Characterizing Collision and Second-Preimage Resistance in Linicrypt*

Ian McQuoid† Trevor Swope* Mike Rosulek*

July 24, 2020

Abstract

Linicrypt (Carmer & Rosulek, Crypto 2016) refers to the class of algorithms that make calls to a random oracle and otherwise manipulate values via fixed linear operations. We give a characterization of collision-resistance and second-preimage resistance for a significant class of Linicrypt programs (specifically, those that achieve domain separation on their random oracle queries via nonces). Our characterization implies that collision-resistance and second-preimage resistance are equivalent, in an asymptotic sense, for this class. Furthermore, there is a polynomial-time procedure for determining whether such a Linicrypt program is collision/second-preimage resistant.

1 Introduction

Collision resistance and second-preimage resistance are fundamental properties of hash functions, and are the basis of security for hash-based signature schemes [10, 11, 4, 7], which are a promising approach for post-quantum security.

We give a new way to reason about and characterize the collision resistance and second-preimage resistance of a large, natural class of programs, in the random oracle model. Specifically, we characterize these properties for the class of **Linicrypt programs**, introduced by Carmer & Rosulek [5]. Roughly speaking, a Linicrypt program is one where all intermediate values are field elements, and the only operations possible are fixed linear combinations, sampling uniformly from the field, and calling a random oracle (whose outputs are field elements). Many of the most practical cryptographic constructions are captured by this model: hash-based signatures and block cipher modes, to name a few.

Carmer & Rosulek showed that such programs admit an algebraic representations that is amenable to reasoning about programs' cryptographic properties. Specifically, they showed a polynomial-time algorithm for deciding whether two Linicrypt programs induce computationally indistinguishable distributions. They also demonstrated the feasibility of using a SAT solver to *automatically synthesize* Linicrypt programs that satisfy given correctness & security constraints, by successfully synthesizing secure Linicrypt constructions of garbled circuits.

Our work follows a similar path, showing that collision properties can also be characterized cleanly in terms of the algebraic representation for Linicrypt programs. Our characterization holds for programs in which distinct oracle queries have the form $H(t_1;\cdot), H(t_2;\cdot), \ldots$ for *distinct nonces* t_i.

*Authors partially supported by NSF award #1617197.

†Oregon State University, {mcquoidi,swopet,rosulekm}@oregonstate.edu. Authors partially supported by NSF award #1617197.

We introduce an algebraic property of Linicrypt programs called a *collision structure*, which completely characterizes both second-preimage resistance and collision resistance. The presence of a collision structure in a program $\mathcal{P}$ can be detected in polynomial time (in the size of $\mathcal{P}$'s algebraic representation).

Theorem 1 (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. (See main text for definitions)*

We emphasize that the theorem statement refers to **standard security properties** (*i.e.*, security against arbitrary, computationally unbounded algorithms that make only a polynomial number of queries to the random oracle) of Linicrypt constructions. We are **not** in a heuristic model that considers Linicrypt *adversaries.*

Our results show that second-preimage resistance and collision resistance are equivalent, *in an asymptotic sense* (i.e., considering only whether a quantity is negligible or not). However, as might be expected, it is quadratically easier to find collisions than second preimages, due to birthday attacks. Our concrete bounds reflect this. In practice, reducing security to second-preimage resistance rather than collision resistance can result in constructions with 50% smaller parameters; e.g., [6, 8, 2].

Addendum July 2020: We thank Catherine Meadows for pointing out a bug in how we defined the notion of a "degenerate" program (Definition 4). We have repaired the definition in this updated version. The fix did not involve changing any theorem statements.

1.1 Related Work & Comparison

Bellare & Micciancio [1] discuss the collision resistance of the function $H^*(x_1, \ldots, x_n) = H(1; x_1) \oplus \cdots \oplus H(n; x_n)$, where H is collision-resistant. Indeed, this function is naturally modeled in Linicrypt over a field $GF(2^\lambda)$. They show that this function fails to be collision-resistant if n is allowed to vary with the input (in particular, when $n \geq \lambda + 1$). Our characterization shows that an adversary making q oracle queries breaks collision resistance with probability bounded by $(q/n)^{2n}/2^\lambda$ since the function lacks a "collision structure." These two results are not in conflict, since our bound is meaningless when $n \geq \lambda + 1$. In short, the Linicrypt model is best suited for programs whose only dependence on the security parameter is the choice of field, but where (in particular) the number of inputs and calls to H are fixed constants.

Another related work is that of Wagner [13], who gives an algorithm for a generalized birthday problem. The problem (translated to our notation) is to find $x_1, \ldots, x_k$ such that $H(x_1) \oplus \cdots \oplus H(x_k) = 0$. The case of $k = 2$ corresponds to the well-known birthday problem. One can see that by generating a list L_i of roughly $2^{\lambda/k}$ candidates for each x_i (i.e., so $|L_1 \times \cdots \times L_k| \geq 2^\lambda$),

there is likely to exist some solution to the problem. Wagner's focus is on the *algorithmic* aspect of actually identifying the appropriate candidates. In Linicrypt, all adversaries are considered to be computationally unbounded but bounded in the number of queries to the random oracle H. As such, our results do not provide any upper/lower bounds on attack complexity (other than in random oracle query complexity).

Black, Rogaway, and Shrimpton [3] categorize 64 ways to construct a compression function (suitable for Merkle-Damgård hashing) from an ideal cipher, building on prior work by Preneel, Govaerts, and Vandewalle [12]. These constructions can be thought of as $GF(2^\lambda)$-Linicrypt programs that use only XOR (e.g., linear combinations with coefficients of 0 or 1 only). However, the reasoning is tied to the ideal cipher model rather than the random oracle model, as in Linicrypt (see Section 5.3 for more information). We leave it as interesting future work to extend results in Linicrypt to the ideal cipher model, and potentially re-derive the characterization of BRS from a linear-algebraic perspective.

2 Preliminaries

We write scalar field elements as lowercase non-bold letters (e.g., $v \in \mathbb{F}$). We write vectors as lowercase bold letters (e.g., $\boldsymbol{q} \in \mathbb{F}^n$). We write matrices as uppercase bold letters (e.g., $\boldsymbol{M} \in \mathbb{F}^{n \times m}$). We write vector inner product as $\boldsymbol{q} \cdot \boldsymbol{v}$, and matrix-vector multiplication as $\boldsymbol{M} \times \boldsymbol{v}$ or $\boldsymbol{M}\boldsymbol{v}$.

2.1 Linicrypt

The Linicrypt model was introduced in [5]. We present a brief summary of the model and its important properties.

A Linicrypt program (over field $\mathbb{F}$) is one in which every intermediate value is an element of $\mathbb{F}$, and the program is a fixed, straight-line sequence of the following kinds of operations:

- Call a random oracle (whose inputs/outputs are field elements).
- Sample a random field element.
- Combinine existing values using a fixed linear combination.

The sequence of operations (including choice of arguments to the oracle, coefficients of linear combinations, etc) is entirely fixed. In particular, these cannot depend on intermediate values in the computation.

The only source of cryptographic power in Linicrypt is the random oracle, whose outputs are $\mathbb{F}$-elements. We therefore require the size of the field $|\mathbb{F}|$ to be exponential in the security parameter λ. Since the field depends on the security parameter, we sometimes write $\mathbb{F} = \mathbb{F}_\lambda$ to make the association explicit.

If the field depends on the security parameter, then the program does too (since it is parameterized by specific coefficients of linear combinations). One can either consider a Linicrypt program to be a non-uniform family of programs (one for each choice of field / security parameter), or one can fix all coefficients in the program from $\widetilde{\mathbb{F}}$ which is a subfield of every $\mathbb{F}_\lambda$ (for example, a program that uses only $\{0, 1\}$ coefficients can be instantiated over any field $GF(2^\lambda)$). Our treatment of security is concrete (not asymptotic), so these distinctions are not important in this work.

We can reason about Linicrypt programs in the following *algebraic* way. Let $\mathcal{P}$ be such a program, and let $v_1, \ldots, v_n$ denote all of its intermediate variables. Say the first k of them are $\mathcal{P}$'s input and the last l of them are $\mathcal{P}$'s output. We say that v_i is a **base variable** if v_i is either

an input variable, the result of a call to the oracle, or the result of sampling a field element. All variables can therefore be expressed as a fixed linear combination of base variables.

Let $\boldsymbol{v}_{\mathsf{base}}$ denote the vector of all base variables. For each variable v_i, let $\boldsymbol{r}_i$ denote the vector such that $v_i = \boldsymbol{r}_i \cdot \boldsymbol{v}_{\mathsf{base}}$. For example, for base variables, $\boldsymbol{r}_i$ is a canonical basis vector (0s everywhere except 1 in one component).

