter. As the ideal cipher model is a standard setting for the construction of cryptographic hash functions, in particular modeling block-cipher-based construction of compression functions, this is a natural and relevant extension of the previous work. The original Linicrypt framework [8] primarily facilitates calls to a random oracle. To enable calls to an ideal cipher within Linicrypt, it is necessary to model encryption and decryption queries. Notably, in contrast to random oracle calls, ideal cipher calls involve two inputs, specifically the key and the message, and a unique nonce as part of the input, ensuring the independence of these calls. In this chapter, we will address these challenges and provide effective solutions to incorporate ideal cipher calls seamlessly into the Linicrypt framework.

Additionally, it's worth mentioning that numerous real-world compression functions such as Davies–Meyer, Matyas–Meyer–Oseas, Miyaguchi–Preneel (single-block-length compression functions) [4] and MDC-2, MDC-4, Hirose (double-block-length compression functions) [30][19] are constructed using block ciphers. Therefore, to

effectively model these functions in Linicrypt, a transition to the ideal cipher mode becomes necessary.

Furthermore, the 64 PGV compression functions are also block-cipher-based and they fit into the extended Linicrypt framework proposed in this chapter. As we mentioned before Black, Rogaway, and Shrimpton [4] prove that of the 64 PGV compression functions, 12 of them, referred to as *group-1*, are collision-resistant and preimage resistant up to the birthday bound, and 8 of them, which are called *group-2* are only collision-resistant after some iteration. The proofs in [4] are given on a per-function basis, with canonical examples and an indication of how these could be generalized to any function in the corresponding group. In subsequent work, [5] characterized group-1 and group-2 PGV compression functions via a more general approach which considers fundamental combinatorial properties of the definitions, based on *pre-* and *post-processing functions.* A related work by Stam [36] presents these properties in an algebraic setting, partly anticipating the approach presented here, but with a much less general approach. In Section 3.3, the PGV compression functions are modeled in Linicrypt, revealing that the properties proposed in [5] can indeed be derived as a specific instance within our general characterization 3.2.5.

Contributions The main contributions of this chapter are summarized as follows:

1. A formulation in the ideal cipher model of the notion of *collision structure*, introduced in [25] for the random oracle model (Definition 3.2.5).

2. A characterization showing a Linicrypt program is collision-resistant (and 2nd-preimage-resistant) if and only if it does not have a collision structure (Theorem 3.2.8).

3. An efficient algorithm for finding collision structures (Algorithm 1).

4. In the rate-1 setting, giving an alternate proof the collision-resistance of *group-1* PGV compression functions as originally established in [4] using our characterization (Theorem 3.3.10.)

While our approach largely follows that of [25], the extension to the ideal cipher model is non-trivial, and the application to the PGV functions presents an interesting case where the Linicrypt approach sheds new light on existing approaches to hash function security.

3.1 Extending the Linicrypt Framework for Ideal Cipher Calls

In moving from the programs in the random oracle model considered in previous work on Linicrypt [8, 25, 20] to programs in the ideal cipher model, we now have calls to an ideal cipher $E(t,.,.)$ on a pair of values which are either program inputs or previously assigned variables. These inputs to E correspond to the key and input of an encryption query. Thus, supposing $\mathcal{P}$ has n oracle calls, for $i \in [n]$, each call is of the form $\boldsymbol{a}_i \cdot \boldsymbol{v}_{base} = E(t_i, \boldsymbol{q}_{k_i} \cdot \boldsymbol{v}_{base}, \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}_{base})$ and can be represented by an *oracle constraint* $c = (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a})$ where $\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a} \in \mathbb{F}^{k+n}$. To simplify the presentation we define $K_i := \boldsymbol{q}_{K_i} \cdot \boldsymbol{v}_{base}$, $X_i := \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}_{base}$ and $Y_i := \boldsymbol{a}_i \cdot \boldsymbol{v}_{base}$. Letting $\mathcal{C}$ denote the set of oracle constraints and $\boldsymbol{M}$ the output matrix, we again have an algebraic representation $(\boldsymbol{M}, \mathcal{C})$. We also assume the programs make only encryption queries while the adversary makes both encryption and decryption queries. In this context, nonces denoted as t are employed to ensure the independence of ideal cipher calls. Even when adversaries use the same key, the utilization of nonces guarantees that each call remains independent, preventing patterns or correlations in the encrypted outputs. The following is a simple example of such a program:

$$\begin{aligned} &\underline{\mathcal{P}^E(v_1, v_2):} \\ &\quad v_3 := E(t, v_1, v_2) \\ &\quad return\ v_3 + v_2 \end{aligned}$$

$$\boldsymbol{M} = (0,1,1),\ \mathcal{C} = (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a}),\ \boldsymbol{v}_{base} = (v_1, v_2, v_3)^\top$$

$$\boldsymbol{q}_K = (1,0,0) \quad \boldsymbol{q}_X = (0,1,0) \quad \boldsymbol{a} = (0,0,1)$$

Further examples are given in Section 3.5.

3.2 Characterizing Collision Resistance

The definitions of collision resistance and second-preimage resistance given above for the random oracle model, have a straightforward generalization to the ideal cipher. Recall that in this setting, the adversary has access to both E and E^{-1}. Let q_E denote the number of oracle queries the adversary makes to E and q_D the number of queries it makes to E^{-1}.

Definition 3.2.1. *Program $\mathcal{P}$ is* (q,ϵ)-collision-resistant *if any oracle adversary $\mathcal{A}$ making at most $q = q_E + q_D$ queries has probability of success at most ϵ in the following game:*

$$(\boldsymbol{x}, \boldsymbol{x}') \leftarrow \mathcal{A}^{E,E^{-1}}(\lambda);\ \textit{return}\ (\boldsymbol{x} \neq \boldsymbol{x}')\ \textit{and}\ \mathcal{P}^E(\boldsymbol{x}) = \mathcal{P}^E(\boldsymbol{x}') \tag{3.1}$$

Definition 3.2.2. *Program $\mathcal{P}$ is* (q,ϵ)-2nd-preimage resistant *if any oracle adversary $\mathcal{A}$ making at most $q = q_E + q_D$ queries has probability of success at most ϵ in the following game:*

$$\boldsymbol{x} \leftarrow \mathbb{F}^k; \boldsymbol{x}' \leftarrow \mathcal{A}^{E,E^{-1}}(\boldsymbol{x}, \lambda);\ \textit{return}\ (\boldsymbol{x} \neq \boldsymbol{x}')\ \textit{and}\ \mathcal{P}^E(\boldsymbol{x}) = \mathcal{P}^E(\boldsymbol{x}') \tag{3.2}$$

In this section, an algebraic condition is provided to characterize collision resistance and second-preimage resistance for Linicrypt programs in the ideal cipher model (Definition 3.2.5). Before giving the definition we note some programs fail trivially to be collision-resistant because two different inputs produce exactly the same queries to the oracle. This is formalized in the following:

Definition 3.2.3. *Program $\mathcal{P} = (M, \mathcal{C})$ is* degenerate *if*

$$\begin{aligned}\mathsf{span}(\{\mathbf{e}_1, \ldots, \mathbf{e}_{k+n}\}) \not\subseteq \mathsf{span}(&\{\boldsymbol{q}_K \mid (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a}) \in \mathcal{C}\} \ \cup\ \{\boldsymbol{q}_X \mid (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a}) \in \mathcal{C}\} \\ &\cup\ \{\boldsymbol{a} \mid (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a}) \in \mathcal{C}\} \ \cup\ \mathsf{rows}(\boldsymbol{M}))\end{aligned}$$

Lemma 3.2.4. *If $\mathcal{P}$ is degenerate then 2nd-preimages can be found with probability 1.*

Proof. Assume the adversary $\mathcal{A}$ is given a preimage $\boldsymbol{x}$, and it determines the base vector $\boldsymbol{v}_{base}$ in the execution of $\mathcal{P}(\boldsymbol{x})$. We define the matrix $\boldsymbol{P} = \begin{bmatrix} \boldsymbol{Q}_K \\ \hdashline \boldsymbol{Q}_X \\ \hdashline \boldsymbol{A} \\ \hdashline \boldsymbol{M} \end{bmatrix}$ where $\boldsymbol{Q}_K, \boldsymbol{Q}_X, \boldsymbol{A}$ are matrices whose rows correspond to the components of the elements of $\mathcal{C}$ (ordered arbitrarily.) If the adversary can determine a 2nd-preimage $\boldsymbol{x}' \neq \boldsymbol{x}$ where $\boldsymbol{P}\boldsymbol{v}_{base} = \boldsymbol{P}\boldsymbol{v}'_{base}$ where $\boldsymbol{v}'_{base}$ is the base vector in the calculation of $\mathcal{P}(\boldsymbol{x}')$ then $\mathcal{P}(\boldsymbol{x}) = \mathcal{P}(\boldsymbol{x}')$ and $\mathcal{A}$ wins. Since program $\mathcal{P}$ is degenerate, the rows of $\boldsymbol{P}$ cannot span all $k+n$ basis vectors which means $\mathrm{rank}(\boldsymbol{P}) < k+n$, and thus, for some $\boldsymbol{v} \neq \boldsymbol{0}$, $\boldsymbol{P}\boldsymbol{v} = \boldsymbol{0}$. The adversary can solve for this $\boldsymbol{v}$ and set $\boldsymbol{v}'_{base} = \boldsymbol{v}_{base} + \boldsymbol{v}$. Then $\boldsymbol{P}(\boldsymbol{v}'_{base} - \boldsymbol{v}_{base}) = \boldsymbol{0}$, so $\boldsymbol{v}'_{base} \neq \boldsymbol{v}_{base}$ and $\boldsymbol{P}\boldsymbol{v}'_{base} = \boldsymbol{P}\boldsymbol{v}_{base}$. In particular, this allows $\mathcal{A}$ to compute $\boldsymbol{x}' \neq \boldsymbol{x}$ such that $\mathcal{P}(\boldsymbol{x}') = \mathcal{P}(\boldsymbol{x})$.

□

The following example gives the idea behind the definition of collision structure for Linicrypt programs in the ideal cipher model, and how a collision attack can be applied to a program. A more formal and precise algorithm to find a collision is given in the proof of Lemma 4.2.4.

$$\underline{\mathcal{P}^E(v_1, v_2):}$$
$$v_3 := E(t_1, v_1, v_2)$$
$$v_4 = E(t_2, v_1, v_3)$$
$$return\ v_4 + v_1$$

$$\boldsymbol{q}_{K_1} = (1,0,0,0) \quad \boldsymbol{q}_{X_1} = (0,1,0,0) \quad \boldsymbol{a}_1 = (0,0,1,0)$$
$$\boldsymbol{q}_{K_2} = (1,0,0,0) \quad \boldsymbol{q}_{X_2} = (0,0,1,0) \quad \boldsymbol{a}_2 = (0,0,0,1)$$
$$\boldsymbol{m} = (1,0,0,1)$$

This program is not collision resistant and the adversary can find a collision as follows

- The adversary $\mathcal{A}$ picks an arbitrary input $\boldsymbol{x} = (x_1, x_2) \in \mathbb{F}^2$ and runs the program on $\boldsymbol{x}$ and gets the output l.
- $\mathcal{A}$ picks arbitrary $x_1' \neq x_1 \in \mathbb{F}$ and makes the query $v_3' = E^{-1}(t_1, x_1', l + x_1')$.
- $\mathcal{A}$ makes the query $x_2' = E^{-1}(t_2, x_1', v_3')$ to find x_2'.
- $\mathcal{A}$ returns $\boldsymbol{x}' = (x_1', x_2')$.

This attack was possible because of the following,

- x_1' was independent of the output in other words $\boldsymbol{q}_{K_1} \neq \boldsymbol{m}$.
- In step two after fixing x_1' and the output l the value of v_3' was not fixed which in algebraic representation means $\boldsymbol{q}_{X_1} \notin \mathsf{span}(\boldsymbol{q}_{K_1}, \boldsymbol{m})$ so the adversary could determine this value by making a decryption query.
- In the third step because $\boldsymbol{q}_{X_2} \notin \mathsf{span}(\boldsymbol{q}_{K_1}, \boldsymbol{q}_{K_2}, \boldsymbol{q}_{X_1}, \boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{m})$ the value of x_2' is not fixed so via a decryption query the adversary finds a compatible value for x_2'.

The following definition gives a syntactic condition on programs that will be used to characterize collision resistance. Intuitively, for a program to have a collision $\boldsymbol{x} \neq \boldsymbol{x}'$, there must first be a query for which the adversary can pick an arbitrary input. This means at least two of $K = \boldsymbol{v}_{base} \cdot \boldsymbol{q}_K, X = \boldsymbol{v}_{base} \cdot \boldsymbol{q}_X$ and $Y = \boldsymbol{v}_{base} \cdot \boldsymbol{a}$ need to be independent of all other fixed values. Secondly, to get the same output value on this different input $\boldsymbol{x}'$, the results of the remaining queries must be independent of other fixed values which implies one of Y or X must be independent of the previous queries and output values. This leads to the following definition:

Definition 3.2.5. *Let $\mathcal{P} = (\boldsymbol{M}, \mathcal{C})$ be a Linicrypt program. A collision structure for $\mathcal{P}$ is a tuple $(i^*, c_1, \ldots, c_n)$ where $c_1, \cdots, c_n$ is an ordering of $\mathcal{C}$ and $i^* \in [1, n]$, such that for $i = i^*$ at least two of the following conditions are true, and for all $i > i^*$ at least one of (C2) or (C3) is true.*

(C1) $\boldsymbol{q}_{K_i} \notin \mathsf{span}(\{\boldsymbol{q}_{K_1}, \ldots, \boldsymbol{q}_{K_{i-1}}\}, \{\boldsymbol{q}_{X_1}, \ldots \boldsymbol{q}_{X_{i-1}}\}, \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1}\}, \mathsf{rows}(\boldsymbol{M}))$

(C2) $\boldsymbol{q}_{X_i} \notin \mathsf{span}(\{\boldsymbol{q}_{K_1}, \ldots, \boldsymbol{q}_{K_i}\}, \{\boldsymbol{q}_{X_1}, \ldots \boldsymbol{q}_{X_{i-1}}\}, \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_i\}, \mathsf{rows}(\boldsymbol{M}))$

(C3) $\boldsymbol{a}_i \notin \mathsf{span}(\{\boldsymbol{q}_{K_1}, \ldots, \boldsymbol{q}_{K_i}\}, \{\boldsymbol{q}_{X_1}, \ldots \boldsymbol{q}_{X_i}\}, \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1}\}, \mathsf{rows}(\boldsymbol{M}))$

Lemma 3.2.6. *If a Linicrypt program $\mathcal{P}$ with n constraints has a collision structure $(i^*, c_1, \ldots, c_n)$ then there exists a collision adversary $\mathcal{A}$ with access to E and E^{-1} which given an input $\boldsymbol{x}$ makes at most $2n$ queries and returns $\boldsymbol{x}' \neq \boldsymbol{x}$ such that $\mathcal{P}^E(\boldsymbol{x}') = \mathcal{P}^E(\boldsymbol{x})$, and so has success probability of 1 in Game 4.3.*

Proof. The adversary $\mathcal{A}$ first determines a setting of the base variables $\boldsymbol{v}$ by running $\mathcal{P}^E(\boldsymbol{x})$, and creates linear constraints on unknowns $\boldsymbol{v}'$ as follows:

- add constraint $\boldsymbol{M}\boldsymbol{v}' = \boldsymbol{M}\boldsymbol{v}$
- for $i < i^*$, add constraints $\boldsymbol{q}_{K_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{K_i} \cdot \boldsymbol{v}$, $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}$ and $\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v}$
- For $i \geq i^*$,
  - if (C1) holds, choose $K_i' \in \mathbb{F}$ so that $K_i' \neq \boldsymbol{q}_{K_i} \cdot \boldsymbol{v}$ and add the constraint $\boldsymbol{q}_{K_i} \cdot \boldsymbol{v}' = K_i'$
  - if (C2) holds, set $X_i' := E^{-1}(t_i, \boldsymbol{q}_{K_i} \cdot \boldsymbol{v}', \boldsymbol{a}_i \cdot \boldsymbol{v}')$ and add the constraint $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = X_i'$
  - if (C3) holds, set $Y_i' := E(t_i, \boldsymbol{q}_{k_i} \cdot \boldsymbol{v}', \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}')$ and add the constraint $\boldsymbol{a}_i \cdot \boldsymbol{v}' = Y_i'$

– if (C2) and (C3) both hold, choose $X'_i \in \mathbb{F}$ such that $X'_i \neq \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}$, set $Y_i := E(t_i, \boldsymbol{q}_{k_i} \cdot \boldsymbol{v}', X'_i)$ and add the constraints $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = X'_i$ and $\boldsymbol{a}_i \cdot \boldsymbol{v}' = Y'_i$

We claim that the constraints have a unique solution $\boldsymbol{v}' \neq \boldsymbol{v}$ such that if $x'_i = \mathbf{e}_i \cdot \boldsymbol{v}'$, $1 \leq i \leq k$, then $\boldsymbol{x}' \neq \boldsymbol{x}$ and $\mathcal{P}^E(\boldsymbol{x}') = \mathcal{P}^E(\boldsymbol{x})$.

To see that $\boldsymbol{v}' \neq \boldsymbol{v}$, note that for $i = i^*$, either (C1) holds, or both (C2) and (C3) hold. The choice of K'_{i^*} in the first case and X'_{i^*} in the second, ensure $\boldsymbol{v}' \neq \boldsymbol{v}$.

The constraints that are added for the output matrix and for $i < i^*$ are consistent, as they already have a solution, namely $\boldsymbol{v}$. For $i \geq i^*$, a new constraint is added only in the case that the corresponding $\boldsymbol{q}_{K_i}, \boldsymbol{q}_{X_i}$ or $\boldsymbol{a}_i$ is independent of the vectors added in previous constraints, and so consistency is maintained as constraints are added. Once all constraints are added, nondegeneracy ensures that $\boldsymbol{v}'$ is unique.

Finally, $\boldsymbol{v}'$ is consistent with the values returned by E and E^{-1}. This means that $\boldsymbol{v}'$ corresponds to the setting of base variables resulting from evaluating $\mathcal{P}^E(\boldsymbol{x}')$, so from $\boldsymbol{v}' \neq \boldsymbol{v}$ we conclude $\boldsymbol{x}' \neq \boldsymbol{x}$, and from the $\boldsymbol{M}$ constraint, $\mathcal{P}^E(\boldsymbol{x}') = \mathcal{P}^E(\boldsymbol{x})$.

