Arithmetization for Probabilistically Checkable and Interactive Oracle Proofs

Yinon Horesh
Arithmetization for Probabilistically Checkable and Interactive Oracle Proofs

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Yinon Horesh

Submitted to the Senate of the Technion — Israel Institute of Technology Adar 5780 Haifa March 2020
This research was carried out under the supervision of Prof. Eli Ben-Sasson and Prof. Yuval Ishai, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s master research period, the most up-to-date versions of which being:

<table>
<thead>
<tr>
<th>Authors</th>
<th>Title</th>
<th>Conference/Journal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev</td>
<td>Scalable zero knowledge with no trusted setup</td>
<td>CRYPTO, 2019.</td>
</tr>
</tbody>
</table>

Chapters 1, 2, 4 are adapted from paper [BBHR18b]. The main contribution of this thesis is described in Chapter 3.

ACKNOWLEDGEMENTS

I would like to thank Prof. Eli Ben-Sasson and Prof. Yuval Ishai for their guidance and help in the research leading to the writing of this thesis.

To my parents, family, friends who encouraged me to complete my thesis, and supported me in challenging moments. To my fiance, Efrat, who also helped me with the writing and the proofreading of this thesis.

I would also like to give special thanks to Itay Hazan, for his help in the writing process of this article. I appreciate your time, effort, and thoughts.

The generous financial help of the Technion is gratefully acknowledged.
Contents

List of Figures

Abstract

1 Introduction

1.1 Scalable verification of computational integrity over confidential datasets 3
1.2 From theory to practice ............................................. 5
1.3 Our results .......................................................... 6

2 Arithmetization

2.1 Arithmetization Case Study: The DNA Profile Match Problem .................... 7
2.2 A General Overview on Arithmetization .............................................. 8
2.3 Arithmetization I — Algebraic Intermediate Representation (AIR) ............. 9
2.4 Arithmetization II — Algebraic Linking Interactive Oracle Proof (ALI) ........ 10

3 Polynomial Constraints for Real-World Problems .......................................... 13

3.1 Overview ............................................................ 13
3.2 DNA profile matching .................................................. 14
3.2.1 Rijndael overview .................................................. 15
3.2.2 Implementation technique of the Rijndael cipher ............................... 16
3.2.3 State machine of the Rijndael cipher .................................... 17
3.2.4 From encryption to hash function: Davies-Meyer .............................. 18
3.2.5 DNA profile match (DPM) .......................................... 20
3.3 SHA2 ................................................................. 24
3.3.1 Bit Extraction ....................................................... 24
3.3.2 Integer Decrement .................................................. 24

4 From AIR to ZK-STARK ................................................................................. 29

4.1 Overview ........................................................................ 29
4.2 Algebraic Intermediate Representation (AIR) .................................... 30
4.3 Realization considerations ....................................................... 31
4.4 Measurements of the ZK-STARK for the DNA profile match ................. 33
4.4.1 Prover .................................................................. 33
5 Conclusion and Open Questions

5.1 Using arithmetization for making PCPs/IOPs practical

5.2 New directions

Hebrew Abstract
List of Figures

2.1 Main complexity parameters of basic cryptographic primitives and the benchmark DNA profile match search program ........................................... 10
2.2 The maximal degree for old construction and for ZK-STARK. .................... 11

3.1 Constraints polynomial for the Rijndael-160 SubBytes S-box. ..................... 18
3.2 Constraint polynomials for compression and decompression .......................... 19
3.3 Illustration of the hashchain program, the data structure used in the DNA profile match program. ................................................................. 20
3.4 CheckPairs subroutine .............................................................................. 22
3.5 MatchingResult subroutine ....................................................................... 22
3.6 Algebraic constraints for one pair in the CheckPairs subroutine. ................. 23
3.7 Algebraic constraints for the MatchingResult subroutine. ......................... 23
3.8 Polynomial constraints for decrementing an integer. .................................. 24
3.9 Exemplary Merkle tree with $\text{HEIGHT} = 4$ and $n = 14$. .......................... 26
3.10 Illustration of a Merkle tree membership query. ........................................ 27

4.1 The reduction from an AIR instance to a pair of RPT problems, solved using the FRI protocol. ................................................................. 29
4.2 The time and witness-size rate functions of the DPM benchmark ZK-STARK. .... 32
4.3 Prover running time and total size of all oracles generated by its .................... 33
4.4 Verifier running time and communication complexity ............................... 34
Abstract

Recently, there is an increase in delegating computations to powerful remote servers. This trend raises questions of computational integrity and efficient verification. For instance, the server can be unreliable, therefore it may output erroneous results.

