# Preimage Awareness in Linicrypt

Zahra Javar
*Computer Science Department*
*University of Victoria*
Victoria, BC, CANADA
z.javar@gmail.com

Bruce M. Kapron
*Computer Science Department*
*University of Victoria*
Victoria, BC, CANADA
bmkapron@uvic.ca

***Abstract*—We extend the analysis of collision-resistant hash functions in the Linicrypt model presented by McQuoid, Swope & Rosulek (TCC 2019) in order to characterize preimage awareness, a security property defined by Dodis, Ristenpart & Shrimpton (Eurocrypt 2009), who also demonstrate its utility in the construction of indifferentiable hash functions. We present a simple and efficiently-checkable property of Linicrypt programs which characterizes preimage awareness. Finally, we show that this characterization may be efficiently automated and as an example, use it to enumerate all preimage-aware compression functions which use two calls to the random oracle. This includes several functions shown to be preimage aware by Dodis et. al. using hand-crafted proofs.**

***Index Terms*—hash function, compression function, preimage awareness, Linicrypt, indifferentiability**

## I. Introduction

Hash functions are a fundamental cryptographic primitive used as a building block in a large number of cryptographic constructions [MF21]. Applications including domain extension for message-authentication codes [BCK96] and digital signatures [Dam87], as well as hash-based signatures [BC92], [EGM96], [Lam79], [Mer87], rely on collision resistance properties for their security.

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* [Dam89], [Mer89], 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 [BR93], 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 [CGH04]., it has proven useful as a means of providing evidence that a 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 [CDMP05] 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 [MRH04]. 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 [DRS09a], 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 Section II.

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 which support automated verification, or even automated generation of such functions.* This is the problem we address in this paper. In particular, we propose an approach using the Linicrypt framework [CR16], which allows constructions specified by straight-line programs over a finite field, with access to a random oracle. Informally, Linicrypt provides definitions of hash functions which combine successive calls to a random oracle, along with the formation of linear combinations (over some finite field) of the results of these calls and the function inputs. Outputs are also linear combinations of this form. In a subsequent work [MSR19], Linicrypt is applied to collision-

Research supported in part by NSERC RGPIN-2021-02481

2023 IEEE 36th Computer Security Foundations Symposium (CSF) | 979-8-3503-2192-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/CSF57540.2023.00017


DOI 10.1109/CSF57540.2023.00017

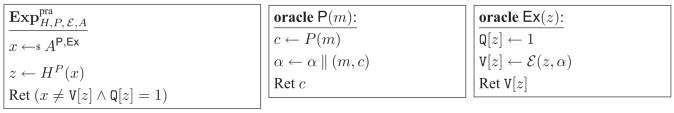


Fig. 1. Preimage awareness game from [DRS09a]. Oracle $\mathsf{P}$ mediates access to the ideal primitive, while recording all queries made, while $\mathsf{Ex}$ mediates access to the extractor, while using $\mathsf{Q}, \mathsf{V}$ to record the image/preimage pairs involved in each call.

resistant hash functions, and it is shown that there is an efficiently-checkable algebraic condition on programs which may be used to characterize collision resistance and second-preimage resistance. A detailed description of Linicrypt (specialized to our setting) is given in Section III. We build on the approach of [MSR19] in order to characterize PrA compression functions defined by Linicrypt programs.

### A. Results

We give an algebraic characterization of PrA for hash functions modeled as Linicrypt programs which have access to a random oracle which is sound, complete and efficiently checkable.

The intuition behind our result 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. We formalize this via the notion of *PrA critical query* in Definition V.1 and show in Theorem V.8 that this syntactic condition completely characterize preimage awareness. We also show that the problem of finding critical queries is a simple matter of solving a system of linear equations, and so is solvable in polynomial time. Finally, as an example, we show how this characterization may be used to enumerate all PrA compression functions with two inputs and using two independent random oracle calls.

Our results are significant from both theoretical and practical perspectives. We give a general characterization of PrA compression functions with any number of inputs and random oracle calls, demonstrating the utility and range of the Linicrypt model. In general, the existence of a PrA critical query may be automatically verified (in polynomial time.) We contrast this with known proofs of PrA for the Shrimpton-Stam and Dodis-Pietrzak-Puniya compression functions [DRS09a], 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.

## II. Preimage awareness

We will briefly review the definition and some important facts regarding preimage awareness as defined in [DRS09a]. 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 $\alpha$ 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. The more restrictive version we have described is called 1-PrA. We give a formal definition of 1-PrA, for hash functions defined by Linicrypt programs below in Definition IV.3. We note that by [DRS09a], 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 ( [DRS09a], 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. ( [DRS09a], 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.