□

Lemma 3.2.7. *Let $\mathcal{P}$ be a Linicrypt program with n constraints. If there is an adversary $\mathcal{A}$ for $\mathcal{P}$ making at most N oracle queries with success probability $> (N/n)^{2n}/|\mathbb{F}|$ in the collision-resistance game (Game 4.1) or success probability $> (N/n)^n/|\mathbb{F}|$ in the 2nd-preimage game (Game 4.3) then the $\mathcal{P}$ is either degenerate or has a collision structure $(i^*, c_1, \ldots, c_n)$.*

Proof. We may assume the following without loss of generality:

1. $\mathcal{A}$ does not repeat a query or make the inverse of a query it has already made. This can be achieved by recording queries as they are made.

2. For queries made in the execution of $\mathcal{P}(\boldsymbol{x})$ and $\mathcal{P}(\boldsymbol{x}')$, $\mathcal{A}$ makes either the query or its corresponding inverse query before returning. For Game 4.1, this is achieved by having $\mathcal{A}$ run $\mathcal{P}(\boldsymbol{x})$ and $\mathcal{P}(\boldsymbol{x}')$ before returning and making the corresponding queries subject to restriction (1). For Game 4.3 this is achieved by having $\mathcal{A}$ initially make all the queries that result from running $\mathcal{P}(\boldsymbol{x})$ and also running $\mathcal{P}(\boldsymbol{x}')$ before returning, and making any corresponding query, subject to restriction (1), before returning.

3. $\mathcal{A}$ actually returns $\boldsymbol{v}, \boldsymbol{v}'$ which are the settings of base variables determined by the execution of $\mathcal{P}(\boldsymbol{x})$ and $\mathcal{P}(\boldsymbol{x}')$, respectively.

If the number of adversary's queries with nonce t_i is N_i, then the assumptions imply that for an oracle constraint $c = (t_i, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a})$ occurring in $\mathcal{P}$, $\mathcal{A}$ determines the value of triples $(\boldsymbol{q}_K \cdot \boldsymbol{v}, \boldsymbol{q}_X \cdot \boldsymbol{v}, \boldsymbol{a} \cdot \boldsymbol{v})$ through exactly one of its N_i queries which is either a E-query or E^{-1}-query. Based on this fact, we define two mappings $T, T' : \mathcal{C} \to [N]$ where $\mathcal{C}$ is the set of constraints in $\mathcal{P}$ and $T(c)$th and $T'(c)$th adversary queries correspond to constraint $c = (t_i, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a})$ in the computation of $\mathcal{P}(\boldsymbol{x})$ and $\mathcal{P}(\boldsymbol{x}')$, and determine the triple $(\boldsymbol{q}_K \cdot \boldsymbol{v}, \boldsymbol{q}_X \cdot \boldsymbol{v}, \boldsymbol{a} \cdot \boldsymbol{v})$ and $(\boldsymbol{q}_K \cdot \boldsymbol{v}', \boldsymbol{q}_X \cdot \boldsymbol{v}', \boldsymbol{a} \cdot \boldsymbol{v}')$ respectively. In Game 4.1, $T(c_i)$ and $T'(c_i)$ are each mapped to one of N_i queries made by $\mathcal{A}$, so the number of possible mappings (T, T') is $\prod_{i=1}^{n} N_i^2$. In Game 4.3, Assumption 2 implies that T is fixed, so the number of possible mappings (T, T') is (T, T') is $\prod_{i=1}^{n} N_i$. Considering the product is maximized when $N_i = N/n$, we get the upper bound $(N/n)^{2n}$ on the maximum number of mapping (T, T') in Game 4.1 and $(N/n)^n$ in Game 4.3. When using the pigeonhole principle and $\mathcal{A}$'s assumed advantage in each game, there is a specific mapping (T, T') for which $\mathcal{A}$'s advantage when using this mapping is at least $1/|\mathbb{F}|$. We will assume that the adversary is using this mapping — for any other mapping, it returns $\perp$ as its last action.

Using the same terminology as [25], a query $c \in \mathcal{C}$ is *convergent* if $T(c) = T'(c)$, and *divergent* otherwise. Because $\boldsymbol{x} \neq \boldsymbol{x}'$ is a collision and $\mathcal{P}$ is nondegenerate there is at least one divergent constraint. Define $\texttt{finish}(c) = \max\{T(c), T'(c)\}$. Since each constraint has a distinct nonce, two different program constraints can not be mapped to the same adversary query, thus, function `finish` is injective. Note a non-injective function `finish` makes some attacks possible that can not be covered by a collision structure. An example of such attacks can be found in Appendix A.1.3. Arrange the oracle constraints in C as $(c_1, \ldots, c_n)$, with the convergent queries taking precedence, followed by the divergent queries sorted in ascending order based on their `finish` and letting i^* denote the index of the first divergent constraint, we claim that $(i^*, c_1, \ldots, c_n)$ is a collision structure for $\mathcal{P}$.

For $i < i^*$, since each c_i is convergent we have $\boldsymbol{q}_{K_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{K_i} \cdot \boldsymbol{v}$, $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}$ and $\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v}$ and because $\mathcal{P}(\boldsymbol{x}) = \mathcal{P}(\boldsymbol{x}')$ we have $\boldsymbol{M}\boldsymbol{v}' = \boldsymbol{M}\boldsymbol{v}$.

For $i = i^*$ the query c_i is divergent thus at least one of the following inequalities holds $\boldsymbol{q}_{K_i} \cdot \boldsymbol{v}' \neq \boldsymbol{q}_{K_i} \cdot \boldsymbol{v}$, $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' \neq \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}$ or $\boldsymbol{a}_i \cdot \boldsymbol{v}' \neq \boldsymbol{a}_i \cdot \boldsymbol{v}$. If $\boldsymbol{a}_i \cdot \boldsymbol{v}' \neq \boldsymbol{a}_i \cdot \boldsymbol{v}$ or $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' \neq \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}$ then at least one of the other inequalities hold since the ideal cipher is a permutation when the key is fixed.

This gives five possible cases. Without loss of generality, in all cases, we assume that $T(c_i) < T'(c_i)$.

1. $K_i = K'_i$ and $X_i \neq X'_i$ and $Y_i \neq Y'_i$.

   In this case, we prove both conditions (C2) and (C3) hold. By way of contradiction assume (C3) does not hold, say

   $$\boldsymbol{a}_i = \sum_{j \leq i} \alpha_j \boldsymbol{q}_{K_j} + \sum_{j \leq i} \beta_j \boldsymbol{q}_{X_j} + \sum_{j < i} \gamma_j \boldsymbol{a}_j + \boldsymbol{\delta M}, \tag{3.3}$$

   for some $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\delta}$. After multiplying both side of the equation by $(\boldsymbol{v}' - \boldsymbol{v})$ and considering that all the queries before i^* are convergent, $\boldsymbol{M}(\boldsymbol{v}' - \boldsymbol{v}) = 0$, and $K_i = K'_i$ we have

   $$\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v} + \beta_i \boldsymbol{q}_{X_i} \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

   Also because $Y_i \neq Y'_i$ and $X_i \neq X'_i$ we know $\beta_i \neq 0$. If $T'(c_i)$ is an encryption query then the right-hand side of the equation is a fixed value but the left-hand side is a query result, so the advantage of the adversary is $\leq 1/|\mathbb{F}|$ contrary to assumption. If $T'(c_i)$ is a decryption query then isolating $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}'$ gives us

   $$\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{X_i} \cdot \boldsymbol{v} + \frac{1}{\beta_i} \boldsymbol{a}_i \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

   and again the right-hand side of the equation is determined while the left-hand side is a random value, again giving a contradiction. The proof for condition (C2) is similar.

2. $K_i \neq K'_i$ and $X_i = X'_i$ and $Y_i \neq Y'_i$.

   Because $K_i \neq K'_i$, condition (C1) holds. We want to show $T'(c_i)$ is an encryption query and (C3) holds. If $T'(c_i)$ is not an encryption then X_i is fixed and X'_i random so $X_i = X'_i$ holds with probability $\leq 1/|\mathbb{F}|$. If condition (C3) does not hold then 4.6 holds. Multiplying the equation to $(\boldsymbol{v}' - \boldsymbol{v})$ and applying $X_i = X'_i$ and $\boldsymbol{M}(\boldsymbol{v} - \boldsymbol{v}') = 0$ gives

   $$\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v} + \alpha_i \boldsymbol{q}_{K_i} \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

   The left-hand side of the above equation is a random value and the right-hand side is a fixed value, so the adversary advantage again is $\leq 1/|\mathbb{F}|$.

3. $K_i \neq K'_i$ and $X_i \neq X'_i$ and $Y_i = Y'_i$.

Because $K_i \neq K_i'$, condition (C1) holds. We want to show $T'(c_i)$ has to be a decryption query and (C2) holds. If it is not a decryption query then the probability of a random value Y_i' being equal to the fixed value Y_i would be $1/|\mathbb{F}|$, which contradicts the assumption, so we assume $T'(c_i)$ is a decryption query. If condition (C2) does not hold then we have

$$\boldsymbol{q}_{X_i} = \sum_{j \leq i} \pi_j \cdot \boldsymbol{q}_{K_j} + \sum_{j<i} \rho_j \cdot \boldsymbol{q}_{X_j} + \sum_{j \leq i} \varsigma_j \cdot \boldsymbol{a}_j + \boldsymbol{\tau M}$$

Multiplying the equation to $(\boldsymbol{v}' - \boldsymbol{v})$ and applying $Y_i = Y_i'$ and $\boldsymbol{M}(\boldsymbol{v} - \boldsymbol{v}') = 0$ and canceling convergent queries, gives

$$\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{X_i} \cdot \boldsymbol{v} + \pi_i \boldsymbol{q}_{K_i} \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

The left-hand side of the above equation is a random value and the right-hand side is a fixed value, so the adversary advantage again is at most $1/|\mathbb{F}|$ which is a contradiction.

4. $K_i \neq K_i'$ and $X_i \neq X_i'$ and $Y_i \neq Y_i'$.

   Because $K_i \neq K_i'$, condition (C1) holds. We show if $T'(c_i)$ is an encryption query then (C3) holds and if it is a decryption query then (C2) holds. In the first case, assume for contradiction that (C3) does not hold, implying Equation 4.6. Applying the assumption $\boldsymbol{M}(\boldsymbol{v} - \boldsymbol{v}') = 0$ and canceling all the queries before i^* gives,

$$\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v} + \alpha_i \boldsymbol{q}_{K_i} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \beta_i \boldsymbol{q}_{X_i} \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

   Here when the adversary is making query $T'(c_i)$, all the values on the right-hand side of the equation are fixed and so the adversary's advantage is $\leq 1/|\mathbb{F}|$, contrary to assumption. The case that $T'(c_i)$ is a decryption query and (C2) holds is similar.

5. $K_i \neq K_i'$ and $X_i = X_i'$ and $Y_i = Y_i'$.

   The probability of this case occurring is $\leq 1/|\mathbb{F}|$.

For $i > i^*$ by way of contradiction, assume (C2) and (C3) both fail:

$$\boldsymbol{q}_{X_i} = \sum_{j \le i} \pi_j \cdot \boldsymbol{q}_{K_j} + \sum_{j < i} \rho_j \cdot \boldsymbol{q}_{X_j} + \sum_{j \le i} \varsigma_j \cdot \boldsymbol{a}_j + \boldsymbol{\tau M}$$

$$\boldsymbol{a}_i = \sum_{j \le i} \alpha_j \cdot \boldsymbol{q}_{K_j} + \sum_{j \le i} \beta_j \cdot \boldsymbol{q}_{X_j} + \sum_{j < i} \gamma_j \cdot \boldsymbol{a}_j + \boldsymbol{\delta M}$$

Multiplying both sides by $(\boldsymbol{v}' - \boldsymbol{v})$ and canceling the terms having index less than i^* and noting $\boldsymbol{M}(\boldsymbol{v} - \boldsymbol{v}') = 0$ we get,

$$\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{X_i} \cdot \boldsymbol{v} + \sum_{i^* \le j \le i} \pi_j \boldsymbol{q}_{K_j} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{i^* \le j < i} \rho_j \boldsymbol{q}_{X_j} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{i^* \le j \le i} \varsigma_j \boldsymbol{a}_j \cdot (\boldsymbol{v}' - \boldsymbol{v}) \quad (3.4)$$

$$\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v} + \sum_{i^* \le j \le i} \alpha_j \boldsymbol{q}_{K_j} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{i^* \le j \le i} \beta_j \boldsymbol{q}_{X_j} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{i^* \le j < i} \gamma_j \boldsymbol{a}_j \cdot (\boldsymbol{v}' - \boldsymbol{v}) \quad (3.5)$$

Now, if the adversary is making query $T'(c_i)$ as an encryption query then in Equation 4.8 all the values on the right-hand side of the equation are fixed and the left-hand side is random so the adversary advantage is at most $1/|\mathbb{F}|$, and if it is a decryption query, Equation 4.7 will give the same contradiction. So at least one of the conditions (C2) or (C3) has to hold.

□

Combining the Lemmas of this section, we obtain:

Theorem 3.2.8. *(Main Theorem) Suppose $\mathcal{P}$ is a nondegenerate Linicrypt program in the ideal cipher model over $\mathbb{F}_\lambda$, with n constraints. For sufficiently large λ the following are equivalent*

- *$\mathcal{P}$ is $(N, (N/n)^{2n}/|\mathbb{F}|)$-collision resistant*
- *$\mathcal{P}$ is $(N, (N/n)^{n}/|\mathbb{F}|)$-2nd preimage resistant*
- *$\mathcal{P}$ is $(2n, 1)$-2nd preimage resistant*
- *$\mathcal{P}$ does not have a collision structure*

3.2.1 Efficiently Finding Collision Structures

An immediate benefit of the characterization given in the proceeding section is provided by Algorithm 1, which gives an efficient procedure for deciding whether a program has a collision structure. The algorithm splits the constraints into two stacks using two loops. In the beginning, all the constraints $\mathcal{C}$ are in the LEFT stack and the first loop runs until all the constraints that satisfy at least one of (C2) or, (C3) are assigned to the RIGHT stack. In the second loop, those constraints that don't satisfy at least two of (C1), (C2), or (C3) will be pushed back to the LEFT stack.

Each loop makes at most n iterations, where each iteration involves several span computations. This gives a total running time of $O(n^{\omega+1})$, where ω is the exponent in the complexity of matrix multiplication.

Lemma 3.2.9. *Algorithm* ***FindColStruct*** $\mathcal{P}$ *returns a collision structure for* $\mathcal{P}$ *iff one exists.*

Proof. First, we prove if the algorithm returns $(i^*, c_1, \ldots, c_n)$, this is a collision structure. Note that after the second loop ends,

$$V = \{\boldsymbol{q}_{K_1}, \ldots, \boldsymbol{q}_{K_{i^*-1}}\} \cup \{\boldsymbol{q}_{X_1}, \ldots, \boldsymbol{q}_{X_{i^*-1}}\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i^*-1}\} \cup \mathsf{rows}(\boldsymbol{M}).$$

Also, the query c_{i^*} is still in RIGHT, so at least two of the conditions from Definition 3.2.5 hold, otherwise c_{i^*} would be sent back to LEFT.

For $i > i^*$, immediately before moving c_i from LEFT to RIGHT in the first loop, sLEFT still included $\{c_1, \ldots, c_{i-1}\}$ which means

$$V = \{\boldsymbol{q}_{K_1}, \ldots, \boldsymbol{q}_{K_i}\} \cup \{\boldsymbol{q}_{X_1}, \ldots, \boldsymbol{q}_{X_i}\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_i\} \cup \mathsf{rows}(\boldsymbol{M}),$$

and $\boldsymbol{q}_X \notin \mathsf{span}(V \setminus \{\boldsymbol{q}_X\})$ or $\boldsymbol{a} \notin \mathsf{span}(V \setminus \{\boldsymbol{a}\})$. Hence,

Algorithm 1 FindColStruct $\mathcal{P}(\boldsymbol{M}, \mathcal{C})$

LEFT $\coloneqq \mathcal{C}$
RIGHT $\coloneqq$ empty stack
$V \coloneqq \{\boldsymbol{q}_K | (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a}) \in \mathcal{C}\} \cup \{\boldsymbol{q}_X | (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a}) \in \mathcal{C}\} \cup \{\boldsymbol{a} | (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a}) \in \mathcal{C}\} \cup$ rows(M)

while $\exists (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a}) \in$ LEFT where $\boldsymbol{q}_X \notin \mathsf{span}(V \setminus \{\boldsymbol{q}_X\})$ or $\boldsymbol{a} \notin \mathsf{span}(V \setminus \{\boldsymbol{a}\})$ **do**
  remove $(t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a})$ from LEFT
  push $(t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a})$ to RIGHT
  reduce multiplicity of $\boldsymbol{q}_K, \boldsymbol{q}_X$ and $\boldsymbol{a}$ in V by 1
end while

while $\exists (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a}) \in$ RIGHT where
  $(t, \boldsymbol{q}_K \in \mathsf{span}(V) \wedge \boldsymbol{q}_X \in \mathsf{span}(V \cup \boldsymbol{q}_K \cup \boldsymbol{a}))$ or
  $(\boldsymbol{q}_K \in \mathsf{span}(V) \wedge \boldsymbol{a} \in \mathsf{span}(V \cup \boldsymbol{q}_K \cup \boldsymbol{q}_X))$ or
  $(\boldsymbol{q}_X \in \mathsf{span}(V \cup \boldsymbol{q}_K \cup \boldsymbol{a}) \wedge \boldsymbol{a} \in \mathsf{span}(V \cup \boldsymbol{q}_K \cup \boldsymbol{q}_X))$ **do**
  remove $(t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a})$ from RIGHT
  add $(t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a})$ to LEFT
  increase multiplicity of $\boldsymbol{q}_K, \boldsymbol{q}_X$ and $\boldsymbol{a}$ in V by 1
end while

if RIGHT is nonempty **then**
  $i^* \coloneqq |\text{LEFT}| + 1$
  let LEFT $= (c_1, \ldots, c_{i^*-1})$ where order doesn't matter
  let RIGHT $= (c_{i^*}, \ldots, c_n)$ in reverse order of insertion
  return $(i^*, c_1, \ldots, c_n)$
else return $\perp$
end if

$$\boldsymbol{q}_{X_i} \notin \mathsf{span}(\{\boldsymbol{q}_{K_1}, \ldots, \boldsymbol{q}_{K_i}\} \cup \{\boldsymbol{q}_{X_1}, \ldots, \boldsymbol{q}_{X_{i-1}}\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_i\} \cup \mathsf{rows}(\boldsymbol{M}))$$

or

$$\boldsymbol{a}_i \notin \mathsf{span}(\{\boldsymbol{q}_{K_1}, \ldots, \boldsymbol{q}_{K_i}\} \cup \{\boldsymbol{q}_{X_1}, \ldots, \boldsymbol{q}_{X_i}\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1}\} \cup \mathsf{rows}(\boldsymbol{M}))$$

satisfying one the conditions (C2), (C3) from Definition 3.2.5.