Suppose the output of $\mathcal{P}$ consists of $v_{n-l+1}, \ldots, v_n$. Then the **output matrix** of $\mathcal{P}$ is defined as: $\boldsymbol{M} \stackrel{\text{def}}{=} \begin{bmatrix} \boldsymbol{r}_{n-l+1} \\ \vdots \\ \boldsymbol{r}_n \end{bmatrix}$. This matrix captures the fact that $\mathcal{P}$'s output can be expressed as $\boldsymbol{M} \times \boldsymbol{v}_{\mathsf{base}}$.

Each oracle query in $\mathcal{P}$ is of the form "$v_i := H(t; v_{i_1}, \ldots, v_{i_m})$," where t is a string (e.g., *nonce*) and $i_1, \ldots, i_m < i$ are indices, all *fixed* as part of $\mathcal{P}$. For each such query we define an associated **oracle constraint** $c = \left(t, \begin{bmatrix} \boldsymbol{r}_{i_1} \\ \vdots \\ \boldsymbol{r}_{i_m} \end{bmatrix}, \boldsymbol{r}_i \right)$. In other words, an oracle constraint $(t, \boldsymbol{Q}, \boldsymbol{a})$ captures the fact that if the oracle is queried as $H(t; \boldsymbol{Q} \times \boldsymbol{v}_{\mathsf{base}})$, then the response is $\boldsymbol{a} \cdot \boldsymbol{v}_{\mathsf{base}}$. When t is the empty string, we often omit it from our notation and simply write $H(\cdot)$ instead of $H(\epsilon; \cdot)$.

The **algebraic representation** of $\mathcal{P}$ is $\mathcal{P} = (\boldsymbol{M}, \mathcal{C})$, where $\boldsymbol{M}$ is the output matrix of $\mathcal{P}$ and $\mathcal{C}$ is the set of all oracle constraints. Indeed, these two pieces of information completely characterize the behavior of $\mathcal{P}$ (as established in [5]).

Example. In this work we focus on deterministic Linicrypt programs. One such example is given below. Its base variables are $(v_1, \ldots, v_5, v_7)$.

$$\boxed{\begin{array}{l} \underline{\mathcal{P}^H(v_1, v_2, v_3)\text{:}} \\ \quad v_4 := H(\texttt{foo}; v_1) \\ \quad v_5 := H(\texttt{bar}; v_3) \\ \quad v_6 := v_4 + v_5 + v_2 \\ \quad v_7 := H(\texttt{foo}; v_6) \\ \quad v_8 := v_7 + v_1 \\ \quad \text{return } (v_8, v_5) \end{array}} \quad \Rightarrow \quad \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_7 \end{bmatrix}$$

Hence, the algebraic representation of $\mathcal{P}$ is:

$$\boldsymbol{M} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}; \qquad \mathcal{C} = \left\{ \begin{array}{l} \Big(\texttt{foo}, [1\ 0\ 0\ 0\ 0\ 0], [0\ 0\ 0\ 1\ 0\ 0]\Big), \\ \Big(\texttt{bar}, [0\ 0\ 1\ 0\ 0\ 0], [0\ 0\ 0\ 0\ 1\ 0]\Big), \\ \Big(\texttt{foo}, [0\ 1\ 0\ 1\ 1\ 0], [0\ 0\ 0\ 0\ 0\ 1]\Big) \end{array} \right\}$$

2.2 Security Definitions

The Linicrypt model is meant to capture a special class of *construction*, but not adversaries. In this work we characterize **standard security definitions**, against arbitrary (i.e., not necessarily Linicrypt) adversaries. As in Impagliazzo's "Minicrypt" [9] we consider computationally unbounded adversaries that are *bounded-query:* they make only at most $p(\lambda)$ queries to the random oracle, for some polynomial p.

Definition 2. *Let $\mathcal{P}$ be a Linicrypt program over a family of fields $\mathbb{F} = (\mathbb{F}_\lambda)_\lambda$. Then $\mathcal{P}$ is (q, ϵ)-**collision-resistant (in the random oracle model)** if for all q-query adversaries $\mathcal{A}$, $\Pr[\mathsf{ColGame}(\mathcal{P}, \mathcal{A}, \lambda) = 1] \leq \epsilon$, where:*

$\underline{\mathsf{ColGame}(\mathcal{P}, \mathcal{A}, \lambda)}$:
instantiate a random oracle $H : \{0,1\}^* \times (\mathbb{F}_\lambda)^* \to \mathbb{F}_\lambda$
$(\boldsymbol{x}, \boldsymbol{x}') \leftarrow \mathcal{A}^H(\lambda)$
return $(\boldsymbol{x} \neq \boldsymbol{x}') \wedge (\mathcal{P}^H(\boldsymbol{x}) = \mathcal{P}^H(\boldsymbol{x}'))$

Definition 3. *Let $\mathcal{P}$ be as above (with k inputs). $\mathcal{P}$ is (q, ϵ)-**2nd-preimage-resistant (in the random oracle model)** if for all q-query adversaries $\mathcal{A}$, $\Pr[\mathsf{2PIGame}(\mathcal{P}, \mathcal{A}, \lambda) = 1] \leq \epsilon$, where:*

$\underline{\mathsf{2PIGame}(\mathcal{P}, \mathcal{A}, \lambda)}$:
instantiate a random oracle $H : \{0,1\}^* \times (\mathbb{F}_\lambda)^* \to \mathbb{F}_\lambda$
$\boldsymbol{x} \leftarrow (\mathbb{F}_\lambda)^k$
$\boldsymbol{x}' \leftarrow \mathcal{A}^H(\lambda, \boldsymbol{x})$
return $(\boldsymbol{x} \neq \boldsymbol{x}') \wedge (\mathcal{P}^H(\boldsymbol{x}) = \mathcal{P}^H(\boldsymbol{x}'))$

3 Characterizing Collision-Resistance in Linicrypt

We now present our main technical result, which is a characterization of collision-resistance for Linicrypt programs.

In order to simplify the notation, we present the results for the **special case** of Linicrypt programs that make 1-ary calls to H. That is, every call to H is of the form $H(t; v)$ for a *single* $v \in \mathbb{F}$ (note that Linicrypt supports more general calls of the form $H(t; v_1, \ldots, v_k)$). With this simplification, every oracle constraint has the form $(t, \boldsymbol{q}, \boldsymbol{a})$ where $\boldsymbol{q}$ is a simple vector (rather than a matrix as in the most general form).

This special case simplifies the notation required to express our theorems/proofs, but does not gloss over any meaningful complexity. Later in Section 5.1 we discuss what minor changes are necessary to extend these results to the unrestricted general case.

3.1 Easy Case: Degeneracy

Some Linicrypt programs allow easy collisions. Consider the program $\mathcal{P}^H(x, y) = H(x + y)$. An obvious collision in $\mathcal{P}$ is $\mathcal{P}^H(x, y) = \mathcal{P}^H(x + c, y - c)$ for any $c \neq 0$. This program does not use its inputs x or y individually, but depends only on $x + y$. Put differently, the program can be written as the composition of a *lossy linear function* $(x, y \mapsto x + y)$ and another Linicrypt program $(z \mapsto H(z))$. No program of this form can be collision-resistant, since it suffices to simply find a collision in the lossy linear function.

Definition 4. *Let $\mathcal{P} = (\boldsymbol{M}, \mathcal{C})$ be a deterministic Linicrypt program with n base variables. In the algebraic representation, $\mathcal{P}$'s variables are associated with canonical basis vectors $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n$ ($\boldsymbol{r}_i$ has 0s everywhere except a 1 in the ith component). We say that $\mathcal{P}$ is **degenerate** if*

$$\mathsf{span}(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n) \not\subseteq \mathsf{span}\Big(\{\boldsymbol{q} \mid (t, \boldsymbol{q}, \boldsymbol{a}) \in \mathcal{C}\} \cup \{\boldsymbol{a} \mid (t, \boldsymbol{q}, \boldsymbol{a}) \in \mathcal{C}\} \cup \mathsf{rows}(\boldsymbol{M})\Big)$$

Lemma 5. *If $\mathcal{P}$ is degenerate, then second preimages can be found with probability 1.*

Proof. The right-hand-side of the expression in Definition 4 considers all vectors in $\mathcal{P}$'s algebraic representation. These vectors are generally expressed in terms of the basis $\mathcal{B} = \{\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n\} = \{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_k\} \cup \{\boldsymbol{a} \mid (t, \boldsymbol{q}, \boldsymbol{a}) \in \mathcal{C}\}$, representing the base variables in $\mathcal{P}$ with the $\boldsymbol{e}_i$'s being the input of $\mathcal{P}$ in canonical-basis form. If $\mathcal{P}$ is degenerate, then the vectors in its algebraic representation do not actually span this space.