In other contexts, where preserving information privacy is crucial, as in medical and forensic data, other issues might arise. The veils of secrecy, that are designed to keep the data hidden from the public, may be abused to cover up lies and deceit by institutions entrusted with data, unjustly harming citizens and eroding trust in central institutions. Zero knowledge (ZK) proof systems are an ingenious cryptographic solution to this tension between the ideals of personal privacy and institutional integrity, enforcing the latter in a way that does not compromise the former. For ZK systems to be used with Big Data, it is imperative that the public verification process scale sublinearly in data size.

One can use Probabilistically Checkable Proofs (PCP) to verify the correctness of the computations. Furthermore, the verification time is significantly less than the time which would be needed if the client ran the computation by itself. Until recently, however, PCPs have been considered efficient only in theory. Interactive Oracle Proofs (IOP), however, which are interactive versions of PCPs, are more practical.

Here we report how to use arithmetization for building programs for some problems like above. Arithmetization is the encoding of transition functions of computational models as algebraic problems. This technique is used in many theoretical and practical results, and is valuable in making PCPs/IOPs more practical. In order to encode the transition function, we define polynomials over a finite field $F$ such that these polynomials are set to zero if and only if the computation corresponding to that function is executed correctly.

The main contribution of our work is in building arithmetizations for real-world problems. We open our discussion with primitive cryptographic functions, e.g. AES or SHA2, and attempt to arithmetize them as optimally as possible. Then, we compare the arithmetizations in terms of efficiency, stemming from algebraic properties of the functions. Afterwards, we progress to more complex data structures e.g. Merkle-Trees or hashchains. We conclude this work by presenting an efficient construction for several real-world problems.
Chapter 1

Introduction

1.1 Scalable verification of computational integrity over confidential datasets

Privacy and integrity of computation have always been a concern of major importance in multi-party settings. Consider, for instance, the following hypothetical example: Suppose the Police (Prover), that is in charge of the national forensic DNA profile database D, claims that the DNA profile \( p \) of a soon-to-be-appointed and alleged-to-be-corrupt Presidential Candidate, does not appear in D. Can cryptographic protocols convince the doubtful public to believe this claim, without compromising D or \( p \), and without relying on any external trusted party (e.g., the Chief Justice), and with “reasonable” computational resources?

The DNA profile match (DPM) example is a special case of a more general problem, in which one party \( P \) executes a computation \( C \) on a dataset \( D \), but may have an incentive to misreport the correct output \( C(D) \). Such scenarios give rise to the problem of computational integrity (CI)\(^1\), where one or more parties, called verifiers, ought to ensure that \( P \) indeed reports \( C(D) \) rather than an output more favorable to \( P \). When the dataset \( D \) is public, any party \( V \) interested in verifying CI can naïvely re-execute \( C \) on \( D \) and compare its output to that reported by \( P \), as a customer might inspect a restaurant bill, or as a new Bitcoin node will verify its blockchain [Nak09]. This naïve solution does not scale because the time spent by the verifier \( T_V \) is as large as the time required to execute the program \( T_C \) and \( V \) might be required to read the dataset \( D \) in its entirety. Commitment schemes based on cryptographic hash functions [BCC88] are commonly used to compute a short immutable “fingerprint” \( cm_t \) for the state at time \( t \) of a large dataset \( D_t \) [BCC88]. Typically \( cm_t \) is negligible in length\(^2\) compared to \( D_t \), and may be easily posted on a block-chain to serve as a public notice\(^3\). Thus, the CI solution we seek should have scalable verification, one in which verification time and communication complexity scale roughly like \( \log T_C \) and \( |cm_t| \) (the bit-length of \( cm_t \)), rather than like \( T_C \) and \( |D_t| \); at the very least verification time/communication should be strictly less than \( T_C \) and \( |D_t| \).