## III. Linicrypt Programs

The Linicrypt formalism, introduced in [CR16], 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 [CR16], 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.[1] While [CR16] focused

[1]More recently ( [HRR22]), Linicrypt was also used to characterize the security of block cipher modes of operation.

on indistinguishability-based security for inputless Linicrypt programs, subsequent work [MSR19] demonstrated the utility of Linicrypt for characterizing collision resistance and second-preimage resistance for fixed input-length hash functions constructed using a random oracle. Our presentation follows the model of [MSR19], 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[2], 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 matrix-vector product. Note that we will sometimes think of matrices as a column vector of row vectors (so we may write $\boldsymbol{M} = (\boldsymbol{m}_1^\top, \ldots, \boldsymbol{m}_k^\top)^\top$.)

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[3]

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 two-input Linicrypt program with random oracle $H$

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

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 [DPP08], which is one of the compression functions shown to be PrA in [DRS09a]). We will return to this example after giving our characterization of PrA and verifying that it satisfies the critical query condition. Other examples of Linicrypt programs may be found in [CR16], [MSR19].

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 [MSR19], 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 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 [DRS09a]. 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 [MSR19], Section 5.1, for a discussion of extending the model to random oracles with multiple field inputs.

As noted in [MSR19], 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 $\lambda$. 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 $\lambda$, 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 [MSR19], in the concrete setting of their work (and ours), this choice is not significant. In all the examples given in this paper, we work over $\mathbb{GF}(2^\lambda)$ and assume that programs work with coefficients from $\{0, 1\}$.

An important observation of [CR16] 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 variables are represented as canonical basis vectors, so for a $\mathcal{P}$ with $k$ inputs and $n$ oracle queries the base variables are $\mathbf{e}_1, \ldots, \mathbf{e}_k, \mathbf{e}_{k+1}, \ldots, \mathbf{e}_{k+n}$ where, $\mathbf{e}_i$ denotes the $i$th 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 [CR16], 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

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

[3]The Linicrypt model, introduced in [CR16], also allows the assignment of a random field element to a variable. Here, as in [MSR19], we consider only deterministic programs.

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 [MSR19]) 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 $1 \le i \le k$ 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

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

### A. Dealing with Constant Values

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. [BRSS10] 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 [Sta09], where it is noted that translation by a constant preserves bijectivity. We take a more general approach, 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 security properties of hash functions defined by Linicrypt programs using constants without any changes to our 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 [BRSS10]. The following example shows how a compression function with a constant value $c$ can be modeled in Linicrypt.

**Example III.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,*

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

$$\boldsymbol{v}_{base} = (x_1, x_2, c, v_5, v_6)^\top$$
$$\boldsymbol{M} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$
$$\mathcal{C} = \langle (t_1, \boldsymbol{q}_1, \boldsymbol{a}_1), (t_2, \boldsymbol{q}_2, \boldsymbol{a}_2) \rangle$$

*where*

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

## IV. 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 [MSR19],) as well as for preimage awareness, using the 1-PrA formulation as discussed in Section II.

In general, if we are working in a finite field $\mathbb{F}_\lambda$, the field size will be exponential in $\lambda$ and we may take $\lambda$ as our security parameter. The adversary $\mathcal{A}$ is defined with respect to $\lambda$; 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 $\lambda$. 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 IV.1** ( [MSR19] Definition 2)**.** Program $\mathcal{P}$ is $(q,\epsilon)$-*collision resistant* if any oracle adversary $\mathcal{A}$ making at most $q$ queries has probability of success at most $\epsilon$ in the following game:

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

**Definition IV.2** ( [MSR19] Definition 3)**.** Program $\mathcal{P}$ is $(q,\epsilon)$-*2nd-preimage resistant* if any oracle adversary $\mathcal{A}$ making at most $q$ queries has probability of success at most $\epsilon$ in the following game:

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

To define preimage awareness we first need the notion of *extractor*, which we will take it 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 [MSR19]) 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 IV.3.** Program $\mathcal{P}$ is $(q,\epsilon)$-*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 $\epsilon$ in the following game:

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

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 $\boldsymbol{\ell}$.