Consider the matrix $A_{\mathcal{P}}$, for some Linicrypt program $\mathcal{P}$, whose rows are comprised of each vector in the set $\{\boldsymbol{q} \mid (t, \boldsymbol{q}, \boldsymbol{a}) \in \mathcal{C}\} \cup \{\boldsymbol{a} \mid (t, \boldsymbol{q}, \boldsymbol{a}) \in \mathcal{C}\} \cup \mathsf{rows}(\boldsymbol{M})$. We can consider $A_{\mathcal{P}} \times \boldsymbol{v}_{\mathsf{base}}$ for $\mathcal{P}(\boldsymbol{x})$ on some given input $\boldsymbol{x} = \begin{bmatrix} \boldsymbol{e}_1 \\ \boldsymbol{e}_2 \\ \vdots \\ \boldsymbol{e}_k \end{bmatrix} \cdot \boldsymbol{v}_{\mathsf{base}}$. $A_{\mathcal{P}} \times \boldsymbol{v}_{\mathsf{base}}$ then determines concrete values for queries, answers, and outputs in $\mathcal{P}$. If two distinct inputs $\boldsymbol{x}, \boldsymbol{x}'$ exist such that $A_{\mathcal{P}} \times \boldsymbol{v}_{\mathsf{base}} = A_{\mathcal{P}} \times \boldsymbol{v}'_{\mathsf{base}}$ then certainly $\boldsymbol{x}'$ is a preimage of $\mathcal{P}(\boldsymbol{x})$ as they both produce the same output — indeed, $P(\boldsymbol{x}')$ even makes the exact same queries to H! These instances arise exactly when $A_{\mathcal{P}} \times \boldsymbol{v}_{\mathsf{base}} - A_{\mathcal{P}} \times \boldsymbol{v}'_{\mathsf{base}} = A_{\mathcal{P}} \times (\boldsymbol{v}_{\mathsf{base}} - \boldsymbol{v}'_{\mathsf{base}}) = 0$ which occurs when the kernel of $A_{\mathcal{P}}$ has dimension greater than 0.

If $\mathcal{P}$ is degenerate, then the span of the rows of $A_{\mathcal{P}}$ does not contain all the basis vectors: $\mathcal{B} \not\subseteq \mathsf{span}(\mathsf{rows}(A_{\mathcal{P}}))$. In this case, the matrix $A_{\mathcal{P}}$ has a non-trivial kernel. □

To find a concrete set of preimages for a degenerate linicrypt program $\mathcal{P}(\boldsymbol{x})$ we can consider the projection $B_{\mathcal{P}}$ of $\ker(A_{\mathcal{P}})$ onto $A_{\mathcal{P}}$'s input space. The projection can be constructed by embedding the $k \times k$ identity matrix $I_{k \times k}$ into the $k \times n$ 0 matrix (where k is the number of input vectors to $\mathcal{P}$ and n is the number of base vectors of $\mathcal{P}$).

$$B_{\mathcal{P}} = \begin{bmatrix} I_{k \times k} & 0 \end{bmatrix}$$

Then we may find a class of preimages $\texttt{degenerate}(\mathcal{P}) = B_{\mathcal{P}} \cdot \ker(A_{\mathcal{P}})$. For each vector $\delta \in \texttt{degenerate}(\mathcal{P})$ we have that $\mathcal{P}(\boldsymbol{x}) = \mathcal{P}(\boldsymbol{x} + \boldsymbol{\delta})$.

Applying this idea to the previous example $\mathcal{P}^H(x, y) = H(x + y)$, we arrive at the matrices $A_{\mathcal{P}} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, B_{\mathcal{P}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ with $B_{\mathcal{P}} \cdot \ker(A_{\mathcal{P}}) = \left\{ c \cdot \begin{bmatrix} 1 \\ -1 \end{bmatrix} : c \in \mathbb{F} \right\}$ as we would expect.

3.2 Running Example: An Interesting Second-Preimage Attack

Consider the example program below. In fact, it is the example from Section 2.1 but with the nonces omitted and most intermediate variables unnamed:

$\mathcal{P}^H(x, y, z)$:
  $w := H(x) + H(z) + y$
  return $\big(H(w) + x,\ H(z)\big)$

Suppose we are given x, y, z and are asked to find a second preimage x', y', z' with $\mathcal{P}^H(x, y, z) = \mathcal{P}^H(x', y', z')$. Here is how to do it:

1. The second component of $\mathcal{P}$'s output is $H(z)$. Since we cannot hope to find a second preimage directly in H, we must set $z' = z$.

2. The key insight is to now set $w' \neq w$ arbitrarily (hence, why we gave this value a name). We make a promise to choose x', y' so that $w' = H(x') + H(z') + y'$.

3. To have a collision, we must have $H(w') + x' = H(w) + x$. Importantly, x' is the only unknown value in this expression, and it is possible to simply solve for x'.

4. It is time to fulfill the promise that $w' = H(x') + H(z') + y'$. Since w', x', z' are already fixed, we can solve for y'.

Note that we are guaranteed that $(x, y, z) \neq (x', y', z')$ since the two computations of $\mathcal{P}$ lead to different intermediate values $w \neq w'$ (and $\mathcal{P}$ is deterministic).

Perspective. This example is representative of how second preimages can be computed in arbitrary Linicrypt programs. Given an input $\boldsymbol{x}$ for $\mathcal{P}^H$, we compute a second preimage $\boldsymbol{x'}$ by focusing on the oracle queries that $\mathcal{P}^H(\boldsymbol{x})$ and $\mathcal{P}^H(\boldsymbol{x'})$ will make:

1. Designate some of the oracle queries to take the same values in both $\mathcal{P}^H(\boldsymbol{x})$ and $\mathcal{P}^H(\boldsymbol{x'})$. In our example, we decided that the oracle query $H(z)$ would take the same values in both computations.

2. Identify the first query that we will assign different values in the two computations. Set the input to this query arbitrarily in $\mathcal{P}^H(\boldsymbol{x'})$. In our example, we identify the $H(w)$ query to take on different values and set $w' \neq w$ arbitrarily.

3. Repeatedly make followup oracle queries as they become possible, while using linear algebra to solve for other intermediate values. In our example, we call $H(w')$, which allows us to solve for x', which allows us to call $H(x')$, which allows us to solve for y'.

3.3 Collision Structures for Finding Second Preimages

We have given a rough outline of how (we claim) Linicrypt second preimages must be found. The next step is to formalize what is required of $\mathcal{P}$ in terms of its algebraic representation.

In step 2 above, we identify a query whose input will be chosen arbitrarily. Suppose that query corresponds to constraint $(t, \boldsymbol{q}, \boldsymbol{a})$. Since this is the first value that is fixed differently in $\mathcal{P}^H(\boldsymbol{x})$ and $\mathcal{P}^H(\boldsymbol{x'})$, we must have $\boldsymbol{q}$ linearly independent of the vectors that are already fixed by step 1. Otherwise it would not be possible to find two consistent values for this query.

In steps 2 and 3 above, we repeatedly query H, and we have written the attack outline to suggest we never get "stuck." One way we could get stuck is to make some query $H(x')$ for the first time, when we have already fixed (either directly or indirectly) what $H(x')$ must be. If this is the case, then we cannot succeed with probability better than $1/|\mathbb{F}_\lambda|$. To avoid this case, every query we make in steps 2 & 3 of the outline must correspond to a constraint $(t, \boldsymbol{q}, \boldsymbol{a})$ where $\boldsymbol{a}$ is linearly independent of the values that have already been fixed.

The following definition formalizes these algebraic intuitions:

Definition 6. *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:*

1. *$c_1, \ldots, c_n$ is an ordering of $\mathcal{C}$, and we write $c_i = (t_i, \boldsymbol{q}_i, \boldsymbol{a}_i)$.*
2. *$\boldsymbol{q}_{i^*} \notin \mathsf{span}\Big(\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_{i^*-1}\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i^*-1}\} \cup \mathsf{rows}(\boldsymbol{M})\Big)$*
3. *For $j \geq i^*$: $\boldsymbol{a}_j \notin \mathsf{span}\Big(\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_j\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{j-1}\} \cup \mathsf{rows}(\boldsymbol{M})\Big)$*

Connecting to the previous intuition, a collision-finding attack will let oracle queries $c_1, \ldots, c_{i^*-1}$ be the same in both executions $\mathcal{P}^H(\boldsymbol{x})$ and $\mathcal{P}^H(\boldsymbol{x'})$. Then c_{i^*} is the first oracle query that the attack fixes differently for the two executions. Property (2) of the definition ensures that it is possible to

find 2 query values that are consistent with the previously fixed values. Property (3) captures the fact that from this point forward, no query should be forced to result in an output value that has already been fixed.