When the dataset \( D \) contains confidential data, the naïve solution can no longer be implemented and

\(^1\)This problem is also known as delegation of computation [GKR08], certified computation [CMT12] and verifiable computation [GGP10].

\(^2\)Commonly, \( cm_t \) is the SHA2 hash of \( D_t \) which is 256 bits long for any dataset length.

\(^3\)A recent report by the World Economic Forum mentions several use cases, among them monitoring blood diamonds and curbing human trafficking [Hut16].
the party \( P \) in charge of \( D \) may conceal violations of computational integrity under the veil of secrecy. Prevailing methods for enforcing CI over confidential data rely on a “trusted party”, like an auditor or accountant to naïvely verify the computation on behalf of the public. This solution still offers no scaling, much like when the data is public. Worse still, it requires the public to trust a third party, which creates a potential single point of failure in the protocol, as this third party can be breached, bribed, or coerced by malicious parties.

Zero knowledge (ZK) proof and argument systems are automated protocols that replace human auditors as means of guaranteeing computational integrity over confidential data for any efficient computation\(^4\), eliminating corruptibility and reducing costs [GMR89]. A ZK system \( S \) for a computation \( C \) is a pair of randomized algorithms, \( S = (P, V) \); the prover \( P \) is the algorithm used to prove computational integrity and the verifier \( V \) checks such proofs. The completeness and soundness of \( S \) imply that \( P \) can efficiently prove all truisms but will fail to convince \( V \) of any falsities (with all but negligible probability). The very first theoretical constructions of ZK systems with scalable verifiers for general computations\(^5\), discussed in the early 1990s, were based on Probabilistically Checkable Proofs (PCP).

The celebrated PCP/IOP Theorem [BFL90, BFLS91, AS98, ALM+98] offered a surprising trade-off between the running time spent by the prover constructing the proof (\( T_P \)), and the running time consumed by the verifier checking it (\( T_V \)); this trade-off means proving time increases polynomially compared to naïve computation time (\( T_P = O(1) T_C \)) whereas verification time decreases exponentially with respect to it (\( T_V = \log O(1) T_C \)).

A ZK system based on the PCP/IOP Theorem (ZK-PCP) [Kil92, Mic00, DFK+92, KPT97, IMSX15] has three additional advantages that are essential for ongoing public trust in computational integrity. First, the assumptions on which the security of these constructions is founded are: (1) the existence of collision-resistant hash functions [Kil92] for interactive solutions. (2) Common access to a random function\(^6\) (the “random oracle model” (FS86)) for non-interactive ones [Mic00]. These are not known to be susceptible to attacks by large-scale quantum computers; we call such solutions post-quantum secure. The anticipated increase in scale of quantum computers [Cou17] and the call for post-quantum cryptographic protocols, e.g., by the USA National Institute of Standards and Technology (NIST) [Che16], highlight the importance of a post-quantum secure ZK solution.

Second, ZK-PCPs/IOPs are proof of knowledge (POK) systems, or, when realized as described above, argument of knowledge (ARK) systems [BCC88, BG93]. Informally, in the context of the DPM example, a ZK-ARK is a proof that convinces the public that the Police has used “the true” dataset \( D_t \) and Presidential Candidate DNA profile \( p \) whose commitments were previously announced.

Third, and most importantly, ZK-PCPs/IOPs are transparent, meaning that the randomness\(^7\) used by the verifier is public; in particular, setting up a ZK-PCP/IOP requires no external trusted setup phase, in contrast to newer ZK solutions, including the one used by the Zcash™ cryptocurrency. Transparency

\(^4\)In the interactive oracle proof model that we consider, as in the model of multi-prover interactive proofs, ZK proof systems exist for any language in nondeterministic exponential time (\( NEXP \)) [BGG+88, BCF+16].