### A. 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 $\boldsymbol{M}=(\boldsymbol{m}_1^\top,\ldots,\boldsymbol{m}_k^\top)^\top$ and $1\le i\le k$, $\boldsymbol{M}_{-i}$ denotes $\boldsymbol{M}$ with its $i$th row removed, i.e.. $(\boldsymbol{m}_1^\top,\ldots,\boldsymbol{m}_{i-1}^\top,\boldsymbol{m}_{i+1}^\top,\ldots,\boldsymbol{m}_k^\top)^\top$. 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 IV.4.** 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 $i\in\bigcup_{j=1}^n\mathsf{useless}^j(\emptyset)$.

**Remark.** The definition of useless query corresponds to that of [CR16] but extended to the case of programs with inputs.

**Lemma IV.5.** *Let $\mathcal{P}'$ be obtained by removing all useless queries from $\mathcal{P}$. Then $\mathcal{P}'$ is $(q,\epsilon)$-PrA iff $\mathcal{P}$ is $(q,\epsilon)$-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.[4]

**Definition IV.6.** 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, second preimages may be found with probability 1 [MSR18], Lemma 5 and so such programs trivially fail to be PrA. We can also prove this directly:

**Lemma IV.7.** *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 $\boldsymbol{P}=\begin{bmatrix}\boldsymbol{Q}\\ \boldsymbol{A}\\ \boldsymbol{M}\end{bmatrix}$ 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, 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

[4]Our definition corresponds to the version given in revision 2 of [MSR18], correcting an earlier version (which is the version appearing in [MSR19])

$\text{rank}(\boldsymbol{P}) < k+n$, and thus, for some $\boldsymbol{v} \neq \boldsymbol{0}$, $\boldsymbol{P} \times \boldsymbol{v} = \boldsymbol{0}$. The advesary 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.

## V. Characterizing Preimage Awareness

The main result of [MSR19] 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 V.1 we define the notion of *PrA-critical* query, and in Theorem V.8 we use this to characterize preimage awareness. We begin by presenting a motivating example.

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

(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 $\alpha_1$ and $\ell$ at random from $\mathbb{F}$, calls $H(t_1, \alpha_1)$ to obtain $\beta_1$, and outputs $st = \alpha_1$, $\boldsymbol{qs} = (\alpha_1, \beta_1)$ and $\ell$. 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 $\beta_2$ 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 returns 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 V.1.** Let $1 \leq i^* \leq n$. 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 V.8 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 [MSR19]. Recall that in the presentation of [MSR19], constraints are treated as an (unordered) set. We then have:

**Definition V.2** ( [MSR19], 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 $1 \leq i^* \leq n$ 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 $j \geq i^*$, $a'_j \notin \mathsf{span}(\{\boldsymbol{q}'_1, \ldots, \boldsymbol{q}'_j\} \cup \{\boldsymbol{a}'_1, \ldots, \boldsymbol{a}'_{j-1}\} \cup \mathsf{rows}(\boldsymbol{M}))$

**Remark.** We have denoted the $i$th 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 $i$th 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 [MSR19] 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 V.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 V.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 [MSR19], in particular the following

**Lemma V.4** ( [MSR19], 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 V.7 below. While a direct proof would be possible, the use of Lemmas V.3 and V.4 makes things considerbly 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 V.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* $H(t_i, \alpha_i)$*,* $1 \le i \le n$*,* $i \ne i^*$*.*
2) $\beta_i = H(t_i, \alpha_i)$*,* $1 \le i \le n$*,* $i \ne i^*$*.*
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* $\alpha_{i^*}$ *is a random element of* $\mathbb{F}$*.*

*Proof.* For $1 \le i \le n$, 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 previous queries to determine $\alpha_i$, otherwise, it chooses $\alpha_i$ uniformly from $\mathbb{F}$. If $i \ne i^*$, it then sets $\beta_i = H_i(t_i, \alpha_i)$. Finally, for $1 \le j \le r$, 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 $\ell_j$; otherwise it chooses $\ell_j$ uniformly from $\mathbb{F}$. Note that by (†), no $\boldsymbol{q}_i$ or $\boldsymbol{m}_j$ depends on $\boldsymbol{a}_{i^*}$, so $\alpha_i$ and $\ell_j$ 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 $\beta_i$'s are chosen using the corresponding calls to $H$. □

**Lemma V.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 V.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 IV.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 V.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 $j$th row, there exists $j$, $0 \le j \le 2n + r, j \ne n + i^*, i^*$ such that