To prove the other direction, we prove if there is a collision structure for $\mathcal{P}$ then the second phase c_{i^*} is not sent back from RIGHT to LEFT, and so RIGHT $\neq \perp$. By contradiction suppose the algorithm adds c_{i^*} to LEFT. Denote by S the set of indices of constraints in LEFT immediately before c_{i^*} is added.

Then, for c_{i^*} to be sent back, at least 2 of the following conditions must hold

$$\boldsymbol{q}_{K_i^*} = \sum_{j\in S} \kappa_j \boldsymbol{q}_{K_j} + \sum_{j\in S} \lambda_j \boldsymbol{q}_{X_j} + \sum_{j\in S} \mu_j \boldsymbol{a}_j + \boldsymbol{\nu M} \tag{3.6}$$

$$\boldsymbol{q}_{X_i^*} = \sum_{j\in S\cup\{i^*\}} \pi_j \boldsymbol{q}_{K_j} + \sum_{j\in S} \rho_j \boldsymbol{q}_{X_j} + \sum_{j\in S\cup\{i^*\}} \varsigma_j \boldsymbol{a}_j + \boldsymbol{\tau M} \tag{3.7}$$

$$\boldsymbol{a}_{i^*} = \sum_{j\in S\cup\{i^*\}} \alpha_j \boldsymbol{q}_{K_j} + \sum_{j\in S\cup\{i^*\}} \beta_j \boldsymbol{q}_{X_j} + \sum_{j\in S} \gamma_j \boldsymbol{a}_j + \boldsymbol{\delta M} \tag{3.8}$$

Since, after the first loop $\{c_{i^*}, \ldots, c_n\} \subseteq$ RIGHT, and if any c_j for $j > i^*$ is sent back to LEFT it means at least 2 of $\{\boldsymbol{q}_{K_j}, \boldsymbol{q}_{X_j}, \boldsymbol{a}_j\}$ were already in the right-hand side of the above equations which is the span of vectors in LEFT and rows($\boldsymbol{M}$), so we can rewrite Equations 3.6, 3.7 and 3.8 as follows where the $S_1 \cup S_2 \cup S_3 = \{i^*, \ldots, n\}$ and each index appears at least 2 times in the unions of these three sets.

$$\boldsymbol{q}_{K_i^*} = \sum_{j\in S\setminus S_1} \kappa'_j \boldsymbol{q}_{K_j} + \sum_{j\in S\setminus S_2} \lambda'_j \boldsymbol{q}_{X_j} + \sum_{j\in S\setminus S_3} \mu'_j \boldsymbol{a}_j + \boldsymbol{\nu' M} \tag{3.9}$$

$$\boldsymbol{q}_{X_i^*} = \sum_{j\in S\setminus S_1} \pi'_j \boldsymbol{q}_{K_j} + \sum_{j\in S\setminus S_2} \rho'_j \boldsymbol{q}_{X_j} + \sum_{j\in S\cup\{i^*\}\setminus S_3} \varsigma'_j \boldsymbol{a}_j + \boldsymbol{\rho' M} \tag{3.10}$$

$$\boldsymbol{a}_{i^*} = \sum_{j\in S\setminus S_1} \alpha'_j \boldsymbol{q}_{K_j} + \sum_{j\in S\cup\{i^*\}\setminus S_2} \beta'_j \boldsymbol{q}_{X_j} + \sum_{j\in S\setminus S_3} \gamma'_j \boldsymbol{a}_j + \boldsymbol{\delta' M} \tag{3.11}$$

Assume 2 of these equations hold. If in these equations j_1, j_2, and j_3 be the maximum indices in $S \setminus S_1$, $S \setminus S_2$ and $S \setminus S_3$ then there are 2 cases,

If $j_1, j_2, j_3 \leq i^*$ then all the indices on the right-hand side are less than i^* and because at least two of the Equations 3.9, 3.10 and 3.11 are true, this contradicts with the condition for i^* in Definition 3.2.5.

If j_3, j_2, and j_1 are more than i^* then the coefficients with these indices in the above equations are nonzero. If the $\max\{j_1, j_2, j_3\} = j_1$ it means $\boldsymbol{q}_{K_{j_1}}$ was not in the span of V but both $\boldsymbol{q}_{X_{j_1}}$ and $\boldsymbol{a}_{j_1}$ where in the span of constraints with smaller indices

and $\mathsf{rows}(\boldsymbol{M})$ which is a contradiction otherwise they would not be in the RIGHT after the first loop. Thus, the $\max\{j_1, j_2, j_3\}$ is either j_2 or j_3. If j_3 is the max then $\boldsymbol{a}_{j_3}$ was not in the span of LEFT and $\mathsf{rows}(\boldsymbol{M})$ so the other two vectors $\boldsymbol{q}_{K_{j_3}}$ and $\boldsymbol{q}_{X_{j_3}}$ had to be in the span of LEFT and $\mathsf{rows}(\boldsymbol{M})$ and all the vectors in LEFT have smaller index than j_3 thus for c_{j_3} to be sent to RIGHT in the first loop, $\boldsymbol{a}_{j_3}$ had to be independent of previous constraints and the output (condition (C3)), but we can rewrite Equation 3.9 as follow,

$$\boldsymbol{a}_{j_3} = -\frac{1}{\gamma_{j_3}}\Big(\sum_{j\in S\setminus S_1}\alpha'_j\boldsymbol{q}_{K_j} - \boldsymbol{q}_{K_{i^*}} + \sum_{j\in S\setminus S_2}\beta'_j\boldsymbol{q}_{X_j} + \sum_{j\in S\setminus\{S_3\cup j_3\}}\gamma'_j\boldsymbol{a}_j + \boldsymbol{\delta}'\boldsymbol{M}\Big), \quad (3.12)$$

contradicting condition (C3) of Definition 3.2.5. The case for j_2 is similar.

□

3.3 Rate-1 Compression Functions

A compression function is *rate-1* if it uses one call to an underlying primitive, such as a random oracle or ideal cipher. In the latter case, we assume (without loss of generality) that there is one call to the ideal encryption function. The systematic study of such compression functions has a long history. Building on the initial work of [32], [4] gives a definition of 64 possible rate-1 compression functions mapping $D \times D \to D$, where $D = \mathbb{GF}(2^\lambda)$, that is definable using $\oplus$ and a constant value from D. Typically, these functions are referred to as *PGV compression functions*. Among its results, [4] identifies a subset of these functions, the *group-1* functions, and proves that these are exactly the PGV compression functions that are collision resistant.

The goal of this section is to give a characterization of the group-1 functions in Linicrypt. In particular, we will show that a PGV compression function is group-1 iff it does not have a collision structure. Thus the characterization of [4] may be viewed as a special case of our general characterization.

The results of [4] are revisited in [5], and proven via a more unified approach, building on the approach of [36]. This is based on the factorization of a compression function f into two component functions f' and f''. In particular, for a *message* m and *chaining value* h, $f(h, m) = f''(h, m, y)$, where $y = E_k(x)$ and $(k, x) = f'(h, m)$. In the case that f' is bijective, the function f^* is defined by $f^*(k, x, y) = f''(h, m, y)$ where $(h, m) = f^{-1}(k, x)$. Using f', f'', and f^*, the following properties of f are

defined:

P1 f' is bijective.

P2 $f''(h, m, \cdot)$ is bijective for all (h, m).

P3 $f^*(k, \cdot, y)$ is bijective for all (k, y).

In [4, 5], a stronger notion of collision-resistance for compression functions is used:

Definition 3.3.1 ([5] Definition 3)**.** *Program $\mathcal{P}$ is* (q, ϵ)-collision resistant *if for any $h_0 \in \mathbb{F}$, any oracle adversary $\mathcal{A}$ making at most $q = q_E + q_D$ queries has probability of success at most ϵ in the following game:*

$$(\boldsymbol{x}, \boldsymbol{x}') \leftarrow \mathcal{A}^{E,E^{-1}}(\lambda);\ \textit{return}\ (\boldsymbol{x} \neq \boldsymbol{x}')\ \textit{and}\ \mathcal{P}^E(\boldsymbol{x}) = \mathcal{P}^E(\boldsymbol{x}')\ \textit{or}\ \mathcal{P}^E(\boldsymbol{x}) = h_0 \quad (3.13)$$

Letting T1 denote the conjunction of P1, P2, and P3, we have

Lemma 3.3.2 ([5] Lemma 3)**.** *Suppose $f : \mathbb{GF}(2^\lambda) \times \mathbb{GF}(2^\lambda) \to \mathbb{GF}(2^\lambda)$ is a PGV compression function satisfying T1. Then for any $h_0 \in \mathbb{GF}(2^\lambda)$ the advantage of any adversary $\mathcal{A}$ making q queries in Game 3.13 is at most $q(q+1)/2^\lambda$.*

The compression functions satisfying T1 are exactly the *group 1* functions defined in [4].

In the Linicrypt framework, a PGV compression function f is specified via

- An output matrix $\boldsymbol{M} = \begin{bmatrix} \boldsymbol{m}_1 \\ \boldsymbol{m}_2 \end{bmatrix}$, where $\boldsymbol{m}_2$ is always $(0, 0, 1, 0)$ (corresponding to the fixed value c, as described below,) while $\boldsymbol{m}_1$ is one of $(1, 0, 0, 1)$, $(0, 1, 0, 1)$, $(1, 1, 0, 1)$ or $(0, 0, 1, 1)$.
- A single query constraint $\left(\begin{bmatrix} \boldsymbol{q}_K \\ \boldsymbol{q}_X \end{bmatrix}, \boldsymbol{a} \right)$, where each $\boldsymbol{q}_i$, $i \in \{K, X\}$, is one of $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, $(1, 1, 0, 0)$ or $(0, 0, 1, 0)$, and $\boldsymbol{a}$ is $(0, 0, 0, 1)$.

Here use $\boldsymbol{m}_2$ to capture the use of a constant value in the Linicrypt setting, as described in Section 2.1. Up to our convention regarding the constant value c, f'' corresponds to $\boldsymbol{m}_1$ while f' corresponds to $\begin{bmatrix} \boldsymbol{q}_K \\ \boldsymbol{q}_X \end{bmatrix}$. In particular, writing $\boldsymbol{q}_i = (q_i^1, q_i^2, q_i^3, q_i^4)$, $i \in \{K, X\}$, we have $f'(h, m) = \begin{bmatrix} \boldsymbol{q}'_K \\ \boldsymbol{q}'_X \end{bmatrix} \times (h, m, c)$, where $\boldsymbol{q}'_i = (q_i^1, q_i^2, q_i^3)$, $i \in \{K, X\}$. We also have $f''(h, m, y) = \boldsymbol{m}_1 \cdot (h, m, c, y)$.

We will use the following simple fact in several proofs below:

Proposition 3.3.3. *Over any field $\mathbb{F}$, a function of the form $g(x) = rx + s$, where $r, s \in \mathbb{F}$, is bijective iff $r \neq 0$.*

In the following, let f denote a compression function defined by $\boldsymbol{m}_1$,$\boldsymbol{m}_2$,$\boldsymbol{q}_K$,$\boldsymbol{q}_X$,$\boldsymbol{a}$, as described above, with respect to a fixed constant c. Define the following matrices

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{q}_K \\ \boldsymbol{q}_X \\ \boldsymbol{m}_2 \\ \boldsymbol{a} \end{bmatrix} \qquad \boldsymbol{S} = \begin{bmatrix} \boldsymbol{q}_K \\ \boldsymbol{m}_1 \\ \boldsymbol{m}_2 \\ \boldsymbol{a} \end{bmatrix}$$

Lemma 3.3.4. *Writing $\boldsymbol{m}_1 = (m_1^1, m_1^2, m_1^3, m_1^4)$, f satisfies P2 iff $m_1^4 \neq 0$.*

Proof. For fixed h, m, $f'(h, m, y) = \boldsymbol{m}_1 \cdot (h, m, c, y) = m_1^4 y \oplus s$, where s is fixed.

We note that every PGV compression function satisfies $m_1^4 \neq 0$ and hence P2. □

Lemma 3.3.5. *f satisfies P1 iff $\boldsymbol{R}$ is nonsingular.*

Proof. Let $\boldsymbol{R}' = \boldsymbol{R}$[1-3; 1-3] be the 3×3 principle submatrix of $\boldsymbol{R}$. Given the possible values of $\boldsymbol{q}_K$ and $\boldsymbol{q}_X$, $\boldsymbol{R}$ is nonsingular iff $\boldsymbol{R}'$ is nonsingular. For any h, m, $f'(h, m) = \boldsymbol{R}' \times (h, m, c)$, which is a bijection iff $\boldsymbol{R}'$ is nonsingular.

□

Lemma 3.3.6. *Assume $\boldsymbol{R}$ is nonsingular. Then f satisfies P3 iff $\boldsymbol{S}$ is nonsingular.*

Proof. First note that $f^*(k, x, y) = \boldsymbol{m}_1 \cdot (h, m, c, y) = \boldsymbol{m}_1 \cdot (\boldsymbol{R}^{-1} \times (k, x, c, y)) = (\boldsymbol{m}_1 \boldsymbol{R}^{-1}) \times (k, x, c, y)$. Let $\boldsymbol{u} = (u_1, u_2, u_3, u_4) = \boldsymbol{m}_1 \boldsymbol{R}^{-1}$. Then for fixed k, y, $f^*(k, x, y) = u_2 x \oplus s$, where s is fixed. Thus, it is enough to show $\boldsymbol{S}$ is nonsingular iff $u_2 \neq 0$. Since $\boldsymbol{R}$ is nonsingular, $\boldsymbol{S}$ is nonsingular iff $\boldsymbol{S}\boldsymbol{R}^{-1}$ is nonsingular. But $\boldsymbol{S}\boldsymbol{R}^{-1} = (\boldsymbol{e}_1, \boldsymbol{u}, \boldsymbol{e}_3, \boldsymbol{e}_4)$, which is nonsingular iff $u_2 \neq 0$.

□

Together, the preceding Lemmas gives the following

Lemma 3.3.7. *A PGV function f satisfies T1 iff $\boldsymbol{R}$ and $\boldsymbol{S}$ are nonsingular.*

In the rate-1 setting conditions (C1), (C2), (C3) become

(C1) $\boldsymbol{q}_K \notin \mathsf{span}(\{\boldsymbol{m}_1, \boldsymbol{m}_2\})$

(C2) $\boldsymbol{q}_X \notin \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{a}, \boldsymbol{m}_1, \boldsymbol{m}_2\})$

(C3) $\boldsymbol{a} \notin \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2\})$

Given the possible values of $\boldsymbol{q}_K, \boldsymbol{m}_1$ and $\boldsymbol{m}_2$, (C1) may be further simplified to

(C1) $\boldsymbol{q}_K \neq \boldsymbol{m}_2$

Lemma 3.3.8. *Suppose f is a PGV compression function specified by $\boldsymbol{q}_K$, $\boldsymbol{q}_X$, $\boldsymbol{m}_1$, $\boldsymbol{m}_2$, $\boldsymbol{a}$. If both $\boldsymbol{R}$ and $\boldsymbol{S}$ are nonsingular, then two of the conditions (C1), (C2), (C3) must fail.*

Proof. If (C1) fails, then S is necessarily singular, so we must show that under the assumption, both (C2) and (C3) fail. Clearly, if $\boldsymbol{S}$ is nonsingular,

$$\boldsymbol{q}_X \in \mathsf{span}(\mathsf{rows}(\boldsymbol{S})) = \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{a}\}) \qquad (*)$$

so (C2) fails. Now since $\boldsymbol{R}$ is nonsingular $\boldsymbol{q}_X \notin \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{m}_2, \boldsymbol{a}\})$, which in combination with (*) means that $\boldsymbol{m}_1 \in \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_2, \boldsymbol{a}\})$ (**). Noting that the last component of $\boldsymbol{m}_1$ is always nonzero, we have $\boldsymbol{m}_1 \notin \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_2\})$. Combining this last fact with (**), we conclude

$$\boldsymbol{a} \in \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2\}),$$

so that (C3) also fails.

□

Lemma 3.3.9. *Suppose f is a nondegenerate PGV compression function specified by $\boldsymbol{q}_K$, $\boldsymbol{q}_X$, $\boldsymbol{m}_1$, $\boldsymbol{m}_2$, $\boldsymbol{a}$. If one of $\boldsymbol{R}, \boldsymbol{S}$ is singular, then two of the conditions (C1), (C2), (C3) must hold.*

Proof. First, suppose (C2) fails, so $\boldsymbol{q}_X \in \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{a}\})$. Then, if $\boldsymbol{S}$ is singular,

$$\mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{a}\}) = \mathsf{span}(\mathsf{rows}(S)) \not\supseteq \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4\},$$

which means f is degenerate. Thus $\boldsymbol{S}$ is nonsingular. As in the proof of Lemma 3.3.8, this means $\boldsymbol{q}_K \neq \boldsymbol{m}_2$. Also if $\boldsymbol{S}$ in nonsingular then by the assumption, $\boldsymbol{R}$ is singular, so we must have $\boldsymbol{q}_K = \boldsymbol{q}_X$ or $\boldsymbol{q}_X = \boldsymbol{m}_2$. If the former holds, $\boldsymbol{a} \in \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2\})$ implies $\boldsymbol{q}_K = \boldsymbol{q}_X = \boldsymbol{a} \oplus \boldsymbol{m}_1$, and so

$$\mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{a}\}) = \mathsf{span}(\{\boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{a}\}) \not\supseteq \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4\},$$

If the latter holds then $\boldsymbol{a} \in \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2\})$ implies

$$\mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{a}\}) = \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{m}_1, \boldsymbol{m}_2\}) \not\supseteq \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4\},$$

So in both cases, by nondegeneracy $\boldsymbol{a} \notin \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2\})$, and we have that if (C2) fails, both (C1) and (C3) must hold.