\(^5\)Special cases for ZK, like proving membership/non-membership in a hidden-and-committed set, the “ZK-set” problem, are efficiently solved by other cryptographic means [MRK03]

\(^6\)Even though the random oracle model, per se, is unattainable, it’s use is prevalent in cryptography and the theoretical justification for it discussed, e.g., in [BR93] and following works.

\(^7\)Randomness is necessary for ZK proof systems for non-trivial computations [Gol11, Section 3.2].
is essential for ongoing public trust because it severely limits the ability of even the most powerful of parties $P$ to abuse the system, and thus transparent systems are ones which the public may reliably trust as long as there exists something unpredictable in the observable universe.

Summarizing, ZK-PCPs/IOPs are an excellent method for ensuring public trust in CI over confidential data, as they possess six core virtues: (i) **transparency**, (ii) **universality** — ZK-PCPs/IOPs apply to any efficient computation $C$, even if it requires auxiliary (and possibly confidential) input like $D_t$ above, (iii) **confidentiality (ZK)** — do not compromise auxiliary inputs like $D_t$, (iv) **post-quantum security**, (v) **proof/argument of knowledge**, and (vi) **scalable verification**. Although ZK-PCPs/IOPs have been known since the mid-1990’s, none have been realized in code thus far because, in the words of a recent survey [WB15], “the proofs arising from the PCP theorem (despite asymptotic improvements) were so long and complicated that it would have taken thousands of years to generate and check them, and would have needed more storage bits than there are atoms in the universe.” Consequently, recent realization efforts of ZK systems for general computations (surveyed in full article [BBHR18b]) focused on alternative techniques that do not achieve all of (i)–(vi), though some are extremely efficient in practice for concrete circuit sizes and for amortized computations.

### 1.2 From theory to practice

This thesis is a part of a series of works that show realizations of a transparent ZK system (ZK-STARK, Zero-Knowledge Scalable Transparent ARguments of Knowledge) in which verification scales exponentially faster than database size. Moreover, this exponential speedup in verification is observed concretely for meaningful and sequential computations. In this thesis, we focus on the first part of the construction (See Chapter 4), encoding the transition function as an algebraic problem. A transition function is a well-known concept describing how a computation evolves from a current machine state to the next state. One can also think of the transition function as a relation, such that two machine states belong to the relation if and only if they were, in fact, two consecutive states during the execution of the computation. We translate the problem of checking if two machine states belong to the relation induced by the transition function into a problem that checks whether a set of polynomials is set to zero over a field $\mathbb{F}$. Such an encoding is called an **arithmetization**, and it is a key component in many theoretical and practical results. The key idea behind arithmetization is that the polynomials can be assigned with myriad assignments over a field, and not only with the possible machine state values.

Our intention, in this work, to develop efficient solutions, tailored for some specific real-world problems, for example DPM. These solutions are based on two familiar concepts taken from the literature: **interactivity** and **algebraic properties**. To demonstrate their usefulness, we shall consider the following examples: (1) For interactivity, inversion operation is considered difficult to compute [GS13], but much easier to be verified: Given $a \in F_{2^n}$, finding $a^{-1}$ is complex, but given $a, b \in F_{2^n}$, the verification is easy: just verify that the polynomial $a \cdot b + 1$ is set to zero. (2) For algebraic properties, Fermat’s Little Theorem [Lis11] can be utilized to verify if an element is a member of a certain field. An efficient arithmetization is of major importance in making PCPs/IOPs practical, because it has a direct effect on both the prover’s and verifier’s running time.
1.3 Our results

We begin by introducing arithmetization, and how it can be used in ZK-STARK for verifying computation on a given dataset. We present the parameters determining the complexity of a prover and a verifier. Thereafter, we focus on our efficient arithmetization systems for cryptographic primitives, such as AES and SHA2, then we show the difference in their complexity because the former has an algebraic presentation, unlike the latter. We present methods based on algebraic properties which achieve a better performance. For example, using linearized polynomial for multiplying by matrix, and implementing bit-extraction without unpacking, by using properties of field trace (see Section 3.3.1). Afterwards, we present the construction of arithmetization systems for search programs in various data structures. Following these arithmetizations, we show our solution to DPM. Finally, we explain briefly the construction of our IOP implementation, called ZK-STARK, and the role of arithmetization in ZK-STARK.
Chapter 2