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

where $b_j \in \mathbb{F}$, $\nu_i \in \{0, 1\}$ and $\gamma_{k+i}$, $1 \le i \le n$, $i \ne i^*$, are all known to $\mathcal{E}$ (in particular, for $i \ne i^*$, $\gamma_{k+i} = \beta_i$.) 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 V.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 $t_i$, $1 \le i \le n$, we have that there are $\prod_{i=1}^n N_i$ possible mappings. Since $\sum_{i=1}^n N_i \le N$, 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 $t$th 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 $1 \le j < k \le n$ implies $T(i_j) < T(i_k)$, and let $s \in [n]$ be such that $T(i_s) \le t$ 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 V.4 that $\mathcal{P}$ has a collision structure, which by Lemma V.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 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

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

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

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

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 V.6 and V.7 gives our main result:

**Theorem V.8.** *Suppose $\mathcal{P}$ is a Linicrypt program making $n$ queries, and $q \geq n$. For sufficiently large $\lambda$, 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 $q \geq n$, this means that it is also $(n, (q^{n+1}/n^n)/|\mathbb{F}|)$-PrA. Choose $\lambda$ so that $|\mathbb{F}| - 1 \geq q^{n+1}/n^n$. Then $\mathcal{P}$ is $(n, 1 - 1/|\mathbb{F}|)$-PrA. The remaining implications follow by Lemmas V.6 and V.7 □

### A. 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 [SS08]. In [DRS09a] 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:

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

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

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

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 $\ell_1$ 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.

### B. Preimage aware compression functions

In this section we use our characterization to generate 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 which supports matrix operations.[5] 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 $a, b, c \in \{x, y, x + y\}$:

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 pre-defined constant, as discussed in Section III-A. Without any additional structural restrictions, this results in an additional 532 possible PrA compression functions.

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

## VI. Conclusion

We have given 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 [DRS09a] which are ad hoc and not obviously amenable to automation. Our approach to PrA provides a uniform and potentially automatable method for constructing indifferentiable hash functions, following the methodology of [DRS09a].

Our work adds to existing results [CR16], [MSR19], [HRR22] that demonstrate the utility of Linicrypt for providing a uniform and potentially automatable framework for security proofs in the random oracle model, using a language that is powerful enough to capture many well-known constructions.

With respect to hash functions, an interesting question is whether a Linicrypt-based characterization of indifferentiability 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 seems to be of a more technical nature than the one we have addressed in this paper. Also, the problem of extending Linicrypt to the ideal cipher model, already suggested in [MSR19], remains an interesting question.

## References


[BC92] J. N. Bos and D. Chaum, "Provably unforgeable signatures," in *CRYPTO '92*, ser. Lecture Notes in Computer Science, E. F. Brickell, Ed., vol. 740. Springer, 1992, pp. 1–14. [Online]. Available: https://doi.org/10.1007/3-540-48071-4_1

[BCK96] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *CRYPTO '96*, ser. Lecture Notes in Computer Science, N. Koblitz, Ed., vol. 1109. Springer, 1996, pp. 1–15. [Online]. Available: https://doi.org/10.1007/3-540-68697-5_1

[BR93] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *CCS '93,*, D. E. Denning, R. Pyle, R. Ganesan, R. S. Sandhu, and V. Ashby, Eds. ACM, 1993, pp. 62–73. [Online]. Available: https://doi.org/10.1145/168588.168596

[BRSS10] J. Black, P. Rogaway, T. Shrimpton, and M. Stam, "An analysis of the blockcipher-based hash functions from PGV," *J. Cryptol.*, vol. 23, no. 4, pp. 519–545, 2010. [Online]. Available: https://doi.org/10.1007/s00145-010-9071-0

[CDMP05] J. Coron, Y. Dodis, C. Malinaud, and P. Puniya, "Merkle-damgård revisited: How to construct a hash function," in *CRYPTO 2005*, ser. Lecture Notes in Computer Science, V. Shoup, Ed., vol. 3621. Springer, 2005, pp. 430–448. [Online]. Available: https://doi.org/10.1007/11535218_26

[CGH04] R. Canetti, O. Goldreich, and S. Halevi, "The random oracle methodology, revisited," *J. ACM*, vol. 51, no. 4, pp. 557–594, 2004. [Online]. Available: https://doi.org/10.1145/1008731.1008734

[CR16] B. Carmer and M. Rosulek, "Linicrypt: A model for practical cryptography," in *CRYPTO 2016, Part III*, ser. Lecture Notes in Computer Science, M. Robshaw and J. Katz, Eds., vol. 9816. Springer, 2016, pp. 416–445. [Online]. Available: https://doi.org/10.1007/978-3-662-53015-3_15