Now suppose (C2) holds and (C1) fails. Then

$$\mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2\}) = \mathsf{span}(\{\boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2\}),$$

and so, if $\boldsymbol{a} \in \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2\})$,

$$\begin{aligned}\mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{a}\}) &= \mathsf{span}(\{\boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2\}) \\ &= \mathsf{span}(\{\boldsymbol{q}_X, \boldsymbol{m}_1, \boldsymbol{m}_2\}) \not\supseteq \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4\}.\end{aligned}$$

In conclusion, if (C2) holds, then one of (C1) or (C3) must hold.

□

Combining Lemmas 3.3.7,3.3.8, and 3.3.9, we obtain

Theorem 3.3.10. *A PGV compression function is group-1 iff it does not have a collision structure.*

Discussion Beyond validating the correspondence between our characterization of collision resistance for rate-1 compression functions and the well-known notion of group-1 for PGV, Theorem 3.3.10 situates our understanding of the group-1 functions as part of a general framework for collision resistance using Linicrypt. As an immediate application of Theorem 3.3.10 and Algorithm 1, we can automatically generate all group 1 PGV compression functions. [1] In particular, this provides a purely *syntactic* characterization using algebraic properties of the defining program $\mathcal{P}$, including the possibility of automated identification using **FindColStruct**. However, we note that [4, 5] provide a finer analysis of the PGV compression functions, also identifying the *group-2* functions which, although not collision resistant as compression functions, are still suitable for constructing collision-resistant hash functions and analyzing the preimage resistance of group-1 and -2 PGV functions. In the next chapter we give a

[1] A sample implementation in Octave is available at `https://github.com/zahrajavar/PGVCollisionResistantCompressionFunctions.git`

characterization of group-2 Linicrypt programs which generalizes the characterization of [4]. We leave a more quantitative analysis, as well as an investigation of preimage resistance properties, as future work.

3.4 Discussion

While a characterization of collision resistance for rate-1 (which is the most significant from a practical perspective) was already provided by [4], the more general Linicrypt setting provides several advantages. Our characterization is uniform and based solely on the syntax of programs (expressed algebraically). This is in contrast to the approach of [4], which is ad-hoc, and that of [5], which depends on semantic properties (i.e. bijectiveness) of the component functions. One benefit of our approach is that it immediately gives an efficient automated enumeration technique. We also note that in order to obtain security properties beyond collision resistance, such as preimage awareness, we may need to consider programs that make more than one call to the ideal cipher. The general characterization for collision resistance may be viewed as a step towards characterizations of other properties. Finally, the results of this chapter demonstrate that the utility of the Linicrypt framework is not limited to the random oracle model.

3.5 Some examples of functions from [4]

Example 3.5.1. *In this example we see a few PGV compression functions from [4] in the Linicrypt model:*

1. $E_m(m) + v$

$$\begin{array}{l} \underline{\mathcal{P}^E(v_1, v_2, v_3):} \\ \quad v_4 := E(t, v_2, v_2) \\ \quad return\ v_4 + v_3 \end{array}$$

This program is degenerate because $\boldsymbol{v}_1 \notin \mathsf{span}(\boldsymbol{v}_2, \boldsymbol{v}_4, \boldsymbol{v}_3, \boldsymbol{v}_3 + \boldsymbol{v}_4)$.

2. $E_h(m) + v$

$$\underline{\mathcal{P}^E(v_1, v_2, v_3) :}$$
$$v_4 := E(t, v_1, v_2)$$
$$return\ v_4 + v_3$$

where

$$\boldsymbol{M} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

and

$$\boldsymbol{q}_K = (1, 0, 0, 0) \quad \boldsymbol{q}_X = (0, 1, 0, 0) \quad \boldsymbol{a} = (0, 0, 1, 0)$$

thus,

(a) $\boldsymbol{q}_K \notin \mathsf{span}(\mathsf{rows}(\boldsymbol{M}))$

(b) $\boldsymbol{q}_X \notin \mathsf{span}(\boldsymbol{q}_K, \boldsymbol{a}, \mathsf{rows}(\boldsymbol{M}))$

(c) $\boldsymbol{a} \in \mathsf{span}(\boldsymbol{q}_K, \boldsymbol{q}_X, \mathsf{rows}(\boldsymbol{M}))$

and this is a collision structure for $i^* = 1$.

3. $E_h(m) + m$

$$\underline{\mathcal{P}^E(v_1, v_2, v_3) :}$$
$$v_4 := E(t, v_1, v_2)$$
$$return\ v_4 + v_2$$

where

$$\boldsymbol{M} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

and

$$\boldsymbol{q}_K = (1, 0, 0, 0) \quad \boldsymbol{q}_X = (0, 1, 0, 0) \quad \boldsymbol{a} = (0, 0, 1, 0)$$

thus,

(a) $\boldsymbol{q}_K \notin \mathsf{span}(\mathsf{rows}(\boldsymbol{M}))$

(b) $\boldsymbol{q}_X \in \mathsf{span}(\boldsymbol{q}_K, \boldsymbol{a}, \mathsf{rows}(\boldsymbol{M}))$

(c) $\boldsymbol{a} \in \mathsf{span}(\boldsymbol{q}_K, \boldsymbol{q}_X, \mathsf{rows}(\boldsymbol{M}))$

so there is no collision structure for this program.

Chapter 4

Further Results in the Ideal Cipher Model

In the first part of this chapter, we characterize compression functions $f : \mathbb{F}^k \to \mathbb{F}$ which are not collision resistant but for which the MD transform results in a collision-resistant hash function. The well-known group-2 PGV compression functions [32] are an instance of this class of programs.

In the second part of the chapter, we model double-block-length (DBL) compression functions in Linicrypt and we give a characterization of collision-resistant DBL compression functions. A DBL compression function is a compression function that generates an output length twice as long as the block length of the underlying cipher used in its construction. In particular, this means that if the block length is l, the birthday attack finds a collision with high probability after $\mathcal{O}(2^l)$ queries, while for a compression function with a single-block output, the bound is $\mathcal{O}(2^{l/2})$.

4.1 Group-2 Compression Functions

In 1994, Preneel, Govaerts, and Vandewalle [32] introduced a comprehensive approach to creating hash functions using a block cipher. They proposed a systematic method for constructing hash functions denoted as $H : (\{0,1\}^n)^* \to \{0,1\}^n$ based on a block cipher $E : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$. These construction involved iterating a compression function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ with a Merkle-Damgård (MD) transformation.

They asserted that 12 of these compression functions provided security. Subse-

quently, in 2002, Black, Rogaway, and Shrimpton [4] categorized these 64 compression functions into three groups. They rigorously demonstrated that the 12 compression functions in group-1 were secure in the ideal cipher model and established tight upper and lower bounds on their collision resistance. Furthermore, they determined that 8 of the compression functions within the PGV compression functions (group-2) remained secure when iterated with the MD transformation. The remaining 44 compression functions, categorized as group-3, were proven to be entirely insecure.

In Chapter 3, we established the characterization of collision-resistant hash functions by introducing a collision structure, which encompassed the group 1 PGV hash functions. This was accomplished by defining a collision structure that describes all the compression functions susceptible to collision attacks.

In this section, our goal is to delineate all the programs that may not be inherently collision-resistant but yield a collision-resistant hash function. This category includes the group-2 PGV compression functions and we refer to these Linicrypt programs as group-2 as well. To achieve this, we present a characterization of programs that we call "group-3". If a program lacks a group-3 collision structure but still possesses a collision structure, it is classified as group-2.

4.1.1 Characterizing Group-3 Compression Functions

Let program $\mathcal{P} : \mathbb{F}^k \to \mathbb{F}$ be a Linicrypt program in the ideal cipher model and $IV = \ell_0 \in \mathbb{F}$ is a fixed value. By iterating $\mathcal{P}$ on input vectors $\boldsymbol{x}_i \in \mathbb{F}^k$ with MD chaining, we get construction $\mathcal{H}^{\mathcal{P}} : \mathbb{F}^* \to \mathbb{F}$ where $l_i = \mathcal{P}(\boldsymbol{x}_i)$ for $i = 1, \cdots, b$ is the chaining value. Note, vector $\boldsymbol{x}_i$ is the input of program $\mathcal{P}$ in ith iteration and the first element of each $\boldsymbol{x}_i = (x_i^1, \cdots, x_i^k)$ is the chaining value so $x_{i+1}^1 = l_i$ for $i \in [0, b-1]$. And the construction $\mathcal{H}^{\mathcal{P}}$ runs on $\boldsymbol{W} = (\boldsymbol{w}_1, \cdots, \boldsymbol{w}_b)^\top$ where $\boldsymbol{w}_i = (x_i^2, \cdots, x_i^k)$ and $\mathcal{H}^{\mathcal{P}}(\boldsymbol{W}) = l_b$.

Security Definitions We assume the number of oracle queries the adversary makes to E and E^{-1} are q_E and q_D respectively.

Definition 4.1.1 ([25] Definition 2)**.** *Program $\mathcal{P}$ is (q, ϵ)-*collision resistant *if any oracle adversary $\mathcal{A}$ making at most $q = q_E + q_D$ queries has probability of success at most ϵ in the following game:*

$$(\boldsymbol{x}, \boldsymbol{x}') \leftarrow \mathcal{A}^{E, E^{-1}}(\lambda);\ \textit{return}\ (\boldsymbol{x} \neq \boldsymbol{x}')\ \textit{and}\ \mathcal{P}^E(\boldsymbol{x}) = \mathcal{P}^E(\boldsymbol{x}') \tag{4.1}$$

Definition 4.1.2. *Construction* $\mathcal{H}^{\mathcal{P}}$ *is* (q, ϵ)-collision resistant *if any oracle adversary* $\mathcal{A}$ *making at most* $q = q_E + q_D$ *queries has probability of success at most* ϵ *in the following game:*

$$(\boldsymbol{W}, \boldsymbol{W}') \leftarrow \mathcal{A}^{E,E^{-1}}(\lambda);\ \textit{return}\ (\boldsymbol{W} \neq \boldsymbol{W}')\ \textit{and}\ \mathcal{H}^{\mathcal{P}}(\boldsymbol{W}) = \mathcal{H}^{\mathcal{P}}(\boldsymbol{W}') \tag{4.2}$$

Definition 4.1.3 ([25] Definition 3)**.** *Program* $\mathcal{P}$ *is* (q, ϵ)-preimage resistant *if any oracle adversary* $\mathcal{A}$ *making at most* $q = q_E + q_D$ *queries has probability of success at most* ϵ *in the following game:*

$$\boldsymbol{\ell} \leftarrow \mathbb{F}^r; \boldsymbol{x} \leftarrow \mathcal{A}^{E,E^{-1}}(\boldsymbol{\ell}, \lambda); \textit{return}\ \mathcal{P}^E(\boldsymbol{x}) = \boldsymbol{\ell} \tag{4.3}$$

Definition 4.1.4. *Program* $\mathcal{P}$ *is* (q, ϵ)-preimage resistant for a fixed first input *if any oracle adversary* $\mathcal{A}$ *making at most* $q = q_E + q_D$ *queries has probability of success at most* ϵ *in the following game:*

$$\ell_0 \leftarrow \mathbb{F}; \boldsymbol{\ell} \leftarrow \mathbb{F}^r; \boldsymbol{x} \leftarrow \mathcal{A}^{E,E^{-1}}(\ell_0, \boldsymbol{\ell}, \lambda); \textit{return}\ \mathcal{P}^E(\boldsymbol{x}) = \boldsymbol{\ell}\ \textit{and}\ (x^1 = \ell_0) \tag{4.4}$$

Definition 4.1.5. *Construction* $\mathcal{H}^{\mathcal{P}}$ *is* (q, ϵ)-preimage resistant *if any oracle adversary* $\mathcal{A}$ *making at most* $q = q_E + q_D$ *queries has probability of success at most* ϵ *in the following game:*

$$\ell \leftarrow \mathbb{F}; \boldsymbol{W} \leftarrow \mathcal{A}^{E,E^{-1}}(\ell, \lambda); \textit{return}\ \mathcal{H}^{\mathcal{P}}(\boldsymbol{W}) = \ell \tag{4.5}$$

The following Lemma is just a special case of the collision resistance of the MD transform for collision-resistant compression functions defined by Linicrypt programs.

Lemma 4.1.6. *If* $\mathcal{P}$ *is collision-resistant then* $\mathcal{H}^{\mathcal{P}}$ *is collision resistant.*

Definition 4.1.7. *Let* $\mathcal{P} = (\boldsymbol{m}, \mathcal{C})$ *be a Linicrypt program with* n *constraints, where the first element of* $\mathcal{P}$*'s input corresponds to the chaining value. A* group-3 collision structure for $\mathcal{P}$ *is a tuple* $(i^*, c_1, \ldots, c_n)$ *where* $c_1, \cdots, c_n$ *is an ordering of* $\mathcal{C}$ *and* $i^* \in [1, n]$*, such that for* $i = i^*$ *at least two of the following conditions are true, and for every* $i \neq i^*$ *at least one of (C2) or (C3) is true.*

(C1) $\boldsymbol{q}_{K_i} \notin \mathsf{span}(\{\boldsymbol{q}_{K_1}, \ldots, \boldsymbol{q}_{K_{i-1}}\}, \{\boldsymbol{q}_{X_1}, \ldots, \boldsymbol{q}_{X_{i-1}}\}, \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1}\}, \mathsf{rows}(\boldsymbol{M}))$

(C2) $\boldsymbol{q}_{X_i} \notin \mathsf{span}(\{\mathbf{e}_1\}, \{\boldsymbol{q}_{K_1}, \ldots, \boldsymbol{q}_{K_i}\}, \{\boldsymbol{q}_{X_1}, \ldots, \boldsymbol{q}_{X_{i-1}}\}, \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_i\}, \mathsf{rows}(\boldsymbol{M}))$

(C3) $\boldsymbol{a}_i \notin \mathsf{span}(\{\mathbf{e}_1\}, \{\boldsymbol{q}_{K_1}, \ldots, \boldsymbol{q}_{K_i}\}, \{\boldsymbol{q}_{X_1}, \ldots, \boldsymbol{q}_{X_i}\}, \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1}\}, \mathsf{rows}(\boldsymbol{M}))$

An examination of the definition immediately gives the following:

Lemma 4.1.8. *Every group-3 collision structure is a collision structure.*

Lemma 4.1.9. *If a Linicrypt program $\mathcal{P} = (\boldsymbol{m}, \mathcal{C})$ with n constraints has a group-3 collision structure $(i^*, c_1, \ldots, c_n)$ then there exists an adversary $\mathcal{A}$ that makes at most n queries and has success with probability of 1 in Game 4.4.*

Proof. Assume (ℓ_0, ℓ) is given to the adversary. The adversary $\mathcal{A}$ creates the following constraints to determine the unknown elements of vector $\boldsymbol{v} = (\ell_0, v_2, \cdots, v_{k+n})^\top$.

- add $\boldsymbol{m}\boldsymbol{v} = \ell$.
- For every $i \in [1, n]$,
  - if (C1) holds, choose $K_i \in \mathbb{F}$ add the constraint $\boldsymbol{q}_{K_i} \cdot \boldsymbol{v} = K_i$
  - if (C2) holds, set $X_i := E^{-1}(t_i, \boldsymbol{q}_{K_i} \cdot \boldsymbol{v}, \boldsymbol{a}_i \cdot \boldsymbol{v})$ and add the constraint $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v} = X_i$
  - if (C3) holds, set $Y_i := E(t_i, \boldsymbol{q}_{K_i} \cdot \boldsymbol{v}, \boldsymbol{q}_{X_i} \cdot \boldsymbol{v})$ and add the constraint $\boldsymbol{a}_i \cdot \boldsymbol{v} = Y_i$
  - if (C2) and (C3) both hold, choose $X_i \in \mathbb{F}$ and set $Y_i := E(t_i, \boldsymbol{q}_{K_i} \cdot \boldsymbol{v}, X_i)$ and add the constraints $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v} = X_i$ and $\boldsymbol{a}_i \cdot \boldsymbol{v} = Y_i$

We claim that the constraints have a unique solution $\boldsymbol{v}$ such that if $\boldsymbol{x} = (\mathbf{e}_1, \ldots, \mathbf{e}_k) \cdot \boldsymbol{v}$, then $\mathcal{P}^E(\boldsymbol{x}) = \ell$.

The constraints that are added for the output vector and for every $i \in [1, n]$ are consistent, as a new constraint is added only in the case that the corresponding $\boldsymbol{q}_{K_i}, \boldsymbol{q}_{X_i}$ or $\boldsymbol{a}_i$ is independent of the vectors added for previous constraints, the image and e_1, so consistency is maintained. Once all constraints are added, nondegeneracy ensures that $\boldsymbol{v}$ is unique.

Finally, $\boldsymbol{v}$ is consistent with the values returned by E and E^{-1}. This means that $\boldsymbol{v}$ corresponds to the setting of base variables resulting from evaluating $\mathcal{P}^E(\boldsymbol{x})$, so from the $\boldsymbol{m}\boldsymbol{v} = \ell$ constraint, $\mathcal{P}^E(\boldsymbol{x}) = \ell$.

□

Lemma 4.1.10. *If a Linicrypt program $\mathcal{P} = (\boldsymbol{m}, \mathcal{C})$ with n constraints has a group-3 collision structure $(i^*, c_1, \ldots, c_n)$ then there exists an adversary $\mathcal{A}$ making at most nb queries that success with probability of 1 in Game 4.2.*

Proof. First, $\mathcal{A}$ picks an arbitrary vector $\boldsymbol{x}_1 \in \mathbb{F}^k$ and defines $\ell_0 = x_1^1$, then it runs $\mathcal{P}^E(\boldsymbol{x}_1)$ to get ℓ_1 then it puts $x_1^2 = \ell_1$ and picks the other $k-1$ elements of $\boldsymbol{x}_2$ arbitrary from $\mathbb{F}$ and it runs $\mathcal{P}^E(\boldsymbol{x}_2)$ to get l_2. It continues these iterations until it gets ℓ_b.

After this step $\mathcal{A}$, having preimage $\boldsymbol{x}_b$ for program $\mathcal{P}^E$, can find a second preimage $\boldsymbol{x}'_b \neq \boldsymbol{x}_b$ by Lemma 4.1.8. Now, by Lemma 4.1.9 and given image $\ell'_i = x'^1_{i+1}$ the adversary finds preimage $\boldsymbol{x}'_i$ for every $i \in \{b-1, \cdots, 1\}$.