Running example. We now revisit the running example from before, to illustrate a collision structure for it. The base variables of this program are x, y, z, $H(x)$, $H(z)$, $H(w)$. Below is the algebraic representation of this program, with the oracle constraints arranged to show a collision structure (we do not write the empty nonces of the oracle constraints):

$$\begin{array}{rcccccc} & x & y & z & H(x) & H(z) & H(w) \\ \boldsymbol{M} = \left[\right. & 1 & 0 & 0 & 0 & 0 & 1 \\ & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline \boldsymbol{q}_1 = \left[\right. & 0 & 0 & 1 & 0 & 0 & 0 \\ \boldsymbol{a}_1 = \left[\right. & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline \boldsymbol{q}_{i^*} = \boldsymbol{q}_2 = \left[\right. & 0 & 1 & 0 & 1 & 1 & 0 \\ \boldsymbol{a}_{i^*} = \boldsymbol{a}_2 = \left[\right. & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline \boldsymbol{q}_3 = \left[\right. & 1 & 0 & 0 & 0 & 0 & 0 \\ \boldsymbol{a}_3 = \left[\right. & 0 & 0 & 0 & 1 & 0 & 0 \end{array} \begin{array}{l} \\ \\ \\ H(z) \\ \\ H(w) = H(y + H(x) + H(z)) \\ \\ H(x) \end{array}$$

This ordering of queries is indeed a collision structure since:

- $\boldsymbol{q}_2$ is linearly independent of all vectors above it in this diagram.
- $\boldsymbol{a}_2$ is linearly independent of all vectors above it in this diagram.
- $\boldsymbol{a}_3$ is linearly independent of all vectors above it in this diagram.

Second-preimage-finding algorithm. In Figure 1 we give an algorithm that finds second preimages by following the intuitive strategy above, from a given collision structure.

Lemma 7. *If a collision structure $(i^*; c_1, \ldots, c_n)$ exists for $\mathcal{P}$, and $\mathcal{P}$ is not degenerate, then the second-preimage resistance of $\mathcal{P}$ is comprehensively broken. Specifically, let $\mathcal{A}$ refer to* $\mathsf{FindSecondPreimage}(\mathcal{P}, (i^*; c_1, \ldots, c_n), \cdot)$. *Then:*

$$\Pr\left[\mathsf{2PIGame}(\mathcal{P}, \mathcal{A}, \lambda) = 1\right] = 1$$

Proof. Given $\boldsymbol{x}$, the goal is to compute a second preimage $\boldsymbol{x}'$. The computation of $\mathcal{P}^H(\boldsymbol{x}')$ has a certain set of base variables $\boldsymbol{v}'$, and it suffices to compute those instead since $\boldsymbol{x}' = (\boldsymbol{e}_1 \cdot \boldsymbol{v}', \ldots, \boldsymbol{e}_k \cdot \boldsymbol{v}')$. The attack FindSecondPreimage fixes one linear constraint of $\boldsymbol{v}'$ at a time, until $\boldsymbol{v}'$ is completely determined.

It suffices to show the following about the behavior of FindSecondPreimage:

1. It computes a different set of base variables $\boldsymbol{v}'$ than those of $\mathcal{P}^H(\boldsymbol{x})$.
2. It never adds incompatible (unsatisfiable) linear constraints on $\boldsymbol{v}'$.
3. Values $\boldsymbol{v}'$ are consistent with H. Namely, if $(t, \boldsymbol{q}, \boldsymbol{a}) \in \mathcal{C}$, then $H(t; \boldsymbol{q} \cdot \boldsymbol{v}') = \boldsymbol{a} \cdot \boldsymbol{v}'$.
4. By the end of the computation, enough constraints have been added to completely determine $\boldsymbol{v}'$.

Property 1 holds since $\boldsymbol{q}_{i^*} \cdot \boldsymbol{v} \neq \boldsymbol{q}_{i^*} \cdot \boldsymbol{v}'$ by design. Regarding property 2:

$\mathsf{FindSecondPreimage}\Big(\mathcal{P} = (\boldsymbol{M}, \mathcal{C}), (i^*; c_1, \ldots, c_n), \boldsymbol{x}\Big)$:

compute $\boldsymbol{v}$, the set of base variables in computation $\mathcal{P}^H(\boldsymbol{x})$
initialize an empty set of linear constraints on unknowns $\boldsymbol{v}'$

add constraint $\boldsymbol{M}\boldsymbol{v} = \boldsymbol{M}\boldsymbol{v}'$
for $i = 1$ to $i^* - 1$:
  add constraints $\boldsymbol{q}_i \cdot \boldsymbol{v} = \boldsymbol{q}_i \cdot \boldsymbol{v}'$ and $\boldsymbol{a}_i \cdot \boldsymbol{v} = \boldsymbol{a}_i \cdot \boldsymbol{v}'$

choose a value $v^* \in \mathbb{F}_\lambda$ arbitrarily, with $v^* \neq \boldsymbol{q}_{i^*} \cdot \boldsymbol{v}$
add constraint $v^* = \boldsymbol{q}_{i^*} \cdot \boldsymbol{v}'$

for $i = i^*$ to n:
  if $\boldsymbol{q}_i \cdot \boldsymbol{v}'$ is not already uniquely determined by current constraints:
    choose $r \in \mathbb{F}_\lambda$ arbitrarily and add constraint $r = \boldsymbol{q}_i \cdot \boldsymbol{v}'$
  call $s := H(t_i, \boldsymbol{q}_i \cdot \boldsymbol{v}')$ // $\boldsymbol{q}_i \cdot \boldsymbol{v}'$ *guaranteed to be uniquely determined here*
  add constraint $s = \boldsymbol{a}_i \cdot \boldsymbol{v}'$

return $(\boldsymbol{e}_1 \cdot \boldsymbol{v}', \ldots, \boldsymbol{e}_k \cdot \boldsymbol{v}')$

Figure 1: Method for computing second preimages

- The constraints on $\boldsymbol{v}'$ that are added for $\boldsymbol{M}$ and in the first for-loop are self-consistent — by construction they already have a valid solution in $\boldsymbol{v}$.
- The constraint involving $\boldsymbol{q}_{i^*}$ is compatible with the previous constraints since $\boldsymbol{q}_{i^*}$ is linearly independent of the previous constraint vectors $\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_{i^*-1}\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i^*-1}\} \cup \mathsf{rows}(\boldsymbol{M})$, by the collision structure property.
- Similarly, a constraint involving $\boldsymbol{q}_i$ for $i \geq i^*$ (if-statement within last for-loop) is only added in the case that $\boldsymbol{q}_i$ is linearly independent of the previous constraint vectors.
- The constraint involving $\boldsymbol{a}_i$ in the second for-loop is consistent since $\boldsymbol{a}_i$ is linearly independent of existing constraint vectors, again by the collision structure property.

Regarding property 3: for oracle constraints c_i with $i < i^*$, consistency with H is ensured by agreeing with the existing values $\boldsymbol{v}$. For constraints c_i with $i \geq i^*$, consistency is guaranteed since the second for-loop actually calls H to determine the consistent way to constrain $\boldsymbol{a}_i \cdot \boldsymbol{v}'$.

Property 4 follows from the fact that $\mathcal{P}$ is not degenerate. We can see that $\boldsymbol{M} \times \boldsymbol{v}'$, $\boldsymbol{q} \cdot \boldsymbol{v}'$, and $\boldsymbol{a} \cdot \boldsymbol{v}'$ are fixed/determined by the end of the computation, for all $(t, \boldsymbol{q}, \boldsymbol{a}) \in \mathcal{C}$. Non-degeneracy implies that the input of $\mathcal{P}$ (and hence all base variables) is uniquely determined. □

3.4 Efficiently Finding Collision Structures

In this section we show that it is possible to efficiently determine whether a Linicrypt program has a collision structure, by analyzing its algebraic representation. The algorithm for finding a collision structure is given in Figure 2.

Lemma 8. *FindColStruct*$(\mathcal{P})$ *(Figure 2) outputs a collision structure for* $\mathcal{P}$ *if and only if one exists. Furthermore, the running time of* FindColStruct *is polynomial (in the size of* $\mathcal{P}$*'s algebraic representation).*

$\underline{\mathsf{FindColStruct}(\mathcal{P} = (\boldsymbol{M}, \mathcal{C})):}$
  $\mathsf{LEFT} := \mathcal{C}$
  $\mathsf{RIGHT} :=$ empty stack
  $V := \{\boldsymbol{q} \mid (t, \boldsymbol{q}, \boldsymbol{a}) \in \mathcal{C}\} \cup \{\boldsymbol{a} \mid (t, \boldsymbol{q}, \boldsymbol{a}) \in \mathcal{C}\} \cup \mathsf{rows}(\boldsymbol{M})$, as a **multi-set**

  // below: "$V \setminus \{\boldsymbol{a}\}$" means "$V$ with multiplicity of $\boldsymbol{a}$ reduced by 1"
  while $\exists (t, \boldsymbol{q}, \boldsymbol{a}) \in \mathsf{LEFT}$ such that $\boldsymbol{a} \notin \mathsf{span}(V \setminus \{\boldsymbol{a}\})$:
    remove $(t, \boldsymbol{q}, \boldsymbol{a})$ from LEFT
    push $(t, \boldsymbol{q}, \boldsymbol{a})$ to RIGHT
    reduce multiplicity of $\boldsymbol{q}, \boldsymbol{a}$ in V by 1

  while $\exists (t, \boldsymbol{q}, \boldsymbol{a}) \in \mathsf{RIGHT}$ such that $\boldsymbol{q} \in \mathsf{span}(V)$:
    remove $(t, \boldsymbol{q}, \boldsymbol{a})$ from RIGHT
    add $(t, \boldsymbol{q}, \boldsymbol{a})$ to LEFT
    increase multiplicity of $\boldsymbol{q}, \boldsymbol{a}$ in V by 1

  if RIGHT is nonempty:
    set $i^* := |\mathsf{LEFT}| + 1$
    write $\mathsf{LEFT} = (c_1, \ldots, c_{i^*-1})$, where order doesn't matter
    write $\mathsf{RIGHT} = (c_{i^*}, \ldots, c_n)$ in reverse order of insertion
    return $(i^*; c_1, \ldots, c_n)$
  else: return $\perp$

Figure 2: Method for finding collision structures in a Linicrypt program.