[Dam87] I. Damgård, "Collision free hash functions and public key signature schemes," in *EUROCRYPT 1987*, ser. Lecture Notes in Computer Science, D. Chaum and W. L. Price, Eds., vol. 304. Springer, 1987, pp. 203–216. [Online]. Available: https://doi.org/10.1007/3-540-39118-5_19

[Dam89] ——, "A design principle for hash functions," in *CRYPTO 1989*, ser. Lecture Notes in Computer Science, G. Brassard, Ed., vol. 435. Springer, 1989, pp. 416–427. [Online]. Available: https://doi.org/10.1007/0-387-34805-0_39

[DPP08] Y. Dodis, K. Pietrzak, and P. Puniya, "A new mode of operation for block ciphers and length-preserving macs," in *EUROCRYPT 2008*, N. P. Smart, Ed., vol. 4965. Springer, 2008, pp. 198–219. [Online]. Available: https://doi.org/10.1007/978-3-540-78967-3_12

[DRS09a] Y. Dodis, T. Ristenpart, and T. Shrimpton, "Salvaging Merkle-Damgard for practical applications," *IACR Cryptol. ePrint Arch.*, p. 177, 2009, full version of [DRS09b]. [Online]. Available: http://eprint.iacr.org/2009/177

[DRS09b] ——, "Salvaging Merkle-Damgård for practical applications," in *EUROCRYPT 2009*, A. Joux, Ed., vol. 5479. Springer, 2009, pp. 371–388. [Online]. Available: https://doi.org/10.1007/978-3-642-01001-9_22

[EGM96] S. Even, O. Goldreich, and S. Micali, "On-line/off-line digital signatures," *J. Cryptol.*, vol. 9, no. 1, pp. 35–67, 1996. [Online]. Available: https://doi.org/10.1007/BF02254791

[HRR22] T. Hollenberg, M. Rosulek, and L. Roy, "A complete characterization of security for linicrypt block cipher modes," in *IEEE 35th Computer Security Foundations Symposium (CSF)*. Los Alamitos, CA, USA: IEEE Computer Society, 2022, pp. 423–438. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CSF54842.2022.00028

[Lam79] L. Lamport, "Constructing digital signatures from a one-way function," SRI International, Tech. Rep. CSL 98, 1979.

[Mer87] R. C. Merkle, "A digital signature based on a conventional encryption function," in *CRYPTO 1987*, ser. Lecture Notes in Computer Science, C. Pomerance, Ed., vol. 293. Springer, 1987, pp. 369–378. [Online]. Available: https://doi.org/10.1007/3-540-48184-2_32

[Mer89] ——, "One way hash functions and DES," in *CRYPTO 1989*, ser. Lecture Notes in Computer Science, G. Brassard, Ed., vol. 435. Springer, 1989, pp. 428–446. [Online]. Available: https://doi.org/10.1007/0-387-34805-0_40

[MF21] A. Mittelbach and M. Fischlin, *The Theory of Hash Functions and Random Oracles - An Approach to Modern Cryptography*, ser. Information Security and Cryptography. Springer, 2021. [Online]. Available: https://doi.org/10.1007/978-3-030-63287-8

[MRH04] U. M. Maurer, R. Renner, and C. Holenstein, "Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology," in *TCC 2004*, ser. Lecture Notes in Computer Science, M. Naor, Ed., vol. 2951. Springer, 2004, pp. 21–39. [Online]. Available: https://doi.org/10.1007/978-3-540-24638-1_2

[MSR18] I. McQuoid, T. Swope, and M. Rosulek, "Characterizing collision and second-preimage resistance in linicrypt," *IACR Cryptol. ePrint Arch.*, p. 458, 2018. [Online]. Available: https://eprint.iacr.org/2018/458

[MSR19] ——, "Characterizing collision and second-preimage resistance in linicrypt," in *TCC 2019, Part I*, ser. Lecture Notes in Computer Science, D. Hofheinz and A. Rosen, Eds., vol. 11891. Springer, 2019, pp. 451–470. [Online]. Available: https://doi.org/10.1007/978-3-030-36030-6_18

[SS08] T. Shrimpton and M. Stam, "Building a collision-resistant compression function from non-compressing primitives," in *ICALP 2008*, L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, Eds., vol. 5126. Springer, 2008, pp. 643–654. [Online]. Available: https://doi.org/10.1007/978-3-540-70583-3_52

[Sta09] M. Stam, "Blockcipher-based hashing revisited," in *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, ser. Lecture Notes in Computer Science, O. Dunkelman, Ed., vol. 5665. Springer, 2009, pp. 67–83. [Online]. Available: https://doi.org/10.1007/978-3-642-03317-9_5