Arithmetization

2.1 Arithmetization Case Study: The DNA Profile Match Problem

As a proof-of-concept “meaningful” computation, we construct a ZK-STARK for the DNA profile match (DPM) problem, which we describe informally next (see Section 3.2 for details). This computation addresses the following hypothetical scenario: Suppose that the Police (acting as the prover $P$) is in charge of the national forensic DNA profile database ($D$), and at previous time $t$ has posted (say, on a block-chain) a hiding commitment $cm_t$ to the state $D_t$ of the database at that point in time. The Police now claims that the DNA profile $p$ of the soon-to-be-appointed and alleged-to-be-corrupt Presidential Candidate, does not appear in $D_t$ and thus wishes to create, in a scalable manner, a proof that will convince the public that the DPM computation was carried out correctly, and the output reported by the Police is correct (with respect to $p$ and $D_t$).

Before we continue describing the method of representing person by DNA profile, we must know what locus is: A locus [JWS09] is a specific, fixed position on a chromosome where a particular gene or genetic marker is located. Each chromosome carries many genes, with each gene occupying a different position or locus. The prevailing standard for DNA profiles, used in over 50 countries, is the Combined DNA Index System (CODIS) format; according to this standard an individual is represented by the Short Tandem Repeat (STR) count of his/her DNA, measured for a set of $20^1$ “core loci” [Fed17].

The commitment $cm_t$ to the state $D_t$ of a CODIS database is assumed to be public information (say, published at time $t$ on a blockchain), as is a commitment $cm_p$ to the profile $p$ of the Presidential Candidate; we assume $p$ was extracted by an independent laboratory that handed it (confidentially) to the Police while publishing $cm_p$ publicly. Assume that the Police declares:

“$\alpha$ is the result of the match search for the profile with commitment $cm_p$ in the database with commitment $cm_t$” (*)

The answer $\alpha$ is one of three possibilities: “no match”, “partial match”, or “full match”. The public (open source) computation $C$ is the one that would have been executed by a trusted third party verifying the claim above. This computation requires three public inputs, $cm_t, cm_p,$ and $\alpha,$ and two

---

1The number of core loci increased from 13 to 20 starting January 2017.
confidential inputs: a DNA profile database $D'$ and individual DNA profile $p'$. The computation $C$ terminates successfully if and only if the public inputs $(cm_t, cm_p, \alpha)$ and the confidential ones $(D', p')$ satisfy three conditions: (i) the result $cm_t'$ of commitment computing of the confidential input $D'$ equals the public input $cm_t$; (ii) the result $cm_p'$ of commitment computing of the confidential input $p'$ equals the public input $cm_p$; and (iii) the output of the match search for the confidential input $p'$ in the confidential dataset $D'$ leads to the publicly announced outcome $\alpha$.

Our ZK-STARK attempts to mimic the computation that would have been carried out by a trusted third party, as presented in the previous paragraph. To understand, informally, why the public should trust a ZK-STARK proof generated for this claim, consider the opposite case, in which the true output of a match search for $p$ on $D_t$ is $B \neq A$ and the Police attempts to construct a ZK-STARK “proof” for a falsity. The binding property of the commitment scheme implies it is infeasible to find any $D'$ other than $D_t$ with a commitment equaling $cm_t$, or to find $p'$ other than $p$ with commitment $cm_p$. Therefore, the Police must choose to either execute $C$ with auxiliary inputs $D_t$ and $p$ which violates condition (iii) above, or feed into $C$ some $D'$ and/or $p'$ that lead to $A$ but which violate (i) and/or (ii) above. In either case, $C$ does terminate successfully based on the combination of public and private inputs that it receives. The soundness property of the ZK-STARK now implies it is infeasible\(^2\) to construct a ZK-STARK proof that will be accepted by $V$. The cryptographic assumptions on which this analysis is based are transparent ones, hence the public should conclude that it is unlikely for the claim being proved to be false. The ZK property of a ZK-STARK means (informally) that the proof reveals no meaningful information about $D_t$ and $p$, beyond what can be surmised from the claim, and thus preserves their confidentiality.