Proof. Some useful invariants in $\mathsf{FindColStruct}$ are that at any time, $\mathsf{LEFT} \cup \mathsf{RIGHT} = \mathcal{C}$ and V is a multiset of the vectors appearing in $\mathsf{rows}(\boldsymbol{M})$ and LEFT. Note that $\mathsf{FindColStruct}$ works in two phases: it starts with all oracle queries in LEFT and in the first phase moves some to RIGHT. In the second phase, it moves some of the oracle queries back into LEFT.

($\Rightarrow$) First, we argue that if $\mathsf{FindColStruct}(\mathcal{P}) = (i^*; c_1, \ldots, c_n) \neq \perp$, then this output is indeed a collision structure. Write each oracle constraint c_i as $c_i = (t_i, \boldsymbol{q}_i, \boldsymbol{a}_i)$.

- At the time the second while-loop terminates, we must have $\boldsymbol{q}_{i^*} \notin \mathsf{span}(V)$ since otherwise c_{i^*} would have been moved to LEFT. But $V = \{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_{i^*-1}\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i^*-1}\} \cup \mathsf{rows}(\boldsymbol{M})$, so this establishes one of the required properties of a collision structure.

- For $j \geq i^*$, consider the time at which c_j is about to be added to RIGHT in the first while-loop (i.e., the point that the while loop body is entered). At that point, $\mathsf{LEFT} = \{c_1, \ldots, c_j\}$, so V contains $\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_j\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_j\} \cup \mathsf{rows}(\boldsymbol{M})$. Since the while-loop condition is fulfilled, we have

$$\boldsymbol{a}_j \notin \mathsf{span}(V \setminus \{\boldsymbol{a}_j\}) = \mathsf{span}\Big(\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_j\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{j-1}\} \cup \mathsf{rows}(\boldsymbol{M})\Big)$$

  which is the other condition required for a collision structure.

($\Leftarrow$) For the other direction, suppose $(i^*, c_1, \ldots, c_n)$ is some collision structure for $\mathcal{P}$. We will show that the algorithm adds $c_{i^*}, \ldots, c_n$ to RIGHT in the first phase, but does *not* move c_{i^*} back to LEFT in the second phase. This implies that the algorithm terminates with $|\mathsf{RIGHT}| \neq \emptyset$, so by the previous reasoning it outputs some valid collision structure (perhaps different than the collision structure we are assuming exists).

The fact that $c_{i^*}, \ldots, c_n$ are added to RIGHT in the first phase is essentially the converse of what was shown above. For example, the collision structure property is that $\boldsymbol{a}_n \notin \mathsf{span}\Big(\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{n-1}\} \cup \mathsf{rows}(\boldsymbol{M})\Big)$, implying that c_n can trigger the while-loop and be added to RIGHT immediately. Note that even if other constraints are added to RIGHT in this phase, it only makes V smaller, so only causes the condition to check a *smaller* span than in the collision-property definition. A simple inductive argument establishes that $c_{i^*}, \ldots, c_n$ are eventually added to RIGHT.

Since $\{c_{i^*}, \ldots, c_n\} \subseteq \mathsf{RIGHT}$ after the first phase, we must have $\mathsf{LEFT} \subseteq \{c_1, \ldots, c_{i^*-1}\}$ after the first phase. We want to show that c_{i^*} is never placed back into LEFT. For the sake of contradiction, suppose not. Define S to be a set of indices such that $\mathsf{LEFT} = \{c_i \mid i \in S\}$ at the time c_{i^*} is about to be moved into LEFT. Then $\boldsymbol{q}_{i^*} \in \mathsf{span}(\mathsf{rows}(\boldsymbol{M}) \cup \{\boldsymbol{q}_i, \boldsymbol{a}_i \mid i \in S\})$. We can then write:

$$\boldsymbol{q}_{i^*} = \sum_{j \in S} \alpha_j \boldsymbol{q}_j + \sum_{j \in S} \beta_j \boldsymbol{a}_j + \boldsymbol{\gamma} \boldsymbol{M}$$

For $j > i^*$, the constraint c_j was previously in RIGHT and was moved back into LEFT. The only way to be moved back into LEFT is for $\boldsymbol{q}_j$ to be in the span of other vectors already in LEFT (and hence already on the right-hand side of this expression). Hence, without loss of generality we can remove the terms involving $\boldsymbol{q}_j$ for $j > i^*$, to obtain:

$$\boldsymbol{q}_{i^*} = \sum_{j \in S \setminus \{i^*, \ldots, n\}} \alpha'_j \boldsymbol{q}_j + \sum_{j \in S} \beta'_j \boldsymbol{a}_j + \boldsymbol{\gamma}' \boldsymbol{M}$$

Let j^* be the highest $j \in S$ for which $\beta'_j \neq 0$. There are two cases.

Case $j^* < i^*$: Then all of the nonzero terms $\boldsymbol{q}_j, \boldsymbol{a}_j$ on the right-hand side have subscript less than i^*. This contradicts the fact (from the original collision structure) that $\boldsymbol{q}_{i^*} \notin \mathsf{span}(\mathsf{rows}(\boldsymbol{M}) \cup \{\boldsymbol{q}_j, \boldsymbol{a}_j \mid j < i^*\})$.

Case $j^* > i^*$: We can solve for $\boldsymbol{a}_{j^*}$ in the above expression, yielding:

$$\boldsymbol{a}_{j^*} = -\frac{1}{\beta'_{j^*}} \left(\sum_{j \in S \setminus \{i^*, \ldots, n\}} \alpha'_j \boldsymbol{q}_j - \boldsymbol{q}_{i^*} + \sum_{j \in S \setminus \{j^*\}} \beta'_j \boldsymbol{a}_j + \boldsymbol{\gamma}' \boldsymbol{M} \right)$$

But now all nonzero $\boldsymbol{q}_j$ and $\boldsymbol{a}_j$ terms on the right-hand side have subscript less than j^*. This contradicts the fact (from the original collision structure) that $\boldsymbol{a}_{j^*} \notin \mathsf{span}(\{\boldsymbol{q}_j \mid j < j^*\} \cup \{\boldsymbol{a}_j \mid j < j^*\} \cup \mathsf{rows}(\boldsymbol{M}))$.

In either case we have a contradiction to the claim that c_{i^*} is moved back into LEFT. Since the algorithm terminates with at least $c_{i^*} \in \mathsf{RIGHT}$, it outputs some valid collision structure. □

3.5 Breaking Collision Resistance implies Collision Structure

So far our discussion has centered around the relationship between collision structures and second-preimage resistance. We now show that if $\mathcal{P}$ fails to be even *collision resistant* (in the random oracle model), then it has a collision structure. The main approach is to observe the oracle queries made by an arbitrary attacker (who computes a collision), and "extract" a collision structure from these queries.

The results in this subsection hold only for the following subclass of Linicrypt programs. In Section 5.2 we discuss specifically why the results are restricted to this subclass.