In the last step, by Lemma 4.1.9 the adversary is given image ℓ'_1 and the first element of the preimage $x'^1_1 = \ell_0$, it finds preimage $\boldsymbol{x}'_1 = (\ell_0, x'^2_1, \cdots, x'^k_1)$. If $\boldsymbol{w}_i = (x_i^2, \cdots, x_i^k)$ and $\boldsymbol{w}'_i = (x'^2_i, \cdots, x'^k_i)$ for $i \in [1, b]$ then $\mathcal{H}^{\mathcal{P}}(\boldsymbol{W}) = \mathcal{H}^{\mathcal{P}}(\boldsymbol{W}') = \ell_b$. □

Lemma 4.1.11. *Let $\mathcal{P}$ be a Linicrypt program with n constraints. If there is an adversary $\mathcal{A}$ for $\mathcal{H}^{\mathcal{P}}$ making at most N oracle queries with success probability $> (N/n)^n/|\mathbb{F}|$ in the collision-resistance game (Game 4.2) then $\mathcal{P}$ is either degenerate or has a group-3 collision structure $(i^*, c_1, \ldots, c_n)$.*

Proof. Without loss of generality, we assume

1. $\mathcal{A}$ does not repeat a query or make the inverse of a query it has already made. This can be achieved by recording queries as they are made.

2. For queries made in the execution of $\mathcal{H}^{\mathcal{P}}(\boldsymbol{W})$ and $\mathcal{H}^{\mathcal{P}}(\boldsymbol{W}')$, $\mathcal{A}$ makes either the query or its corresponding inverse query before returning. For Game 4.2, this is achieved by having $\mathcal{A}$ run $\mathcal{H}^{\mathcal{P}}(\boldsymbol{W})$ and $\mathcal{H}^{\mathcal{P}}(\boldsymbol{W}')$ before returning and making the corresponding queries subject to restriction (1).

3. $\mathcal{A}$ returns $(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_b)$ and $(\boldsymbol{v}'_1, \cdots, \boldsymbol{v}'_{b'})$ which are respectively, the settings of base variables determined by the execution of $\mathcal{P}(\boldsymbol{x}_1), \cdots, \mathcal{P}(\boldsymbol{x}_b)$ and $\mathcal{P}(\boldsymbol{x}'_1), \cdots, \mathcal{P}(\boldsymbol{x}'_{b'})$.

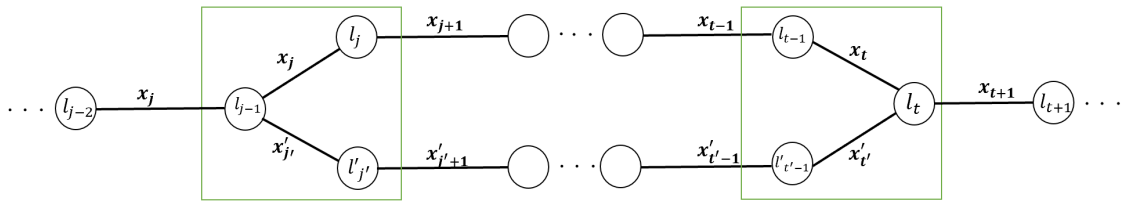

Figure 4.1: Collision attack for MD hash functions with fixed initial value

If $\boldsymbol{W} = (\boldsymbol{w}_1, \cdots, \boldsymbol{w}_b)^\top \neq \boldsymbol{W}' = (\boldsymbol{w}'_1, \cdots, \boldsymbol{w}'_{b'})^\top$ where $b \geq b'$ are the colliding inputs returned by $\mathcal{A}$ then at least one of the following two events must happen at some point in the MD chain. Without loss of generality, we assume for some $1 \leq j \leq t \leq b$ and $1 \leq j' \leq t' \leq b'$,

$$\boldsymbol{x}_j = (\ell_{j-1}, x_j^1, \cdots, x_j^k) \ \wedge \ \boldsymbol{x}'_{j'} = (l_{j'-1}, x_{j'}'^1, \cdots, x_{j'}'^k) \ \wedge \ \mathcal{P}(\boldsymbol{x}_j) \neq \mathcal{P}(\boldsymbol{x}'_{j'}) \qquad (\star)$$

$$\boldsymbol{x}_t = (\ell_{t-1}, x_t^1, \cdots, x_t^k) \neq \boldsymbol{x}'_{t'} = (l_{t'-1}, x_{t'}'^1, \cdots, x_{t'}'^k) \ \wedge \ \mathcal{P}(\boldsymbol{x}_k) = \mathcal{P}(\boldsymbol{x}'_{t'}) \qquad (\star\star)$$

For either of these two cases to occur, the adversary must pick either of the following two orders of making the queries, we show in both cases the adversary has to win Game 4.4. In the first ordering, the order of queries is as follows,

1. ℓ_{j-1} is fixed and determined in the previous iterations.
2. $\mathcal{A}$ picks two different values $\boldsymbol{w}_j \neq \boldsymbol{w}'_{j'}$ and determines $\ell_j = \mathcal{P}^E(\boldsymbol{x}_j)$ and $\ell'_{j'} = \mathcal{P}^E(\boldsymbol{x}'_{j'})$.
3. $\mathcal{A}$ determines all the chaining values $\ell_i = \mathcal{P}^E(\boldsymbol{x}_i)$ for $i = j, \cdots, t-1$ and $\ell_i = \mathcal{P}^E(\boldsymbol{x}'_i)$ for $i = j', \cdots, t'-1$.
4. $\mathcal{A}$ has to find $\boldsymbol{x}_t$ and $\boldsymbol{x}'_{t'}$ where $\ell_{t-1} \neq \ell'_{t'-1}$ are fixed and $\ell_t = \ell'_{t'}$.

In the second ordering, the order of queries is the reverse

1. $\mathcal{A}$ determines the values in $(\star\star)$ first, which is either a collision or a 2nd-preimage for $\mathcal{P}$.
2. $\mathcal{A}$ finds the values in $(\star)$ after, where $\ell'_{j'} \neq \ell_j$ and ℓ_{j-1} are fixed so $\mathcal{A}$ has to win Game 4.4 again.

Thus, we assume the adversary to win Game 4.2, has to win Game 4.4 when $x_1^1 = l_0$ is fixed.

The assumptions (1),(2),(3) imply that for an oracle constraint $c_i = (t_i, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a})$ occurring in $\mathcal{P}$, $\mathcal{A}$ determines the value of triples $(\boldsymbol{q}_K \cdot \boldsymbol{v}, \boldsymbol{q}_X \cdot \boldsymbol{v}, \boldsymbol{a} \cdot \boldsymbol{v})$ through exactly one of its N_i queries with nonce t_i, which is either a E-query or E^{-1}-query.

Based on this fact, for $1 \leq j \leq b$ and $1 \leq j' \leq b'$, we define mappings $T_j, T'_{j'} : \mathcal{C} \to [N]$ where $\mathcal{C}$ is the set of constraints in $\mathcal{P}$ and the $T_j(c_i)$th and $T'_{j'}(c_i)$th adversary queries correspond to constraint c_i in the computation of $\mathcal{P}(\boldsymbol{x}_i)$ and $\mathcal{P}(\boldsymbol{x}'_i)$, and determine the triple $(\boldsymbol{q}_K \cdot \boldsymbol{v}_i, \boldsymbol{q}_X \cdot \boldsymbol{v}_i, \boldsymbol{a} \cdot \boldsymbol{v}_i)$ and $(\boldsymbol{q}_K \cdot \boldsymbol{v}'_i, \boldsymbol{q}_X \cdot \boldsymbol{v}'_i, \boldsymbol{a} \cdot \boldsymbol{v}'_i)$ respectively.

We show if $\mathcal{A}$ wins Game 4.4, then there is a group 3 collision structure. Game 4.4 implies that the maximum number of mapping T is $(N/n)^n$. Using the pigeonhole principle and $\mathcal{A}$'s advantage in each game, there is a specific mapping T for which $\mathcal{A}$'s advantage when using this mapping is at least $1/|\mathbb{F}|$.

We will assume that the adversary is using this mapping — for any other mapping, it returns $\perp$ as its last action.

Since $H^{\mathcal{P}}$ is not collision resistant then $\mathcal{P}$ is not collision resistant 4.1.6, and there is a collision structure $(i^*, c_1, \ldots, c_n)$ for $\mathcal{P}$.

Knowing there are no useless constraints in $\mathcal{P}$, each query result is used as part of the input to another useful query, or as part of the final output of the program. Thus having ℓ_1 and ℓ_0 fixed, for each query, either part of the input or the output is fixed. By way of contradiction, we assume none of (C2) and (C3) hold for a constraint c_i, $1 \leq i \leq n$ in $\mathcal{P}$;

$$\boldsymbol{q}_{X_i} = \sum_{j \leq i} \pi_j \boldsymbol{q}_{K_j} + \sum_{j<i} \rho_j \boldsymbol{q}_{X_j} + \sum_{j \leq i} \varsigma_j \boldsymbol{a}_j + \boldsymbol{\tau M} + \boldsymbol{\kappa} \mathbf{e}_1$$

$$\boldsymbol{a}_i = \sum_{j \leq i} \alpha_j \boldsymbol{q}_{K_j} + \sum_{j \leq i} \beta_j \boldsymbol{q}_{X_j} + \sum_{j<i} \gamma_j \boldsymbol{a}_j + \boldsymbol{\delta M} + \boldsymbol{\nu} \mathbf{e}_1$$

Multiplying both sides by $\boldsymbol{v}$

$$\boldsymbol{q}_{X_i} \cdot \boldsymbol{v} = \sum_{j \leq i} \pi_j \boldsymbol{q}_{K_j} \cdot \boldsymbol{v} + \sum_{j<i} \rho_j \boldsymbol{q}_{X_j} \cdot \boldsymbol{v} + \sum_{j \leq i} \varsigma_j \boldsymbol{a}_j \cdot \boldsymbol{v} + \boldsymbol{\tau M} \cdot \boldsymbol{v} + \boldsymbol{\kappa} \mathbf{e}_1 \cdot \boldsymbol{v}$$

$$\boldsymbol{a}_i \cdot \boldsymbol{v} = \sum_{j \leq i} \alpha_j \boldsymbol{q}_{K_j} \cdot \boldsymbol{v} + \sum_{j \leq i} \beta_j \boldsymbol{q}_{X_j} \cdot \boldsymbol{v} + \sum_{j<i} \gamma_j \boldsymbol{a}_j \cdot \boldsymbol{v} + \boldsymbol{\delta M} \cdot \boldsymbol{v} + \boldsymbol{\nu} \mathbf{e}_1 \cdot \boldsymbol{v}$$

Since in Game 4.4, the output of the program and the first element of the input are fixed, the last two terms on the right-hand side of both equations are fixed. If the adversary is making the query $T(c_i)$ as an encryption query then in the second equation all the terms on the right-hand side are fixed and the advantage of the adversary to make the equality is $\leq 1/|\mathbb{F}|$ which is a contradiction. And if $T(c_i)$ is a decryption query then the first equation on the right-hand side is fixed so the adversary advantage is $\leq 1/|\mathbb{F}|$ which is a contradiction. $\square$

4.2 Double-Block-Length Hash Functions

Most real-world block ciphers have relatively small block lengths, typically not exceeding 128 bits. For example, well-known ciphers like AES, RC5, RC6, and Twofish use 128-bit blocks, while DES, IDEA, and Blowfish use even smaller 64-bit blocks. While short block lengths aren't usually a problem for encryption, they are unsuitable for single block-length compression functions. Using a block cipher with 128-bit blocks for such compression functions is highly vulnerable to collision attacks, with a best-case effective security of at most 64 bits, since by the birthday bound an attacker can find a collision with probability at least $\frac{1}{2}$ by making at most $O(2^{64})$ queries. This level of security is considered inadequate for many cryptographic applications.

A double-block-length (DBL) hash function is a type of hash function that produces an output length twice the size of the block length. Thus any hash function where $f : \mathbb{F}^n \to \mathbb{F}^2$ and $n > 2$ is a DBL hash function. The first time DBL hash functions were used was in the design of MDC-2 and MDC-4 in 1988 by Meyer and Schilling [30]. One class of well-studied DBL hash functions map $3l$ bits to $2l$ bits and can be categorized into two types: both use block ciphers with block-length l, but the first uses a $2l$-bit key (DBL^{2l}) and the second uses and l-bit key (DBL^{l}). Some of the examples of DBL^{2l} compression functions are Tandem-DM and Abreast-DM [22] and Hirose's function [19].

The extension of Linicrypt to the ideal cipher model, proposed in Chapter 3, may already be used to model the construction of DBL^{l} compression functions with three block length input and double-block-length output and characterize their security. In this section, we analyze DBL^{2l} compression functions. This requires an extension of the approach of Chapter 3, as the block cipher in this case will be a function $E : \mathbb{F}^2 \times \mathbb{F} \to \mathbb{F}^1$, of the form of $E_{K_1 \,\|\, K_2}(X)$.

The following is the generalization to an arbitrary finite field $\mathbb{F}$ of the definition of DBL^{2l} compression functions, which we will use to model such functions in Linicrypt

Definition 4.2.1. *[18] Let $f : \mathbb{F}^2 \times \mathbb{F} \to \mathbb{F}^2$ be a compression function such that $(g_i, h_i) = f(g_{i-1}, h_{i-1}, m_i)$, where $g_i, h_i, m_i \in \mathbb{F}$. f consists of functions f_U and f_L which call a block cipher E where $E : \mathbb{F}^2 \times \mathbb{F} \to \mathbb{F}$,*

$$g_i = f_U(g_{i-1}, h_{i-1}, m_i) = E(K_{U1} \,\|\, K_{U2}, X_U) + Z_U$$

[1] We are considering a general $\mathbb{F}$, which will include the special case of $\mathbb{F} = \mathbb{GF}(2^l)$ considered in the existing literature

$$h_i = f_L(g_{i-1}, h_{i-1}, m_i) = E(K_{L1} \,||\, K_{L2}, X_L) + Z_L$$

where $K_{U1}, K_{U2}, X_U, Z_U, K_{L1}, K_{L2}, X_L, Z_L \in \mathbb{F}$ *are represented by linear combinations of* $g_{i-1}, h_{i-1}, m_i \in \mathbb{F}$.

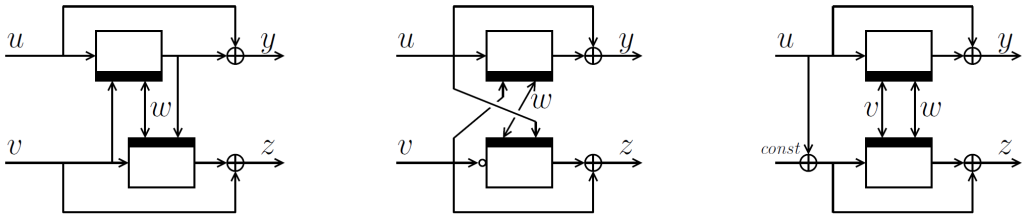

Figure 4.2: Tandem-DM, Abreast-DM and Hirose's functions from [26]

4.2.1 Double-Block-Length Hash Functions in Linicrypt

To incorporate the DBL compression function into Linicrypt, we need to extend the Linicrypt framework to support ideal block ciphers with double the key length, denoted as $E : \mathbb{F}^2 \times \mathbb{F} \to \mathbb{F}$. In the Linicrypt framework, the concatenation of two values, $K1 \,\|\, K2$, is represented as two separate inputs, denoted as $(K1, K2)$. Consequently, the ideal cipher now accepts three input values, namely $E(t, K1, K2, X)$, and it produces the output value Y, with all values belonging to $\mathbb{F}$.

All the existing definitions and theorems from Chapter 3 can be extended naturally to accommodate this change. Here, we present Definitions 3.2.3 and 3.2.5, along with Lemmas 3.2.6 and 3.2.7 for compression functions that use two keys. Similar adjustments can be made to other definitions as needed.

Assuming that $\boldsymbol{q}_{K1_i}$ and $\boldsymbol{q}_{K2_i}$ correspond to the key values $K1$ and $K2$, respectively, and that each constraint is in the form of $c_i = (t_i, \boldsymbol{q}_{K1_i}, \boldsymbol{q}_{K2_i}, \boldsymbol{q}_{X_i}, \boldsymbol{a}_i)$, we define degenerate programs $\mathcal{P}$ as follows:

Definition 4.2.2. *Program* $\mathcal{P} = (M, \mathcal{C})$ *is* degenerate *if*

$$\begin{aligned}\mathsf{span}(\{\mathbf{e}_1, \ldots, \mathbf{e}_{k+n}\}) \not\subseteq \mathsf{span}(&\{\boldsymbol{q}_{K1} \mid (t, \boldsymbol{q}_{K1}, \boldsymbol{q}_{K2}, \boldsymbol{q}_X, \boldsymbol{a}) \in \mathcal{C}\} \\ &\cup \{\boldsymbol{q}_{K2} \mid (t, \boldsymbol{q}_{K1}, \boldsymbol{q}_{K2}, \boldsymbol{q}_X, \boldsymbol{a}) \in \mathcal{C}\} \\ &\cup \{\boldsymbol{q}_X \mid (t, \boldsymbol{q}_{K1}, \boldsymbol{q}_{K2}, \boldsymbol{q}_X, \boldsymbol{a}) \in \mathcal{C}\} \cup \{\boldsymbol{a} \mid (t, \boldsymbol{q}_K, \boldsymbol{q}_X, \boldsymbol{a}) \in \mathcal{C}\} \\ &\cup \mathsf{rows}(\boldsymbol{M}))\end{aligned}$$

To characterize collision resistance within this context, we make a modification to condition 1 as described in 3.2.5. If either of the two key parts is not in the span of previous queries and the program output, we consider the key to be independent of fixed values. Therefore, we define the collision structure for a program $\mathcal{P}$ in the following manner:

Definition 4.2.3. *Let $\mathcal{P} = (\boldsymbol{M}, \mathcal{C})$ be a Linicrypt program. A collision structure for $\mathcal{P}$ is a tuple $(i^*, c_1, \ldots, c_n)$ where $c_1, \cdots, c_n$ is an ordering of $\mathcal{C}$ and $i^* \in [1, n]$, such that for $i = i^*$ at least two of the following conditions are true, and for all $i > i^*$ at least one of (C2) or (C3) is true.*

(C1)

$$\boldsymbol{q}_{K1_i} \notin \mathsf{span}(\{\boldsymbol{q}_{K1_1}, \boldsymbol{q}_{K2_1} \ldots, \boldsymbol{q}_{K1_{i-1}}, \boldsymbol{q}_{K2_{i-1}}, \boldsymbol{q}_{K2_i}\}, \{\boldsymbol{q}_{X_1}, \ldots \boldsymbol{q}_{X_{i-1}}\}, \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1}\}, \mathsf{rows}(\boldsymbol{M}))$$

or

$$\boldsymbol{q}_{K2_i} \notin \mathsf{span}(\{\boldsymbol{q}_{K1_1}, \boldsymbol{q}_{K2_1} \ldots, \boldsymbol{q}_{K1_{i-1}}, \boldsymbol{q}_{K2_{i-1}}, \boldsymbol{q}_{K1_i}\}, \{\boldsymbol{q}_{X_1}, \ldots \boldsymbol{q}_{X_{i-1}}\}, \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1}\}, \mathsf{rows}(\boldsymbol{M}))$$

(C2) $\boldsymbol{q}_{X_i} \notin \mathsf{span}(\{\boldsymbol{q}_{K1_1}, \boldsymbol{q}_{K2_1} \ldots, \boldsymbol{q}_{K1_i}, \boldsymbol{q}_{K2_i}\}, \{\boldsymbol{q}_{X_1}, \ldots \boldsymbol{q}_{X_{i-1}}\}, \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_i\}, \mathsf{rows}(\boldsymbol{M}))$

(C3) $\boldsymbol{a}_i \notin \mathsf{span}(\{\boldsymbol{q}_{K1_1}, \boldsymbol{q}_{K2_1} \ldots, \boldsymbol{q}_{K1_i}, \boldsymbol{q}_{K2_i}\}, \{\boldsymbol{q}_{X_1}, \ldots \boldsymbol{q}_{X_i}\}, \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1}\}, \mathsf{rows}(\boldsymbol{M}))$

In the next lemma, we show if there is a collision structure for a DBL^{2l} Linicrypt program then there is a collision attack.