See Section 3.2.5 for details.

2.2 A General Overview on Arithmetizaion

Many ZK systems, including ours, use arithmetization, a technique first\(^3\) used to prove circuit lower bounds [Raz87, Smo87], then adopted to interactive proof systems [BF91, LFKN92]. Arithmetization is the reduction of computational problems to algebraic problems, that involve “low degree” polynomials over a finite field $\mathbb{F}$; in this context, “low degree” means degree that is significantly smaller than field size.

The starting point for arithmetization in all proof systems is a computational integrity statement which the prover wishes to prove, like

\[
\text{“} \alpha \text{ is the result of executing } C \text{ for } T \text{ steps on (public) input } x \text{”} \quad (**) \]

Notice the DPM statement (*) is a special case of (**)}. For our ZK-STARK, and for related prior systems [BS08, BGH+05, BBC+16], the end point of arithmetization is a pair of Reed-Solomon (RS) proximity testing (RPT) problems. In the following sections we introduce our new ZK-STARK; we}

\(^2\)The soundness error below is $err = 2^{-60}$, meaning the probability of success in an interactive setting is $\leq err$ and non-interactively requires $\approx 1/err$ computational effort to generate.

\(^3\)Earlier reductions, such as the one used in Gödel’s Incompleteness Theorem, involved infinite algebraic domains, in particular the natural numbers [Göd31].
discuss its scalability first, based on a new solution to the RPT problem, and afterwards we explain the arithmetization process in more detail.

2.3 Arithmetization I — Algebraic Intermediate Representation (AIR)

Having discussed its end point, we return to describe the innovative components of our ZK-STARK within the arithmetization process itself. The arithmetization is comprised of several phases that are similar to other program and circuit compilation processes, so we borrow terminology used there and adapt it to our process.

The first phase of arithmetization is that of constructing an algebraic intermediate representation (AIR) of the program $C$. Informally, the AIR is a set

$$\mathcal{P} = \left\{ P_1(\vec{X}, \vec{Y}), \ldots, P_s(\vec{X}, \vec{Y}) \right\}$$

of low degree polynomials with coefficients in $\mathbb{F}$ over a pair of variable sets $\vec{X} = (X_1, \ldots, X_w)$ and $\vec{Y} = (Y_1, \ldots, Y_w)$ that represent respectively the current and next state of the computation\(^4\) (see Section 4.2 and Theorem 4.2.1 for more details).

The AIR defines the transition relation of the computation $C$ in the sense that a pair $(\vec{x}, \vec{y}) \in \mathbb{F}^w \times \mathbb{F}^w$ corresponds to a single valid transition (or “cycle”) of $C$ if and only if

$$P_1(\vec{x}, \vec{y}) = \ldots = P_s(\vec{x}, \vec{y}) = 0,$$

i.e., if and only if $(\vec{x}, \vec{y})$ is a common solution of the AIR system $\mathcal{P}$. The following parameters of $\mathcal{P}$ determine prover and verifier complexity, so minimizing them is a major goal of this phase (See Section 2.4 for more details about bottleneck for prover complexity). The degree of the AIR is $\deg(\mathcal{P}) = \max_{i=1}^{s} \deg(P_i)$; the (state) width is the number of variables ($w$) needed to represent a state; the (AIR) size is the number of constraints ($s$), and the cycle count is the number of cycles needed to execute $^5 C$ in naïve execution, meaning, the length of execution table; when the program processes a large number ($n$) of data elements, as is the case for the DPM benchmark, we are interested in the number of cycles per element, denoted $c$; the total cycle count for $n$ elements is $c \cdot n$. If the computation is “expanded” to a circuit as commonly done in other solutions, the cycle count is a lower bound on circuit depth; for the sake of comparison with those other systems, we compute in the rightmost column of Figure 2.1 the total number of multiplication gates for this expanded circuit, as this measure along with circuit depth, are the complexity measures that dictate prover and verifier complexity.