Definition 9. *Let $\mathcal{P} = (\boldsymbol{M}, \mathcal{C})$ be a Linicrypt program, with $\mathcal{C} = \{(t_1, \boldsymbol{q}_1, \boldsymbol{a}_1), \ldots, (t_n, \boldsymbol{q}_n, \boldsymbol{a}_n)\}$. If all of $\{t_1, \ldots, t_n\}$ are distinct then we say that $\mathcal{P}$ has **distinct nonces**.*

Lemma 10. *Let $\mathcal{P}$ be a deterministic Linicrypt program with distinct nonces that makes n oracle queries. Let $\mathcal{A}$ be an oracle program that makes at most N oracle queries. If*

$$\Pr[\mathsf{ColGame}(\mathcal{P}, \mathcal{A}, \lambda) = 1] > \left(\frac{N}{n}\right)^{2n} / |\mathbb{F}_\lambda|$$
$$\textbf{\textit{or if}}\ \Pr[\mathsf{2PIGame}(\mathcal{P}, \mathcal{A}, \lambda) = 1] > \left(\frac{N}{n}\right)^{n} / |\mathbb{F}_\lambda|$$

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

Proof. Without loss of generality, we can assume the following about $\mathcal{A}$:

- Let $(\boldsymbol{x}, \boldsymbol{x}')$ be the two preimages from the games (in $\mathsf{2PIGame}$ $\mathcal{A}$ gets $\boldsymbol{x}$ as input and gives $\boldsymbol{x}'$ as output; in $\mathsf{ColGame}$ $\mathcal{A}$ outputs both $\boldsymbol{x}$ and $\boldsymbol{x}'$). We assume that $\mathcal{A}^H$ has made the oracle queries that $\mathcal{P}^H(\boldsymbol{x})$ and $\mathcal{P}^H(\boldsymbol{x}')$ will make. In $\mathsf{ColGame}$ this can be achieved by modifying $\mathcal{A}$ to run these two computations as its last action. In $\mathsf{2PIGame}$ this can be achieved by having $\mathcal{A}$ run $\mathcal{P}^H(\boldsymbol{x})$ as its *first* action and $\mathcal{P}^H(\boldsymbol{x}')$ as its last action.
- $\mathcal{A}$ never repeats a query to H. This can be achieved by simple memoization. Note that when $\mathcal{A}$ runs, say, $\mathcal{P}^H(\boldsymbol{x}')$ as its last action, some of those oracle queries may have been made previously.
- $\mathcal{A}^H$ can actually output $(\boldsymbol{v}, \boldsymbol{v}')$, where $\boldsymbol{v}$ is the set of *base variables* in the computation of $\mathcal{P}^H(\boldsymbol{x})$, and $\boldsymbol{v}'$ the base variables in $\mathcal{P}^H(\boldsymbol{x}')$. This is because the base variables are computed during the process of running $\mathcal{P}^H(\boldsymbol{x})$ and $\mathcal{P}^H(\boldsymbol{x}')$.

Note that the base variables have the following property. Let $c = (t, \boldsymbol{q}, \boldsymbol{a})$ be one of the oracle constraints of $\mathcal{P}$. Then the computation $\mathcal{P}^H(\boldsymbol{x})$ (and hence $\mathcal{A}^H$ as well) at some point makes an oracle query $H(t, \boldsymbol{q} \cdot \boldsymbol{v})$ and gets a response $\boldsymbol{a} \cdot \boldsymbol{v}$.

From these assumptions, whenever $\mathcal{A}$ outputs a successful collision there exist well-defined mappings $T, T' : \mathcal{C} \to \mathbb{N}$ such that:

- For every constraint $c = (t, \boldsymbol{q}, \boldsymbol{a}) \in \mathcal{C}$, the $T(c)$th query made by $\mathcal{A}^H$ is the one corresponding to oracle constraint c in the computation of $\mathcal{P}^H(\boldsymbol{x})$. In other words, it is the query in which $\mathcal{A}^H$ "decided" what $\boldsymbol{q} \cdot \boldsymbol{v}$ should be (and learned what $\boldsymbol{a} \cdot \boldsymbol{v}$ was as a result of the query).
- Similarly, the $T'(c)$th query made by $\mathcal{A}^H$ is the one corresponding to oracle constraint c in the computation of $\mathcal{P}^H(\boldsymbol{x}')$. This is the query in which $\boldsymbol{q} \cdot \boldsymbol{v}'$ was determined.

How many possible mappings (T, T') are there if $\mathcal{A}$ makes N oracle queries? Let N_i be the number of oracle queries that $\mathcal{A}$ makes which have nonce t_i. Since the nonces are distinct, we have $\sum_i N_i \leq N$. There are only N_i choices for how T or T' can map $T(c_i)$. Hence there are at most $\prod_{i=1}^{n} N_i^2$ possible (T, T') mappings. However, in the $\mathsf{2PIGame}$, the mapping T is completely fixed since we assume $\mathcal{A}$ performs the computation $\mathcal{P}^H(\boldsymbol{x})$ as its first action. In that case, there are only $\prod_{i=1}^{n} N_i$ choices of the mapping T'. These products are maximized when each $N_i = N/n$, so we get an upper bound of $(N/n)^{2n}$ possible (T, T') mappings in the $\mathsf{ColGame}$ and $(N/n)^n$ mappings in the $\mathsf{2PIGame}$.

Applying the pigeonhole principle and uniting both cases from the statement of the lemma (collision game and second preimage game), there is a *specific* (T, T') such that:

$$\Pr[\mathcal{A}^H \text{ outputs a valid collision while using mappings } (T, T')] > 1/|\mathbb{F}_\lambda|$$

For the rest of the proof, we condition on the event that $\mathcal{A}$ computes a collision while using this *specific* mapping (T, T'). This is without loss of generality by making $\mathcal{A}$, as its final action, output $\perp$ if it observes that some different mapping is used. Hence we can view the association between oracle calls of $\mathcal{P}$ and $\mathcal{A}$ as fixed *a priori.* That is, we can know in advance that a particular oracle call of $\mathcal{A}$ will determine the value of $\boldsymbol{q} \cdot \boldsymbol{v}$ (or $\boldsymbol{q} \cdot \boldsymbol{v}'$) for a specific $\boldsymbol{q}$.

For some $c \in \mathcal{C}$, if $T(c) = T'(c)$, then we call c **convergent**. In this case, $\mathcal{P}^H(\boldsymbol{x})$ and $\mathcal{P}^H(\boldsymbol{x}')$ make the same c-query and receive the same output. In other words, under such a mapping T, T', adversary $\mathcal{A}^H$ will choose that $\boldsymbol{q} \cdot \boldsymbol{v} = \boldsymbol{q} \cdot \boldsymbol{v}'$. If $T(c) \neq T'(c)$, we call c **divergent** — $\mathcal{P}^H(\boldsymbol{x})$ and $\mathcal{P}^H(\boldsymbol{x}')$ make different c-queries, i.e., $\boldsymbol{q} \cdot \boldsymbol{v} \neq \boldsymbol{q} \cdot \boldsymbol{v}'$.

If *all* $c \in \mathcal{C}$ are convergent, then two distinct inputs $\boldsymbol{x}$ and $\boldsymbol{x}'$ cause $\mathcal{P}$ to make identical oracle queries and give identical output. Hence $\mathcal{P}$ is degenerate, and we are done. We continue assuming that some query is divergent, and will conclude that $\mathcal{P}$ has a collision structure.

Define $\mathsf{finish}(c) = \max\{T(c), T'(c)\}$. Note that since $\mathcal{P}$ has distinct nonces, an oracle query made by $\mathcal{A}$ cannot be associated with more than one $c \in \mathcal{C}$. Hence finish is an injective function.

We obtain a collision structure for $\mathcal{P}$ as follows. Order the oracle constraints in $\mathcal{C}$ as $(c_1, \ldots, c_n)$, where all of the convergent queries come first, followed by the divergent queries ordered by increasing finish time. Let i^* be the index of the divergent query with earliest finish time. Then:

- $i^* \leq i \Leftrightarrow c_i$ is divergent
- $i^* \leq i < j \Leftrightarrow \mathsf{finish}(i) < \mathsf{finish}(j)$

Claim 11. $(i^*; c_1, \ldots, c_n)$ *is a collision structure for* $\mathcal{P}$*.*

In the following, we write each oracle constraint c_i as $c_i = (t_i, \boldsymbol{q}_i, \boldsymbol{a}_i)$.

For $j < i^*$, the query c_j is convergent so we have $\boldsymbol{q}_j \cdot \boldsymbol{v} = \boldsymbol{q}_j \cdot \boldsymbol{v}'$ and $\boldsymbol{a}_j \cdot \boldsymbol{v} = \boldsymbol{a}_j \cdot \boldsymbol{v}'$. Since the outputs of the two executions of $\mathcal{P}$ are also identical, we also have $\boldsymbol{M}\boldsymbol{v} = \boldsymbol{M}\boldsymbol{v}'$. Since c_{i^*} is divergent, we have $\boldsymbol{q}_{i^*} \cdot \boldsymbol{v} \neq \boldsymbol{q}_{i^*} \cdot \boldsymbol{v}'$. From this we conclude that:

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

This is the first property required of a collision structure.