Lemma 4.2.4. *If a Linicrypt program $\mathcal{P}$ with n constraints has a collision structure $(i^*, c_1, \ldots, c_n)$ then there exists a collision adversary $\mathcal{A}$ with access to E and E^{-1} which given an input $\boldsymbol{x}$ makes at most $2n$ queries and returns $\boldsymbol{x}' \neq \boldsymbol{x}$ such that $\mathcal{P}^E(\boldsymbol{x}') = \mathcal{P}^E(\boldsymbol{x})$, and so has success probability of 1 in Game 4.3.*

Proof. The adversary $\mathcal{A}$ first determines a setting of the base variables $\boldsymbol{v}$ by running $\mathcal{P}^E(\boldsymbol{x})$, and creates linear constraints on unknowns $\boldsymbol{v}'$ as follows:

- add constraint $\boldsymbol{M}\boldsymbol{v}' = \boldsymbol{M}\boldsymbol{v}$

- for $i < i^*$, add constraints $\boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}$, $\boldsymbol{q}_{K2_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{K2_i} \cdot \boldsymbol{v}$, $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}$ and $\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v}$
- For $i \geq i^*$,
  - If any of two parts of (C1) holds, if it's the first part, choose $K'1_i \in \mathbb{F}$ so that $K'1_i \neq \boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}$ and add the constraint $\boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}' = K'1_i$. If it's the second part, choose $K'2_i \in \mathbb{F}$ so that $K'2_i \neq \boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}$ and add the constraint $\boldsymbol{q}_{K2_i} \cdot \boldsymbol{v}' = K'2_i$. If one of the two-part doesn't hold we add constraint $\boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}' = K1_i$ or $\boldsymbol{q}_{K2_i} \cdot \boldsymbol{v}' = K2_i$.
  - If (C2) holds, set $X'_i := E^{-1}(t_i, \boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}', \boldsymbol{q}_{K2_i} \cdot \boldsymbol{v}' \cdot \boldsymbol{a}_i \cdot \boldsymbol{v}')$ and add the constraint $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = X'_i$.
  - If (C3) holds, set $Y'_i := E(t_i, \boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}', \boldsymbol{q}_{K2_i} \cdot \boldsymbol{v}', \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}')$ and add the constraint $\boldsymbol{a}_i \cdot \boldsymbol{v}' = Y'_i$
  - If (C2) and (C3) both hold, choose $X'_i \in \mathbb{F}$ such that $X'_i \neq \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}$, set $Y_i := E(t_i, \boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}', \boldsymbol{q}_{K2_i} \cdot \boldsymbol{v}', X'_i)$ and add the constraints $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = X'_i$ and $\boldsymbol{a}_i \cdot \boldsymbol{v}' = Y'_i$

We claim that the constraints have a unique solution $\boldsymbol{v}' \neq \boldsymbol{v}$ such that if $x'_i = \mathbf{e}_i \cdot \boldsymbol{v}'$, $1 \leq i \leq k$, then $\boldsymbol{x}' \neq \boldsymbol{x}$ and $\mathcal{P}^E(\boldsymbol{x}') = \mathcal{P}^E(\boldsymbol{x})$.

To see that $\boldsymbol{v}' \neq \boldsymbol{v}$, note that for $i = i^*$, either (C1) holds, or both (C2) and (C3) hold. The choice of K'_{i^*} in the first case and X'_{i^*} in the second, ensure $\boldsymbol{v}' \neq \boldsymbol{v}$.

The constraints that are added for the output matrix and for $i < i^*$ are consistent, as they already have a solution, namely $\boldsymbol{v}$. For $i \geq i^*$, a new constraint is added only in the case that the corresponding $\boldsymbol{q}_{K1_i}$ or $\boldsymbol{q}_{K2_i}, \boldsymbol{q}_{X_i}$ or $\boldsymbol{a}_i$ is independent of the vectors added in previous constraints, and so consistency is maintained as constraints are added. Once all constraints are added, nondegeneracy ensures that $\boldsymbol{v}'$ is unique.

Finally, $\boldsymbol{v}'$ is consistent with the values returned by E and E^{-1}. This means that $\boldsymbol{v}'$ corresponds to the setting of base variables resulting from evaluating $\mathcal{P}^E(\boldsymbol{x}')$, so from $\boldsymbol{v}' \neq \boldsymbol{v}$ we conclude $\boldsymbol{x}' \neq \boldsymbol{x}$, and from the $\boldsymbol{M}$ constraint, $\mathcal{P}^E(\boldsymbol{x}') = \mathcal{P}^E(\boldsymbol{x})$.

□

The following lemma demonstrates that if there exists a collision adversary or a 2nd-preimage adversary with a certain specified success probability for a DBL^{2l} Linicrypt program, then there is a corresponding collision structure for that program.

Lemma 4.2.5. *Let $\mathcal{P}$ be a Linicrypt program with n constraints. If there is an adversary $\mathcal{A}$ for $\mathcal{P}$ making at most N oracle queries with success probability $> (N/n)^{2n}/|\mathbb{F}|$ in the collision-resistance game (Game 4.1) or success probability $> (N/n)^n/|\mathbb{F}|$ in the 2nd-preimage game (Game 4.3) then the $\mathcal{P}$ is either degenerate or has a collision structure $(i^*, c_1, \ldots, c_n)$.*

Proof. We may assume the following without loss of generality:

1. $\mathcal{A}$ does not repeat a query or make the inverse of a query it has already made. This can be achieved by recording queries as they are made.

2. For queries made in the execution of $\mathcal{P}(\boldsymbol{x})$ and $\mathcal{P}(\boldsymbol{x}')$, $\mathcal{A}$ makes either the query or its corresponding inverse query before returning. For Game 4.1, this is achieved by having $\mathcal{A}$ run $\mathcal{P}(\boldsymbol{x})$ and $\mathcal{P}(\boldsymbol{x}')$ before returning and making the corresponding queries subject to restriction (1). For Game 4.3 this is achieved by having $\mathcal{A}$ initially make all the queries that result from running $\mathcal{P}(\boldsymbol{x})$ and also running $\mathcal{P}(\boldsymbol{x}')$ before returning, and making any corresponding query, subject to restriction (1), before returning.

3. $\mathcal{A}$ returns $\boldsymbol{v}, \boldsymbol{v}'$ which are the settings of base variables determined by the execution of $\mathcal{P}(\boldsymbol{x})$ and $\mathcal{P}(\boldsymbol{x}')$, respectively.

The assumptions imply that for an oracle constraint $c_i = (t_i, \boldsymbol{q}_{K1}, \boldsymbol{q}_{K2}, \boldsymbol{q}_X, \boldsymbol{a})$ occurring in $\mathcal{P}$, $\mathcal{A}$ determines the value of triples $(\boldsymbol{q}_{K1} \cdot \boldsymbol{v}, \boldsymbol{q}_{K2} \cdot \boldsymbol{v}, \boldsymbol{q}_X \cdot \boldsymbol{v}, \boldsymbol{a} \cdot \boldsymbol{v})$ through exactly one of its N queries, which is either a E-query or E^{-1}-query. Based on this fact, we define two mappings $T, T' : \mathcal{C} \to [N]$ where $\mathcal{C}$ is the set of constraints in $\mathcal{P}$ and the $T(c_i)$th and $T'(c_i)$th adversary queries correspond to constraint c_i in the computation of $\mathcal{P}(\boldsymbol{x})$ and $\mathcal{P}(\boldsymbol{x}')$, and determine the triple $(\boldsymbol{q}_{K1} \cdot \boldsymbol{v}, \boldsymbol{q}_{K2} \cdot \boldsymbol{v}, \boldsymbol{q}_X \cdot \boldsymbol{v}, \boldsymbol{a} \cdot \boldsymbol{v})$ and $(\boldsymbol{q}_{K1} \cdot \boldsymbol{v}', \boldsymbol{q}_{K2} \cdot \boldsymbol{v}', \boldsymbol{q}_X \cdot \boldsymbol{v}', \boldsymbol{a} \cdot \boldsymbol{v}')$ respectively. In Game 4.1, $T(c_i)$ and $T'(c_i)$ are each mapped to one of N_i queries made by $\mathcal{A}$ using nonce t_i, so the maximum number of possible mappings (T, T') is $(N/n)^{2n}$. In Game 4.3, Assumption 2 implies that T is fixed, so the number of possible mappings (T, T') is $(N/n)^n$. Using the pigeonhole principle and $\mathcal{A}$'s assumed advantage in each game, there is a specific mapping (T, T') for which $\mathcal{A}$'s advantage when using this mapping is at least $1/|\mathbb{F}|$. We will assume that the adversary is using this mapping — for any other mapping, it returns $\perp$ as its last action.

Using the same terminology as [25], a query $c \in \mathcal{C}$ is *convergent* if $T(c) = T'(c)$, and *divergent* otherwise. Because $\boldsymbol{x} \neq \boldsymbol{x}'$ is a collision and $\mathcal{P}$ is nondegenerate there

is at least one divergent constraint. Define $\texttt{finish}(c) = \max\{T(c), T'(c)\}$. Since each constraint has a distinct nonce, two different program constraints can not be mapped to the same adversary query, thus, function `finish` is injective. we will show that there is an ordering of $\mathcal{C}$ as $(c_1, \ldots, c_n)$ where the convergent constraints come first, in any order, followed by divergent constraints in some decreasing order, and letting i^* denote the index of the first divergent constraint, we claim that $(i^*, c_1, \ldots, c_n)$ is a collision structure for $\mathcal{P}$.

For $i < i^*$, since each c_i is convergent we have $\boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{K1_i} \cdot \boldsymbol{v}$, $\boldsymbol{q}_{K2_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{K2_i} \cdot \boldsymbol{v}$, $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}$ and $\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v}$ and because $\mathcal{P}(\boldsymbol{x}) = \mathcal{P}(\boldsymbol{x}')$ we have $\boldsymbol{M}\boldsymbol{v}' = \boldsymbol{M}\boldsymbol{v}$.

For $i = i^*$ the query c_i is divergent thus at least one of the following inequalities holds $\boldsymbol{q}_{Kj_i} \cdot \boldsymbol{v}' \neq \boldsymbol{q}_{Kj_i} \cdot \boldsymbol{v}$ where $j \in \{1, 2\}$, $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' \neq \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}$ or $\boldsymbol{a}_i \cdot \boldsymbol{v}' \neq \boldsymbol{a}_i \cdot \boldsymbol{v}$. If $\boldsymbol{a}_i \cdot \boldsymbol{v}' \neq \boldsymbol{a}_i \cdot \boldsymbol{v}$ or $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' \neq \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}$ then at least one of the other inequalities hold since the ideal cipher is a permutation when the key is fixed.

This gives five possible cases, and we show all the cases satisfy at least two conditions of definition 4.2.3. Without loss of generality, in all cases, we assume that $T(c_i) < T'(c_i)$. Also if we write $K_i = K_i'$ it means both part of the keys are equal ($K1_i = K'1_i$ and $K2_i = K'2_i$) and if the keys are not equal $K_i \neq K_i'$ then either of the two parts can be non-equal ($K1_i \neq K'1_i$ or $K2_i \neq K'2_i$)

1. $K_i = K_i$ and $X_i \neq X_i'$ and $Y_i \neq Y_i'$.

   In this case, we prove both conditions (C2) and (C3) hold. By way of contradiction assume (C3) does not hold, say

   $$\boldsymbol{a}_i = \sum_{j \leq i} \alpha 1_j \boldsymbol{q}_{K1_j} + \alpha 2_j \boldsymbol{q}_{K2_j} + \sum_{j \leq i} \beta_j \boldsymbol{q}_{X_j} + \sum_{j < i} \gamma_j \boldsymbol{a}_j + \boldsymbol{\delta}\boldsymbol{M}, \tag{4.6}$$

   for some $\boldsymbol{\alpha 1}, \boldsymbol{\alpha 2}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\delta}$. After multiplying both side of the equation by $(\boldsymbol{v}' - \boldsymbol{v})$ and considering that all the queries before i^* are convergent, $\boldsymbol{M}(\boldsymbol{v}' - \boldsymbol{v}) = 0$, and $K1_i = K'1_i$ and $K2_i = K'2_i$ we have

   $$\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v} + \beta_i \boldsymbol{q}_{X_i} \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

   Also because $Y_i \neq Y_i'$ and $X_i \neq X_i'$ we know $\beta_i \neq 0$. If $T'(c_i)$ is an encryption query then the right-hand side of the equation is a fixed value but the left-hand side is a query result, so the advantage of the adversary is $\leq 1/|\mathbb{F}|$ contrary to

assumption. If $T'(c_i)$ is a decryption query then isolating $\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}'$ gives us

$$\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{X_i} \cdot \boldsymbol{v} + \frac{1}{\beta_i} \boldsymbol{a}_i \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

and again the right-hand side of the equation is determined while the left-hand side is a random value, again giving a contradiction. The proof for condition (C2) is similar.

2. $K_i \neq K'_i$ and $X_i = X'_i$ and $Y_i \neq Y'_i$.

   Because $K1_i \neq K'1_i$ or $K2_i \neq K'2_i$, condition (C1) holds. We want to show that $T'(c_i)$ is an encryption query and (C3) holds. If $T'(c_i)$ is not an encryption then X_i is fixed and X'_i random so $X_i = X'_i$ holds with probability $\leq 1/|\mathbb{F}|$. If condition (C3) does not hold then Equation 4.6 holds. Multiplying the equation to $(\boldsymbol{v}' - \boldsymbol{v})$ and applying $X_i = X'_i$ and $\boldsymbol{M}(\boldsymbol{v} - \boldsymbol{v}') = 0$ gives

   $$\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v} + \alpha 1_i \boldsymbol{q}_{K1_i} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \alpha 2_i \boldsymbol{q}_{K2_i} \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

   Note, at most one of the $\alpha 1$ and $\alpha 2$, can be equal to zero. The left-hand side of the above equation is a random value and the right-hand side is a fixed value, so the adversary advantage again is $\leq 1/|\mathbb{F}|$.

3. $K_i \neq K'_i$ and $X_i \neq X'_i$ and $Y_i = Y'_i$.

   This is similar to the preceding case. Because $K1_i \neq K'1_i$ or $K2_i \neq K'2_i$, condition (C1) holds. We want to show that $T'(c_i)$ has to be a decryption query and (C2) holds. If it is not a decryption query then the probability of a random value Y'_i being equal to the fixed value Y_i would be $1/|\mathbb{F}|$, which contradicts the assumption, so we assume $T'(c_i)$ is a decryption query. If condition (C2) does not hold then we have

   $$\boldsymbol{q}_{X_i} = \sum_{j \leq i} \pi 1_j \cdot \boldsymbol{q}_{K1_j} + \pi 2_j \cdot \boldsymbol{q}_{K2_j} + \sum_{j < i} \rho_j \cdot \boldsymbol{q}_{X_j} + \sum_{j \leq i} \varsigma_j \cdot \boldsymbol{a}_j + \boldsymbol{\tau} \boldsymbol{M}$$

   Multiplying the equation to $(\boldsymbol{v}' - \boldsymbol{v})$ and applying $Y_i = Y'_i$ and $\boldsymbol{M}(\boldsymbol{v} - \boldsymbol{v}') = 0$ and canceling convergent queries, gives

   $$\boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{X_i} \cdot \boldsymbol{v} + \pi 1_i \boldsymbol{q}_{K1_i} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \pi 2_i \boldsymbol{q}_{K2_i} \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

Note, at most one of the $\pi 1$ and $\pi 2$, can be equal to zero. The left-hand side of the above equation is a random value and the right-hand side is a fixed value, so the adversary advantage again is at most $1/|\mathbb{F}|$ which is a contradiction.

4. $K_i \neq K'_i$ and $X_i \neq X'_i$ and $Y_i \neq Y'_i$.

   Because $K1_i \neq K'1_i$ or $K2_i \neq K'2_i$, condition (C1) holds. We show if $T'(c_i)$ is an encryption query then (C3) holds and if it is a decryption query then (C2) holds. In the first case, assume for contradiction that (C3) does not hold, implying Equation 4.6. Applying the assumption $\boldsymbol{M}(\boldsymbol{v} - \boldsymbol{v}') = 0$ and canceling all the queries before i^* gives,

$$\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v} + \alpha 1_i \boldsymbol{q}_{K_i} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + + \alpha 1_i \boldsymbol{q}_{K_i} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \beta_i \boldsymbol{q}_{X_i} \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

   Here when the adversary is making query $T'(c_i)$, all the values on the right-hand side of the equation are fixed and so the adversary's advantage is $\leq 1/|\mathbb{F}|$, contrary to assumption. The case that $T'(c_i)$ is a decryption query and (C2) holds is similar.