It remains to show that for all $i > i^*$,

$$\boldsymbol{a}_i \notin \mathsf{span}\Big(\{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_i\} \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i-1}\} \cup \mathsf{rows}(\boldsymbol{M})\Big).$$

Suppose for contradiction that the above is false, and that we actually have:

$$\boldsymbol{a}_i = \sum_{j \leq i} \alpha_j \boldsymbol{q}_j + \sum_{j < i} \beta_j \boldsymbol{a}_j + \boldsymbol{\gamma}\boldsymbol{M}$$

Focus on the moment when $\mathcal{A}$ has asked its $\mathsf{finish}(c_i)$th query and is awaiting the response from H. By symmetry, suppose $\mathsf{finish}(c_i) = T'(c_i)$, so that this query is on $\boldsymbol{q}_i \cdot \boldsymbol{v}'$; the result of the query will be assigned to $\boldsymbol{a}_i \cdot \boldsymbol{v}'$. At this moment:

- All queries c_j for $i^* \leq j < i$ are finished. This means that the oracle queries of $\mathcal{A}^H$ have already determined $\boldsymbol{q}_j \cdot \boldsymbol{v}$, $\boldsymbol{a}_j \cdot \boldsymbol{v}$, $\boldsymbol{q}_j \cdot \boldsymbol{v}'$, and $\boldsymbol{a}_j \cdot \boldsymbol{v}'$. Further, the queries (but not responses) of oracle constraint c_i have been fixed as well — these values are $\boldsymbol{q}_i \cdot \boldsymbol{v}$ and $\boldsymbol{q}_i \cdot \boldsymbol{v}'$.
- $\boldsymbol{a}_i \cdot \boldsymbol{v}$ has already been fixed, since this happened at time $T(c_i) < T'(c_i)$. But $\boldsymbol{a}_i \cdot \boldsymbol{v}'$ is about to be chosen as a uniform field element.

Now consider the expression $\boldsymbol{a}_i \cdot (\boldsymbol{v}' - \boldsymbol{v})$:

$$\boldsymbol{a}_i \cdot (\boldsymbol{v}' - \boldsymbol{v}) = \sum_{j \leq i} \alpha_j \boldsymbol{q}_j \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{j < i} \beta_j \boldsymbol{a}_j \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \boldsymbol{\gamma} \boldsymbol{M} (\boldsymbol{v}' - \boldsymbol{v})$$

For $j < i^*$ we know that query c_j is convergent. This implies that $\boldsymbol{q}_j \cdot (\boldsymbol{v}' - \boldsymbol{v}) = 0$ and $\boldsymbol{a}_j \cdot (\boldsymbol{v}' - \boldsymbol{v}) = 0$. We also know that $\boldsymbol{M}(\boldsymbol{v}' - \boldsymbol{v}) = 0$, in the case that $\mathcal{A}^H$ is successful generating a collision. Cancelling these terms gives:

$$\boldsymbol{a}_i \cdot (\boldsymbol{v}' - \boldsymbol{v}) = \sum_{j=i^*}^{i} \alpha_j \boldsymbol{q}_j \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{j=i^*}^{i-1} \beta_j \boldsymbol{a}_j \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

Isolating $\boldsymbol{a}_i \cdot \boldsymbol{v}'$ gives:

$$\boldsymbol{a}_i \cdot \boldsymbol{v}' = -\boldsymbol{a}_i \cdot \boldsymbol{v} + \sum_{j=i^*}^{i} \alpha_j \boldsymbol{q}_j \cdot (\boldsymbol{v}' - \boldsymbol{v}) + \sum_{j=i^*}^{i-1} \beta_j \boldsymbol{a}_j \cdot (\boldsymbol{v}' - \boldsymbol{v})$$

But all terms on the right-hand side have already been fixed, while the term on the left is chosen uniformly in $\mathbb{F}$. So equality holds with probability $1/|\mathbb{F}_\lambda|$. This contradicts the assumption that $\mathcal{A}$ succeeds with strictly greater probability. □

3.6 Putting Everything Together

Our main characterization shows that second-preimage resistance and collision resistance coincide for this class of Linicrypt programs, in a very strong sense:

Theorem 12. *Let $\mathcal{P}$ be a deterministic Linicrypt program with distinct nonces, making n oracle queries. Then the following are equivalent:*

1. *There is an adversary $\mathcal{A}$ making N oracle queries such that*

$$\Pr[\mathsf{ColGame}(\mathcal{P}, \mathcal{A}, \lambda) = 1] > \left(\frac{N}{n}\right)^{2n} / |\mathbb{F}_\lambda|.$$

2. *There is an adversary $\mathcal{A}$ making N oracle queries such that*

$$\Pr[\mathsf{2PI}(\mathcal{P}, \mathcal{A}, \lambda) = 1] > \left(\frac{N}{n}\right)^{n} / |\mathbb{F}_\lambda|.$$

3. *There is an adversary $\mathcal{A}$ making at most $2n$ oracle queries such that*

$$\Pr[\mathsf{2PIGame}(\mathcal{P}, \mathcal{A}, \lambda) = 1] = 1.$$

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

Corollary 13. *The collision resistance (equivalently, second-preimage resistance) of deterministic, distinct-nonce Linicrypt programs $\mathcal{P}$ can be decided in polynomial time (in the size of $\mathcal{P}$'s algebraic representation).*

Proof. Using standard linear algebraic operations (e.g., Gaussian elimination), one can check $\mathcal{P}$ for degeneracy or for the existence of a collision structure in polynomial time. □

4 A Simple Application

We can illustrate the use of our main theorem with a simple example application. Suppose we have access to a random oracle which is compressing by a factor of 2-to-1. In the Linicrypt notation, this would be an oracle that takes 2 field elements (and the oracle nonce) as input and produces one field element as output — $H : \{0,1\}^* \times \mathbb{F}^2 \to \mathbb{F}$. If we require a collision resistant function that compresses by k-to-1 (for some fixed k), the following natural Merkle-Damgård-style iterative hash comes to mind:

$\mathcal{P}^H(x_1, x_2, \ldots, x_k)$:
$y_1 := x_1$
$y_2 := H(2; y_1, x_2)$
$y_3 := H(3; y_2, x_3)$
$\vdots$
$y_k := H(k; y_{k-1}, x_k)$
return y_k

The algebraic representation of this program is:

$$
\begin{array}{r|c}
 & \begin{array}{cccccccccc} x_2 & x_3 & \cdots & x_k & y_1 & y_2 & y_3 & \cdots & y_{k-1} & y_k \end{array} \\
\hline
\boldsymbol{M} = & \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \\
\hline
\boldsymbol{Q}_2 = & \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \\
\boldsymbol{a}_2 = & \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 1 & 0 & \cdots & 0 & 0 \end{bmatrix} \\
\hline
\boldsymbol{Q}_3 = & \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \\
\boldsymbol{a}_3 = & \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 0 & 1 & \cdots & 0 & 0 \end{bmatrix} \\
\hline
\vdots & \\
\hline
\boldsymbol{Q}_k = & \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 1 & 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \\
\boldsymbol{a}_k = & \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix}
\end{array}
$$

We have numbered the oracle constraints so that constraint $(i, \boldsymbol{Q}_i, \boldsymbol{a}_i)$ corresponds to the statement "$y_i := H(i; y_{i-1}, x_i)$" in $\mathcal{P}$.

To determine whether this program is collision-resistant, we execute the FindColStruct algorithm.Initially all oracle constraints start in the set LEFT, and RIGHT starts out empty. The first loop in FindColStruct moves oracle constraints from LEFT to RIGHT whenever their $\boldsymbol{a}_i$ value is linearly independent of all other vectors appearing in LEFT (the multiset of vectors is represented as the variable V in FindColStruct).

In this program, every $\boldsymbol{a}_i$ vector is zeroes everywhere except for a 1 corresponding to the "y_i" column. Also note that $\boldsymbol{a}_k$ is identical to $\boldsymbol{M}$, and $\boldsymbol{a}_i$ (for $i < k$) appears as the first row of $\boldsymbol{Q}_{i+1}$ (see the example with $\boldsymbol{a}_2$ and $\boldsymbol{Q}_3$ above). In other words, every $\boldsymbol{a}_i$ is always in the span of other vectors appearing in LEFT, so no oracle constraint will ever be added to RIGHT.

Hence, FindColStruct will terminate with RIGHT $= \emptyset$ and return $\perp$. From our main characterization, this proves that the function is collision-resistant.

5 Extensions, Limitations, Future Work

5.1 Generalizing to Higher Arity

For simplicity our results were proven for Linicrypt programs in which all oracle calls have arity 1. That is, $H : \{0,1\}^* \times \mathbb{F} \to \mathbb{F}$, and all oracle constraints have the form $(t, \boldsymbol{q}, \boldsymbol{a})$ where $\boldsymbol{q}$ is a single row. This reflects a program that always queries the oracle as $H(t; v)$ where v is a *single* field element.

More generally, Linicrypt allows calls to H with multiple field elements as arguments. This leads to oracle constraints of the form $(t, \boldsymbol{Q}, \boldsymbol{a})$ where $\boldsymbol{Q}$ is now a matrix. We briefly discuss the changes necessary to support such programs. Basically, whenever the definitions (of degeneracy & collision structure) or algorithms (to find a second preimage or to find a collision structure) refer to $\boldsymbol{q}$, the analogous condition should hold with respect to all rows of $\boldsymbol{Q}$.