5. $K_i \neq K'_i$ and $X_i = X'_i$ and $Y_i = Y'_i$.

   The probability of this case occurring is $\leq 1/|\mathbb{F}|$.

For $i > i^*$ by way of contradiction, assume (C2) and (C3) both fail:

$$\boldsymbol{q}_{X_i} = \sum_{j \leq i} \pi 1_j \cdot \boldsymbol{q}_{K1_j} + \sum_{j \leq i} \pi 2_j \cdot \boldsymbol{q}_{K2_j} + \sum_{j < i} \rho_j \cdot \boldsymbol{q}_{X_j} + \sum_{j \leq i} \varsigma_j \cdot \boldsymbol{a}_j + \boldsymbol{\tau M}$$

$$\boldsymbol{a}_i = \sum_{j \leq i} \alpha 1_j \cdot \boldsymbol{q}_{K_j} + \sum_{j \leq i} \alpha 2_j \cdot \boldsymbol{q}_{K2_j} + \sum_{j \leq i} \beta_j \cdot \boldsymbol{q}_{X_j} + \sum_{j < i} \gamma_j \cdot \boldsymbol{a}_j + \boldsymbol{\delta M}$$

Multiplying both sides by $(\boldsymbol{v}' - \boldsymbol{v})$ and canceling the terms having index less than i^* and noting $\boldsymbol{M}(\boldsymbol{v} - \boldsymbol{v}') = 0$ we get,

$$\begin{aligned} \boldsymbol{q}_{X_i} \cdot \boldsymbol{v}' = \boldsymbol{q}_{X_i} \cdot \boldsymbol{v} + \sum_{i^* \leq j \leq i} \pi 1_j \boldsymbol{q}_{K1_j} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{i^* \leq j \leq i} \pi 2_j \boldsymbol{q}_{K2_j} \cdot (\boldsymbol{v}' - \boldsymbol{v}) \\ + \sum_{i^* \leq j < i} \rho_j \boldsymbol{q}_{X_j} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{i^* \leq j \leq i} \varsigma_j \boldsymbol{a}_j \cdot (\boldsymbol{v}' - \boldsymbol{v}) \end{aligned} \tag{4.7}$$

$$\begin{aligned}\boldsymbol{a}_i \cdot \boldsymbol{v}' = \boldsymbol{a}_i \cdot \boldsymbol{v} + \sum_{i^* \le j \le i} \alpha 1_j \boldsymbol{q}_{K1_j} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{i^* \le j \le i} \alpha 2_j \boldsymbol{q}_{K2_j} \cdot (\boldsymbol{v}' - \boldsymbol{v}) \\ + \sum_{i^* \le j \le i} \beta_j \boldsymbol{q}_{X_j} \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{i^* \le j < i} \gamma_j \boldsymbol{a}_j \cdot (\boldsymbol{v}' - \boldsymbol{v})\end{aligned} \tag{4.8}$$

Now, if the adversary is making query $T'(c_i)$ as an encryption query then in Equation 4.8 all the values on the right-hand side of the equation are fixed and the left-hand side is random so the adversary advantage is at most $1/|\mathbb{F}|$, and if it is a decryption query, Equation 4.7 will give the same contradiction. So at least one of the conditions (C2) or (C3) has to hold. □

These two lemmas demonstrate that the presence of a collision structure for a nondegenerate DBL^{2l} Linicrypt program $\mathcal{P}$ with n constraints is equivalent to the existence of either a collision adversary with a success probability exceeding $N^{2n}/|\mathbb{F}|$ or a 2nd-preimage adversary with a success probability exceeding $N^n/|\mathbb{F}|$.

Appendix A.2 contains the Octave code capable of verifying the collision resistance of a DBL^{2l} program $\mathcal{P}$. The input to the code consists of the program description represented as an 8×5 matrix: $[\boldsymbol{q}_{K1_1}, \boldsymbol{q}_{K2_1}, \boldsymbol{q}_{X_1}, \boldsymbol{q}_{K1_2}, \boldsymbol{q}_{K2_2}, \boldsymbol{q}_{X_2}, \boldsymbol{m}_1, \boldsymbol{m}_2]^\top$. We ran this code on the Tandem compression function [22] which is modeled in Linicrypt as follows:

Example 4.2.6. ***Tandem Compression Function in Linicrypt***

$$\begin{aligned}&\underline{\mathcal{P}^E(v_1, v_2, v_3):} \\ &\quad v_4 := E(t_1, v_2, v_3, v_1) \\ &\quad v_5 := E(t_2, v_3, v_4, v_2) \\ &\quad return\ (v_4 + v_1, v_5 + v_2)\end{aligned}$$

The corresponding matrix of this program is:

$$
\begin{array}{c} \\ \boldsymbol{q}_{K1_1} \\ \boldsymbol{q}_{K2_1} \\ \boldsymbol{q}_{X_1} \\ \boldsymbol{q}_{K1_2} \\ \boldsymbol{q}_{K2_2} \\ \boldsymbol{q}_{X_2} \\ \boldsymbol{m}_1 \\ \boldsymbol{m}_2 \end{array}
\begin{array}{c}
\begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\
\left(\begin{array}{ccccc}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1
\end{array}\right)
\end{array}
$$

Running the code from Section A.2 that verifies collision resistance for DBL^{2l} compression functions produces the following output:

```
C1 holds for c1
C2 holds c1
order changed
C1 holds for c1
C2 holds for c1
Collision-Resistant
```

This code examines all four possible collision structures in this program, namely $(c_1, c_2, 1)$, $(c_1, c_2, 2)$, $(c_2, c_1, 1)$, and $(c_2, c_1, 2)$ searching for any of these collision structures. It is evident that no collision structure exists for the Tandem compression function, confirming its collision resistance.

To implement code capable of generating all collision-resistant DBL^{2l} programs with two ideal cipher calls, we must iterate through all $2^{5\times 8}$ matrices and filter out programs without a collision structure. Even with optimizations and parallel computation, this process will still take a considerable amount of time. A solution to this challenge is to fix certain vectors and generate all other vectors that yield collision-resistant compression functions. As an illustrative example, we provide code in Section A.2 that gets six vectors as an input: $\boldsymbol{q}_{K1_1}$, $\boldsymbol{q}_{K2_1}$, $\boldsymbol{q}_{X_1}$, $\boldsymbol{q}_{K1_2}$, $\boldsymbol{q}_{K2_2}$, and $\boldsymbol{q}_{X_2}$. These vectors represent ideal cipher calls and generate all possible output vectors $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ resulting in the establishment of a collision-resistant DBL compression function.

Chapter 5

Conclusion and Future Work

The results of this thesis build upon the previous studies conducted by [8] and [25] that demonstrate the utility of Linicrypt for providing a uniform and automatable framework for security proofs in the random oracle model, using a language that is powerful enough to capture many well-known constructions. We extended the work of [25] by using Linicrypt to model notions of hash function security beyond standard collision resistance, in particular giving a characterization of preimage awareness for hash functions defined by Linicrypt programs using a random oracle. This is based on a simple and poly-time checkable algebraic condition which may be easily implemented. Our method supports automated verification and generation of PrA hash functions. This is in contrast to existing approaches to proving PrA [13] which are ad hoc and not obviously amenable to automation. Our approach to PrA provides a uniform and automatable method for constructing indifferentiable hash functions, following the methodology of [13].

We have also demonstrated the utility of the Linicrypt framework beyond the random oracle model by giving characterizations of collision resistance property for Linicrypt programs in the ideal cipher model and we have related our general approach to the previous characterization of the PGV compression function[4]. We showed that our characterization is equivalent to the well-known notion of group-1 for the PGV functions. A general notion of group-2 for arbitrary Linicrypt programs was also given and shown to include the group 2 PGV compression functions. Recognizing the vast potential within the Linicrypt framework, we have extended its capabilities to accommodate DBL hash functions and have provided a characterization of collision-resistant DBL hash functions. This expansion of the framework serves as a significant step towards a broader understanding of Linicrypt and its applications.

Overall, building on the existing work of [8, 25, 20], the results of this thesis demonstrate the utility of Linicrypt in providing a uniform and automatable framework for security proofs in the random oracle model and ideal cipher model, in a language that is powerful enough to capture many well-known constructions.

There are a number of possibilities for extending the work presented here. Some are relatively straightforward, such as adapting the characterization of preimage awareness to the ideal cipher model. As mentioned at the end of Section 4.2, there is more work to be done in order to automatically generate collision-resistant DBL^{2l} compression functions. We plan to investigate more efficient matrix algebra techniques for dealing with this problem, from both a theoretical and practical perspective.

In a more speculative direction, an interesting question is whether a Linicrypt-based characterization of indifferentiable hash functions is possible. Given the methodology for constructing indifferentiable hash functions using PrA compression functions, and the fact that Linicrypt is particularly suited to the fixed-input-length setting, this question is more theoretical in nature than the others considered in this thesis. The following is an outline of an approach for modeling indifferentiable compression functions within Linicrypt programs in the ideal cipher model:

1. Initially, it's important to note that the advantage the adversary holds over the simulator is that the adversary knows the program's input.

2. Adversarial Query to Random Oracle/Compression Function: The adversary initiates the process by making a query to the random oracle or compression function.

3. Computation of Program Values: The adversary uses the inputs of the program and calculates all program values, which can be expressed as linear combinations of the inputs.

4. Querying for Additional Program Values: To determine additional program values, the adversary must make encryption or decryption queries.

5. Identifying Inconsistencies: For the adversary to mount an effective attack, it must detect inconsistencies within the values calculated in the preceding steps. Achieving this typically requires making some specific queries within each program. (It is essential to understand that the critical step in characterizing

indifferentiability in Linicrypt is knowing the specific queries required in step 4 to identify inconsistencies.)

6. Determining Program Input: In the final step, the evaluation revolves around assessing, based on the adversary's queries, whether it is feasible to deduce the input to the program. This evaluation is equivalent to ascertaining whether the inputs fall within the span of those queries' inputs and outputs. If it is indeed feasible to determine the program's input, the simulator can find the input and then make a query to the random oracle and provide the adversary with consistent values.

One weakness of the Linicrypt approach is that it is limited to constructions that use only linear operations. A more ambitious goal would be to extend the analysis provided by Linicrypt beyond purely linear algebraic constructions. In particular, it would be very useful to consider using bit-string operations, especially truncation, and concatenation, which are used in many constructions such as the sponge construction which is the basis of Keccak/SHA-3. Here it might be useful to revisit [1], which considers equivalence properties of algebraic programs over $\mathbb{GF}(2^\lambda)$ which include bit-string operations but do not have access to random oracles, ideal ciphers or (as in the case of Keccak) random permutations.

Bibliography

[1] Gilles Barthe, Marion Daubignard, Bruce M. Kapron, Yassine Lakhnech, and Vincent Laporte. On the equality of probabilistic terms. In *LPAR 2010*, volume 6355 of *LNCS*, pages 46–63. Springer, 2010.

[2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO 1996*, volume 1109 of *LNCS*, pages 1–15. Springer, 1996.

[3] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS 1993*, pages 62–73. ACM, 1993.

[4] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In *CRYPTO 2002*, volume 2442 of *LNCS*, pages 320–335. Springer, 2002.

[5] John Black, Phillip Rogaway, Thomas Shrimpton, and Martijn Stam. An analysis of the blockcipher-based hash functions from PGV. *Journal of Cryptology*, 23(4):519–545, 2010.

[6] Jurjen N. Bos and David Chaum. Provably unforgeable signatures. In *CRYPTO 1992*, volume 740 of *LNCS*, pages 1–14. Springer, 1992.

[7] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.

[8] Brent Carmer and Mike Rosulek. Linicrypt: A model for practical cryptography. In *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 416–445. Springer, 2016.

[9] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In *CRYPTO 2005*, volume 3621 of *LNCS*, pages 430–448. Springer, 2005.

[10] Ivan Damgård. Collision free hash functions and public key signature schemes. In *EUROCRYPT 1987*, volume 304 of *LNCS*, pages 203–216. Springer, 1987.

[11] Ivan Bjerre Damgård. A design principle for hash functions. In *Conference on the Theory and Application of Cryptology*, pages 416–427. Springer, 1989.

[12] Yevgeniy Dodis, Krzysztof Pietrzak, and Prashant Puniya. A new mode of operation for block ciphers and length-preserving macs. In *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 198–219. Springer, 2008.

[13] Yevgeniy Dodis, Thomas Ristenpart, and Thomas Shrimpton. Salvaging Merkle-Damgard for practical applications. *IACR Cryptol. ePrint Arch.*, page 177, 2009. Full version of [14].

[14] Yevgeniy Dodis, Thomas Ristenpart, and Thomas Shrimpton. Salvaging Merkle-Damgård for practical applications. In *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 371–388. Springer, 2009.

[15] Shimon Even, Oded Goldreich, and Silvio Micali. On-line/off-line digital signatures. *J. Cryptol.*, 9(1):35–67, 1996.

[16] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[17] Praveen Gauravaram, Nasour Bagheri, and Lars R Knudsen. Building indifferentiable compression functions from the PGV compression functions. *Designs, Codes and Cryptography*, 78:547–581, 2016.

[18] Shoichi Hirose. Provably secure double-block-length hash functions in a black-box model. In *Information Security and Cryptology–ICISC 2004: 7th International Conference, Seoul, Korea, December 2-3, 2004, Revised Selected Papers 7*, pages 330–342. Springer, 2005.

[19] Shoichi Hirose. Some plausible constructions of double-block-length hash functions. In *International Workshop on Fast Software Encryption*, pages 210–225. Springer, 2006.

[20] Tommy Hollenberg, Mike Rosulek, and Lawrence Roy. A complete characterization of security for Linicrypt block cipher modes. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, pages 439–454. IEEE, 2022.

[21] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman and hall/CRC, 2007.

[22] Xucjia Lai and James L Massey. Hash functions based on block ciphers. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 55–70. Springer, 1992.

[23] Leslie Lamport. Constructing digital signatures from a one-way function. Technical Report CSL 98, SRI International, 1979.

[24] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *TCC 2004*, volume 2951 of *LNCS*, pages 21–39. Springer, 2004.

[25] Ian McQuoid, Trevor Swope, and Mike Rosulek. Characterizing collision and second-preimage resistance in Linicrypt. In *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 451–470. Springer, 2019.

[26] Bart Mennink. Optimal collision security in double block length hashing with single length key. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 526–543. Springer, 2012.

[27] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO 1987*, volume 293 of *LNCS*, pages 369–378. Springer, 1987.

[28] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[29] Ralph C. Merkle. One way hash functions and DES. In *CRYPTO 1989*, volume 435 of *LNCS*, pages 428–446. Springer, 1989.

[30] Carl H Meyer and Michael Schilling. Secure program load with manipulation detection code. In *Proc. Securicom*, volume 88, pages 111–130, 1988.

[31] Arno Mittelbach and Marc Fischlin. *The Theory of Hash Functions and Random Oracles - An Approach to Modern Cryptography.* Information Security and Cryptography. Springer, 2021.

[32] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: A synthetic approach. In *CRYPTO 1993*, volume 773 of *LNCS*, pages 368–378. Springer, 1993.

[33] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[34] Claude E Shannon. Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715, 1949.

[35] Thomas Shrimpton and Martijn Stam. Building a collision-resistant compression function from non-compressing primitives. In *ICALP 2008*, volume 5126, pages 643–654. Springer, 2008.

[36] Martijn Stam. Blockcipher-based hashing revisited. In Orr Dunkelman, editor, *FSE 2009*, volume 5665 of *LNCS*, pages 67–83. Springer, 2009.

Appendix A

Appendix

A.1 Additional Information

A.1.1 The Birthday Bound

Consider a hash function $H : \{0,1\}^* \to \{0,1\}^d$. Suppose an adversary chooses distinct $x_1, \ldots, x_q \in \{0,1\}^*$ and evaluates H at each input to see whether any two of them produce a collision. Clearly, if $q > 2^d$ then this works with certainty. However, a simple argument shows that, even when H is a random function, the adversary needs to make $O(2^{d/2})$ queries to obtain a collision with probability $\geq \frac{1}{2}$. A random $H : \{0,1\}^* \to \{0,1\}^d$ has the property that for any x, $H(x)$ is distributed uniformly at random on $\{0,1\}^d$. Also, if $x \neq x'$ then $H(x)$ and $H(x')$ are independent. So now we only need to ask: if values $y_1, \ldots, y_q$ are chosen independently at random from $\{0,1\}^d$, what is the probability that for some $1 \leq i < j \leq q$, $y_i = y_j$? More generally,

$$\mathsf{nocoll}(q, N) = \Pr[\text{no collision among } q \text{ independent variables from } \{1, \ldots, N\}]$$
$$\mathsf{coll}(q, N) = 1 - \mathsf{nocoll}(q, N)$$

An asymptotic bound on $\mathsf{nocoll}(q, N)$ follows, using the fact that for $0 \leq x \leq 1$, $(1-x) \leq e^{-x} \leq (1-\frac{x}{2})$:

$$\begin{aligned}
\mathsf{nocoll}(q, N) &= (1 - \frac{1}{N}) \cdot \cdots \cdot (1 - \frac{q-1}{N}) \\
&\leq e^{-\frac{1}{N}} \cdot \cdots \cdot e^{-\frac{q-1}{N}} \\
&= e^{-\frac{1}{N}\sum_{j=1}^{q-1} j} \\
&= e^{-\frac{q(q-1)}{2N}} \\
&\leq 1 - \frac{q(q-1)}{4N}
\end{aligned}$$

So $\mathsf{coll}(q, N) \geq \frac{q(q-1)}{4N} = \Omega(\frac{q^2}{N})$. This means that for $N = 2^d = |\{0,1\}^d|$, if the adversary tries $O(2^{\frac{d}{2}})$ messages, a collision will occur with probability at least $\frac{1}{2}$

A.1.2 From PrA to Indifferentiability

Theorem A.1.1 ([13] Theorem 4.1)**.** *[RO domain extension via PrA] Let P be an ideal primitive and $H^{\P}$: DomRng be a hash function. Let R be an ideal primitive with two interfaces that implement independent functionalities P and $\mathcal{R} = \mathsf{RF}_{Rng,\ Rng}$. Define $F^R(M) = \mathcal{R}(H^P(M))$. Let $\mathcal{F} = \mathsf{RF}_{Dom,Rng}$. Let $\mathcal{E}$ be an arbitrary extractor for H. Then there exists a simulator $S = (S_1, S_2)$ such that for any PRO adversary A making at most (q_0, q_1, q_2) queries to its three oracle interfaces, there exists a PrA adversary B such that*

$$\boldsymbol{Adv}^{pro}_{F,R,S}(A) \leq \boldsymbol{Adv}^{pra}_{H,P,\mathcal{E}}(B)$$

Simulator S runs in time $\mathcal{O}(q_1 + q_2 \cdot Time(E))$. Let ℓ_{max} the length (in bits) of the longest query made by A to its first oracle. Adversary B runs in time that of A plus $\mathcal{O}(q_0 \cdot \mathsf{Time}(H, \ell_{max}) + q_1 + q_2)$, makes $q_1 + q_0 \cdot \mathsf{NumQueries}(H; \ell_{max})$ primitive queries, q_2 extraction queries, and outputs a preimage of length at most ℓ_{max}.