The generalized definition of degeneracy (Definition 4) is that:

$$\mathsf{span}(\boldsymbol{e}_1, \ldots, \boldsymbol{e}_k) \not\subseteq \mathsf{span}\left(\left[\bigcup_{(t,\boldsymbol{Q},\boldsymbol{a})\in\mathcal{C}} \mathsf{rows}(\boldsymbol{Q})\right] \cup \{\boldsymbol{a} \mid (t, \boldsymbol{Q}, \boldsymbol{a}) \in \mathcal{C}\} \cup \mathsf{rows}(\boldsymbol{M})\right)$$

The generalized definition of collision structure (Definition 6) requires the following change:

2. $\mathsf{rows}(\boldsymbol{Q}_{i^*}) \not\subseteq \mathsf{span}\Big(\mathsf{rows}(\boldsymbol{Q}_1) \cup \cdots \cup \mathsf{rows}(\boldsymbol{Q}_{i^*-1}) \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{i^*-1}\} \cup \mathsf{rows}(\boldsymbol{M})\Big)$
3. For $j \geq i^*$: $\boldsymbol{a}_j \notin \mathsf{span}\Big(\mathsf{rows}(\boldsymbol{Q}_1) \cup \cdots \cup \mathsf{rows}(\boldsymbol{Q}_j) \cup \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{j-1}\} \cup \mathsf{rows}(\boldsymbol{M})\Big)$

Specifically, for item (2) it is enough if *any row* of $\boldsymbol{Q}_{i^*}$ is not in the given span.

In the FindSecondPreimage algorithm (Figure 1), there are times when the algorithm chooses $\boldsymbol{q}_j \cdot \boldsymbol{v}'$ arbitrarily. This happens when such a constraint would be linearly independent of the existing constraints on $\boldsymbol{v}'$. In the analogous generalized case, we might have *only some of the rows* of $\boldsymbol{Q}_j$ linearly independent of the existing constraints. In that case, some of the components of $\boldsymbol{Q}_j \times \boldsymbol{v}'$ are already fixed. We obviously cannot choose these arbitrarily — only the unconstrained positions in $\boldsymbol{Q}_j \times \boldsymbol{v}'$ are fixed arbitrarily. One can verify that the algorithm only attempts to arbitrarily fix some values if there is some row of $\boldsymbol{Q}_j$ linearly independent with existing constraints on $\boldsymbol{v}'$.

In the FindColStruct algorithm (Figure 2) we let V now contain $\boldsymbol{Q}$-matrices as well as simple $\boldsymbol{a}$-vectors. Then we overload notation so that $\mathsf{span}(V)$ considers the span of all of the rows of all matrices/vectors in V. The second "while" condition is modified as follows:

$$\text{while } \exists (t, \boldsymbol{Q}, \boldsymbol{a}) \in \mathsf{RIGHT} \text{ such that } \mathsf{rows}(\boldsymbol{Q}) \subseteq \mathsf{span}(V)$$

In other words, $(t, \boldsymbol{Q}, \boldsymbol{a})$ is moved from LEFTto RIGHTif *all rows of* $\boldsymbol{Q}$ are spanned by V.

With these modifications, all proofs in Section 3 go through with straight-forward modifications.

5.2 Why the Restriction to Distinct Nonces?

The main characterization holds for Linicrypt programs with distinct nonces. It is instructive to understand why the results are limited in this way. Specifically, where do we use the property of distinct nonces?

Suppose $\mathcal{A}$ breaks the collision-resistance of $\mathcal{P}$. We observe the oracle queries made by $\mathcal{A}$ and obtain a mapping between these queries and the ones made in $\mathcal{P}^H(\boldsymbol{x})$ and $\mathcal{P}^H(\boldsymbol{x}')$. When the

nonces are distinct, a query made by $\mathcal{A}$ can only be associated with a unique oracle constraint $c \in \mathcal{C}$. When the nonces are not distinct, a single query of $\mathcal{A}$ can serve double-duty and correspond to two oracle constraints of $\mathcal{P}$. This indeed causes the argument to break down.

We illustrate with the two example Linicrypt programs:

$$\mathcal{P}_1^H(x, y) = H(2, H(1, x)) - H(3, y)$$
$$\mathcal{P}_2^H(x, y) = H(\quad H(\quad x)) - H(\quad y)$$

The first has distinct nonces and is indeed collision resistant (it has no collision structure). The second program is not collision-resistant, because $\mathcal{P}_2^H(x, H(x)) = 0$ for all x. In other words, $(x, H(x))$ and $(x', H(x'))$ constitute a collision.

When given inputs of this form, $\mathcal{P}_2$ makes duplicate queries — both $H(H(x))$ (the outermost H-call) and $H(y)$ receive the same argument. In our previous proofs, we would observe the adversary making such a query, which would have to be associated with two distinct oracle constraints.

Another way of seeing what happens is that in the *algebraic representation* of $\mathcal{P}_2$, the base variables $H(x)$ and y correspond to independent vectors. In this case, the adversary's choice of inputs causes these vectors to coincide, and this has the effect of "collapsing" two oracle queries.

Interestingly, it is possible to give an ad-hoc argument that $\mathcal{P}_2$ is second-preimage resistant. When x and y are chosen uniformly, this has the effect of keeping the vectors (in the algebraic representation) corresponding to $H(x)$ and y independent. We can then argue that the adversary doesn't make any oracle query that is associated with two distinct queries of $\mathcal{P}_2$, so the reasoning of our main theorem also applies in this case. Hence, $\mathcal{P}_2$ demonstrates that our main characterization is different for Linicrypt programs with non-distinct nonces.

5.3 Random Oracle vs Ideal Cipher

A natural application of collision resistance would be the constructions of collision-resistant hash functions from an ideal cipher [12, 3]. It should be possible to use Linicrypt to reason about constructions in the ideal cipher model, although it would require non-trivial modifications. We could interpret $E(k, m)$ as $H(\mathtt{E}, k, m)$ and $D(k, c)$ as $H(\mathtt{D}, k, c)$. The constraint that $D(k, E(k, m)) = m$ adds some extra structure that must be reflected in the algebraic representation. For example, if a program $\mathcal{P}$ makes a query $c = E(k, m)$, we must consider the adversary's ability to make this forward query but also its ability to make the corresponding backwards query $D(k, c)$. Both forward/backwards queries must be considered before deeming the *pair* of queries $E(k, m)$ and $D(k, c)$ unreachable.

We do not foresee the transition to ideal cipher model to be particularly problematic. However, the specific analysis of [3] shows several constructions of hash functions from ideal ciphers where the *round functions are not collision-resistant*, and yet their use in a Merkle-Damgård construction gives a collision-resistant result. So far, the theory of Linicrypt is not developed enough to reason about programs with looping constructs, as in an iterated hash function (despite the fact that such reasoning happens to be tractable for the specific example in Section 4).

References

[1] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 163–192. Springer, Heidelberg, May 1997.

[2] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn. SPHINCS: Practical stateless hash-based signatures. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 368–397. Springer, Heidelberg, Apr. 2015.

[3] J. Black, P. Rogaway, and T. Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 320–335. Springer, Heidelberg, Aug. 2002.

[4] J. N. Bos and D. Chaum. Provably unforgeable signatures. In E. F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 1–14. Springer, Heidelberg, Aug. 1993.

[5] B. Carmer and M. Rosulek. Linicrypt: A model for practical cryptography. In M. Robshaw and J. Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 416–445. Springer, Heidelberg, Aug. 2016.

[6] E. Dahmen, K. Okeya, T. Takagi, and C. Vuillaume. Digital signatures out of second-preimage resistant hash functions. *PQCrypto*, 5299:109–123, 2008.

[7] S. Even, O. Goldreich, and S. Micali. On-line/off-line digital schemes. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 263–275. Springer, Heidelberg, Aug. 1990.

[8] A. Hülsing. W-OTS+ - shorter signatures for hash-based signature schemes. In A. Youssef, A. Nitaj, and A. E. Hassanien, editors, *AFRICACRYPT 13*, volume 7918 of *LNCS*, pages 173–188. Springer, Heidelberg, June 2013.

[9] R. Impagliazzo. A personal view of average-case complexity. In *Proceedings of the Tenth Annual Structure in Complexity Theory Conference, Minneapolis, Minnesota, USA, June 19-22, 1995*, pages 134–147. IEEE Computer Society, 1995.

[10] L. Lamport. Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, Oct. 1979.

[11] R. C. Merkle. A digital signature based on a conventional encryption function. In C. Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Heidelberg, Aug. 1988.

[12] B. Preneel, R. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: A synthetic approach. In D. R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 368–378. Springer, Heidelberg, Aug. 1994.

[13] D. Wagner. A generalized birthday problem. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 288–303. Springer, Heidelberg, Aug. 2002.