Theorem A.1.2 ([13] Theorem 4.2)**.** *[SMD is PrA-preserving] Fix $n, d > 0$ and let P be an ideal primitive. Let $h^P : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^n$. be a compression function, and let $H = SMD[h^P]$. Let $\mathcal{E}_h$ be an arbitrary extractor for the PrA experiment involving h. Then there exists an extractor $\mathcal{E}_h$ such that for all adversaries A making at most q_p primitive queries and q_e extraction queries and outputting a message of at*

most $\ell_{max} \geq 1$ *blocks there exists an adversary* B *such that*

$$\boldsymbol{Adv}^{pra}_{H,P,\mathcal{E}_h}(A) \leq \boldsymbol{Adv}^{pra}_{h,P,\mathcal{E}_h}(B)$$

$\mathcal{E}_h$ *runs in time at most* $\ell_{max}(\mathsf{Time}(\mathcal{E}_h) + \mathsf{Time}(unpad))$. *B runs in time at most that of A plus* $\mathcal{O}(q_e\ell_{max})$, *makes at most* $\ell_{max} \cdot \mathsf{NumQueries}(h, \ell_{max}) + q_p$ *ideal primitive queries, and makes at most* $q_e\ell_{max}$ *extraction queries.*

A.1.3 Essential Role of Nonces in the Ideal Cipher Model

Here is an example that illustrates how, without the use of nonces, adversaries can implement certain attacks that defy characterization within a collision structure.

Example A.1.3.

$$\begin{array}{l}\underline{\mathcal{P}^E(v_1, v_2, v_3):} \\ \quad v_4 := E(v_1, v_2) \\ \quad v_5 := E(v_1, v_3) \\ \quad return\ v_4 + v_5\end{array}$$

In this example, the adversary swaps the values of v_2 *and* v_3 *to discover a second preimage. However, if the two ideal cipher calls were independent, the adversary would be unable to execute this attack.*

A.2 Implementations

For each of the characterizations we have given, we also provided an efficient algorithm for determining whether a Linicrypt program has the corresponding "bad configuration" (critical query, collision structure.) The Octave code below implements these algorithms in programs used to generate programs that satisfy a given security property (in the case of PrA and collision-resistant rate-1 compression functions) or verify that a program satisfies the property (in the case of collision-resistant DBL^{2l} compression functions.) These programs demonstrate the utility of the Linicrypt framework for automated generation and verification of secure compression functions.

The following code generates all 530 rate $\frac{1}{2}$ PrA compression functions with two inputs.

```
clear;
clc;
file=fopen('PrA2Inputs2Calls.txt', 'w');
count =0; %count the number of PrA functions
N = 3;
M = 4;
v = ones(1,N*M);
a1=[0,0,1,0]; %first query answer
a2=[0,0,0,1]; %second query answer
e1=[1,0,0,0]; %first input
e2=[0,1,0,0]; %second input
A = reshape(v,[N,M]);
for i=1:2^(N*M)-1 %iterates over all possible 3*4 binary Matrices
    k = find(v,1);
    v(1:k) = 1-v(1:k);
    A = reshape(v,[N,M]);
    q1 = A(1,:); %first query
    q2 = A(2,:); %second query
    m = A(3,:);  %program output
    if and(q1(M-1)==0, q2(M)==0,or(q1(M)==0,q2(M-1)==0)) %no loops
   in queries
        if and(or(m(M)~=0,q1(M)~=0),or(m(M-1)~=0,q2(M-1)~=0))  %no
   useless query
            if and(rank([A;a1;a2])==rank([A;a1;a2;e1]),rank([A;a1;a2
   ])==rank([A;a1;a2;e2])) %nondegenerate
                if and(rank([A;a2])==rank([A;a2;a1]),rank([A;a1])==
   rank([A;a1;a2])) %no critical query
                    A
                    count=count+1;
                    fprintf(file,'\nA%i= \n',count)
                    fprintf(file,'%5i %i %i %i\n',A')

                endif
            endif

        endif
    endif
end
fclose(file);
```

```
clear;
clc;
```

```
file=fopen('PrAConstant2Inputs2Calls.txt', 'w');
N = 3;
M = 5;
count=0; %count the number of PrA functions
v = ones(1,N*M);
a1=[0,0,0,1,0]; %first query answer
a2=[0,0,0,0,1]; %second query answer
e1=[1,0,0,0,0]; %first input
e2=[0,1,0,0,0]; %second input
m2=[0,0,1,0,0]; %constant value v
A = reshape(v,[N,M]);
for i=1:2^(N*M)-1 %iterates over all possible 3*5 binary Matrices
    k = find(v,1);
    v(1:k) = 1-v(1:k);
    A = reshape(v,[N,M]);
    q1 = A(1,:); %first query
    q2 = A(2,:); %second query
    m1 = A(3,:); %program output
        if or(q1(M-2)==1,q2(M-2)==1,m1(M-2)==1 )  % v appears in the
    function
            if and(q1(M-1)==0,q2(M)==0,or(q1(M)==0,q2(M-1)==0)) %no
   loops in queries
                if and(or(m1(M)~=0,q1(M)~=0),or(m1(M-1)~=0,q2(M-1)
   ~=0))  %no useless query
                    if and(rank([A;a1;a2;m2])==rank([A;a1;a2;m2;e1])
   ,rank([A;a1;a2;m2])==rank([A;a1;a2;m2;e2])) %nondegenerate
                        if and(rank([A;a2;m2])==rank([A;a2;m2;a1]),
   rank([A;a1;m2])==rank([A;a1;m2;a2])) %no critical query
                            A
                            count=count+1;
                            fprintf(file,'\nA%i= \n',count)
                            fprintf(file,'%5i %i %i %i %i\n',A')
                        endif
                    endif
                endif
            endif
        endif
end
fclose(file);
```

The following code generates the 12 collision-resistant PGV compression functions.

```
clear
```

```
clc
file=fopen('Group1PGV.txt', 'w');
count =0; %counts the number of collision resistant functions
N = 3;
M = 4;
v = ones(1,N*M);
a=[0,0,0,1];
e1=[1,0,0,0];
e2=[0,1,0,0];
e3=[0,0,1,0];
m1=e3;
A = reshape(v,[N,M]);
for i=1:2^(N*M)-1 %iterates over all possible 3*4 binary Matrices
    k = find(v,1);
    v(1:k) = 1-v(1:k);
    A = reshape(v,[N,M]);
    m2 = A(3,:); %program output
    qk = A(1,:); %first component of the query (key)
    qx = A(2,:); %second component of the query (input)

    if and(or(qk(M-1)==0 ,and(qk(1)==0, qk(2)==0)),or(qx(M-1)==0 ,
   and(qx(1)==0, qx(2)==0)),or(m2(M-1)==0,and(m2(1)==0, m2(2)==0)))
   %E_a(b)+c where a,b,c \in {x,y,x+y,v}
        if and(qk(M)==0,qx(M)==0) %no loops in the query
            if m2(M)~=0 %no useless query
                if and(rank([A;a;m1])==rank([A;a;m1;e1]),rank([A;a;
   m1])==rank([A;a;m1;e2])) %nondegenerate
                    if   or(and( rank([m1;m2])==rank([m1;m2;qk]) ,
   rank([m1;m2;qk;a])==rank([m1;a;A])) , and( rank([m1;m2])==rank([
   m1;m2;qk]) ,  rank([m1;A])==rank([m1;A;a])) ,  and( rank([m1;A])
   ==rank([m1;A;a]) , rank([m1;m2;qk;a])==rank([m1;A;a])))%no
   collision structure
                        A
                        count=count+1;
                        fprintf(file,'\nA%i= \n',count)
                        fprintf(file,'%4i %i %i %i \n',A')

                    endif
                endif

            endif

```

```
        endif
    endif
end
fclose(file);
```

The following code verifies collision resistance for DBL^{2l} compression functions $f : \mathbb{F}^3 \rightarrow \mathbb{F}^2$ with two ideal cipher calls.

```
file_name = "matrix_data.csv";
% Use the load function to read the matrix from the file
A = dlmread(file_name, ',');
i=0;
j=0;
k=0;

a1=[0,0,0,1,0];
a2=[0,0,0,0,1];
e1=[1,0,0,0,0];
e2=[0,1,0,0,0];
e3=[0,0,1,0,0];

qk11 = A(1,:); %first component of the query (key1)
qk21 = A(2,:); %second component of the query (key2)
qx1  = A(3,:); %third component of the query (input)
qk12 = A(4,:);
qk22 = A(5,:);
qx2  = A(6,:);
m1   = A(7,:); %program output
m2   = A(8,:); %program output
%(C1,C2)
if or(rank([m1;m2;qk21])~=rank([m1;m2;qk21;qk11]),rank([m1;m2;qk11])
    ~=rank([m1;m2;qk21;qk11]))
  disp ("C1 holds for c1")
  i=i+1;
endif
if rank([m1;m2;qk11;qk21])~=rank([m1;m2;qk21;qk11;qx1])
    disp ("C2 holds c1")
    i=i+1;
endif
if rank([m1;m2;qk11;qk21;qx1])~=rank([m1;m2;qk21;qk11;qx1;a1])
  disp ("C3 holds for c1")
  i=i+1;
endif
```

```
35 if or(rank([m1;m2;qk11;qk21;qx1;a1;qk12])~=rank([m1;m2;qk21;qk11;qx1
       ;a1;qk12;qk22]), rank([m1;m2;qk11;qk21;qx1;a1;qk22])~=rank([m1;m2
       ;qk21;qk11;qx1;a1;qk12;qk22]))
36   disp ("C1 holds for c2")
37   k=k+1;
38 endif
39 if rank([m1;m2;qk11;qk21;qx1;a1;qk12;qk22])~=rank([m1;m2;qk21;qk11;
       qx1;a1;qk12;qk22;qx2])
40    disp ("C2 holds for c2")
41    j=j+1;
42    k=k+1;
43 endif
44 if rank([m1;m2;qk11;qk21;qx1;a1;qk12;qk22;qx2])~=rank([m1;m2;qk21;
       qk11;qx1;a1;qk12;qk22;qx2;a2])
45   disp ("C3 holds for c2")
46   j=j+1;
47   k=k+1;
48 endif
49 if or(and(i>=2,j>=1),(k>=2))
50     disp ("Not Collision-Resistant ")
51     return;
52 else
53     i=0;
54     j=0;
55     k=0;
56 end
57 %(c2,c1)
58 disp("order changed")
59 if or(rank([m1;m2;qk22])~=rank([m1;m2;qk22;qk12]), rank([m1;m2;qk12
       ])~=rank([m1;m2;qk12;qk22]))
60     disp ("C1 holds for c1")
61     i=i+1;
62 endif
63
64 if (rank([m1;m2;qk12;qk22])~=rank([m1;m2;qk12;qk22;qx2]))
65     disp ("C2 holds for c1")
66     i=i+1;
67 endif
68 if (rank([m1;m2;qk12;qk22;qx2])~=rank([m1;m2;qk12;qk22;qx2;a2]))
69     disp ("C3 holds for c1")
70     i=i+1;
71 endif
```

```
72
73 if or(rank([m1;m2;qk12;qk22;qx2;a2;qk21])~=rank([m1;m2;qk12;qk22;qx2
       ;a2;qk21;qk11]), rank([m1;m2;qk12;qk22;qx2;a2;qk11])~=rank([m1;m2
       ;qk12;qk22;qx2;a2;qk11;qk21]))
74   disp ("C1 holds for c2")
75   k=k+1;
76 endif
77 if rank([m1;m2;qk12;qk22;qx2;a2;qk21;qk11])~=rank([m1;m2;qk12;qk22;
       qx2;a2;qk21;qk11;qx1])
78    disp ("C2 holds for c2")
79    k=k+1;
80    j=j+1;
81 endif
82 if rank([m1;m2;qk12;qk22;qx2;a2;qk21;qk11;qx1])~=rank([m1;m2;qk12;
       qk22;qx2;a2;qk21;qk11;qx1;a1])
83   disp ("C3 holds for c2")
84   k=k+1;
85   j=j+1;
86 endif
87 if or(and(i>=2,j>=1),(k>=2))
88     disp ("Not Collision-Resistant ")
89     return;
90 else
91     disp('Collision-Resistant')
92 end
```

The following code generates all the collision-resistant DBL^{2l} compression functions with two ideal cipher calls with fixed inputs for the ideal cipher calls.

```
1  file = fopen('DBL.txt', 'w');
2  count = 0; % counts the number of collision-resistant functions
3
4  a1 = [0, 0, 0, 1, 0];
5  a2 = [0, 0, 0, 0, 1];
6  e1 = [1, 0, 0, 0, 0];
7  e2 = [0, 1, 0, 0, 0];
8  e3 = [0, 0, 1, 0, 0];
9
10 %fixed vectors:
11 qk11 = [0, 1, 0, 0, 0];
12 qk21 = [0, 0, 1, 0, 0];
13 qx1 = [1, 0, 0, 0, 0];
14 qk12 = [0, 0, 1, 0, 0];
```

```
qk22 = [0, 0, 0, 1, 0];
qx2 = [0, 1, 0, 0, 0];

N = 2;
M = 5;
v = ones(1, N * M);
A = reshape(v, [N, M]);
%max_iterations = 1000; % Set a maximum number of iterations

for i=1:2^(N*M)-1
    k = find(v, 1);
    v(1:k) = 1 - v(1:k);
    A = reshape(v, [N, M]);

    m1 = A(1, :);
    m2 = A(2, :);

    B = [qk11; qk21; qx1; qk12; qk22;qx2; A];

    if and(qk11(M - 1) == 0 , qk21(M - 1) == 0 , qx1(M - 1) == 0 , qk12(M) == 0 , qk22(M) == 0 , qx2(M) == 0 , any(m1) , any(m2))
        if and(or(m1(M - 1) ~= 0 , m2(M - 1) ~= 0) , or(m1(M) ~= 0 , m2(M) ~= 0))
            if and(rank([B; a1; a2]) == rank([B; a1; a2; e1]) , rank([B; a1; a2]) == rank([B; a1; a2; e2]) , rank([B; a1; a2]) == rank([B; a1; a2; e3])) %non-degenerate
                notc1 = and(rank([m1; m2; qk21]) == rank([m1; m2; qk21; qk11]) , rank([m1; m2; qk11]) == rank([m1; m2; qk11; qk21]));
                notc2 = (rank([m1; m2; qk11; qk21]) == rank([m1; m2; qk21; qk11; qx1]));
                notc3 = (rank([m1; m2; qx1; qk11; qk21]) == rank([m1; m2; qx1; qk11; qk21; a1]));

                notc21 = and(rank([m1; m2; qk21; qk11; qx1; a1; qk22]) == rank([m1; m2; qk21; qk11; qx1; a1; qk22; qk12]) , rank([m1; m2; qk21; qk11; qx1; a1; qk12]) == rank([m1; m2; qk21; qk11; qx1; a1; qk12; qk22]));
                notc22 = (rank([m1; m2; qk11; qk21; qx1; a1; qk12; qk22]) == rank([m1; m2; qk11; qk21; qx1; a1; qk12; qk22; qx2]));
                notc23 = (rank([m1; m2; qk11; qk21; qx1; a1; qk12; qk22; qx2]) == rank([m1; m2; qk11; qk21; qx1; a1; qk12; qk22; qx2
```

```
; a2]));

                notd1 = and(rank([m1; m2; qk22]) == rank([m1; m2;
qk22; qk12]) , rank([m1; m2; qk12]) == rank([m1; m2; qk12; qk22])
);
                notd2 = rank([m1; m2; qk12; qk22]) == rank([m1; m2;
qk22; qk12; qx2]);
                notd3 = rank([m1; m2; qx2; qk12; qk22]) == rank([m1;
 m2; qx2; qk12; qk22; a2]);

                notd21 = and(rank([m1; m2; qk22; qk12; qx2; a2; qk21
]) == rank([m1; m2; qk22; qk12; qx2; a2; qk21; qk11]) , rank([m1;
 m2; qk22; qk12; qx2; a2; qk11]) == rank([m1; m2; qk22; qk12; qx2
; a2; qk11; qk21]));
                notd22 = (rank([m1; m2; qk22; qk12; qx2; a2; qk11;
qk21]) == rank([m1; m2; qk22; qk12; qx2; a2; qk11; qk21; qx1]));
                notd23 = (rank([m1; m2; qk22; qk12; qx2; a2; qk11;
qk21; qx1]) == rank([m1; m2; qk22; qk12; qx2; a2; qk11; qk21; qx1
; a1]));

                P1 = or(and(notc1 , notc2) , and(notc1 , notc3) ,
and(notc2 , notc3));
                P2 = and(notc22 , notc23);
                P3 = or(and(notc21 , notc22) , and(notc21 , notc23)
, and(notc22 , notc23));

                P4 = or(and(notd1 , notd2) , and(notd1 , notd3) ,
and(notd2 , notd3));
                P5 = and(notc22 , notd23);
                P6 = or(and(notd21 , notd22) , and(notd21 , notd23)
, and(notd22 , notd23));

                if and(or(P1 , P2) , P3 , or(P4 , P5) , P6 )% no
collision structure
                    A;
                    count = count + 1;
                    fprintf(file, '\nA%i= \n', count);
                    fprintf(file, '%4i %i %i %i %i\n', A');
                endif
            endif
        endif
    endif
```

```
70 end
71
72 fclose(file);
```