A major contributor to prover complexity in our benchmarks is the cost of proving computational integrity of repeated invocations of a cryptographic hash function; other computations are negligible compared to this cost. Thus, choice of the particular hash function ($H$) is of great importance, as is its definition in terms of $\mathcal{P}$. Our ZK-STARK uses the binary (characteristic 2) field $\mathbb{F}_{2^64}$ because (i) it has efficient arithmetic operations (e.g., addition is equivalent to exclusive-or) and (ii) its algebraic

---

\(^4\)This informal description omits, for simplicity, the boundary conditions, like public inputs and outputs of the computation.\(^5\)In general, this number may depend arbitrarily on the particular input, however, in all our benchmarks it depends linearly on the size ($n$) of the input dataset.
structure is needed for the FRI3rd protocol [BBHR18b], used in ZK-STARK to achieve prover arithmetic complexity that is strictly linear, and verifier arithmetic complexity that is strictly logarithmic. Therefore, the cryptographic hash function we seek is one that is “binary field friendly”, meaning, informally, its AIR has small complexity parameters when defined over binary fields. Figure 2.1 summarizes the main AIR complexity parameters for the DPM benchmark described in Chapter 1 and for three hash functions: the Secure Hash Algorithm 2 (SHA2) family [PUB12] and the Davies-Meyer [Win84] hash based on the Rijndael block cipher [DR99] with 128 bits (AES128+DM) and with 160 bits (Rij160+DM).

Rijndael The hash function used in our benchmarks is the Davies-Meyer construction of a cryptographic hash function from a block cypher [Win84] applied to the Rijndael block cypher with a block-size of 160 bits (corresponding to 80 bits of security). We choose this hash function for two reasons. First, members of this block cypher family were selected in 2001 as the Advanced Encryption Standard (AES) by the U.S. National Institute of Standards and Technology (NIST). Second and most important, this family is “binary field friendly” because the cypher operates on elements of the finite field $F_{2^8}$ which is a subfield of the field used by our ZK-STARK. For example, the Rijndael S-box is essentially inversion in $F_{2^8}$ and the MixColumns step is a linear transformation over that field, which we implement using $F_{2^8}$-linearized polynomials (cf. [LN97, Section 3.4]) over $F_{2^6}$. The “binary field friendliness” of the Rijndael family leads to better parameters than those of SHA2, as can be seen in the bottom rows of Figure 2.1, as well as in the composition degree increase (CDI) discussed later (cf. Section 2.4 and Figure 2.2).

We leave the problem of further improving the efficiency of binary field friendly cryptographic primitives (like hash functions and block ciphers) to future work.

2.4 Arithmetization II — Algebraic Linking Interactive Oracle Proof (ALI)

The main bottleneck for prover time and space complexity is the cost of performing polynomial interpolation and its inverse operation — multi-point polynomial evaluation. The complexity measure that dominates this bottleneck is the maximal degree of a polynomial which the prover must interpolate and/or evaluate; for a computation on a dataset of size $n$ denote this degree by $d_{\text{max}}(n)$. Prior state-of-the-art [BS08, BCGT13, CA15, BBC+16] gave

$$d_{\text{max}}^\text{old}(n) = n \cdot c \cdot w \cdot d + n \cdot c \cdot s. \quad (2.1)$$
which leads to concretely large values (see first column of Figure 2.2). Our ZK-STARK reduces $d_{\text{max}}$ to

$$d_{\text{ZK-STARK}}(n) = n \cdot c \cdot d$$

which results in a multiplicative savings factor of $6.5 \times 10^4$–$1.8 \times 10^5$ over prior works (see the last two columns of Figure 2.2). The improved efficiency of our ZK-STARK is due to two reasons, explained next. The first one completely removes the second summand of (2.1) and the second one removes $w$ from its first summand.

**Algebraic linking IOP (ALI)**  The second summand of (2.1) arises because our prover needs to apply a “local map” induced by the AIR system $P$ (see [BCGV16] for a discussion of “local maps”). Prior state-of-the-art systems, like [BBC$^+$16], used a local map that checks each constraint of the AIR separately, leading to this second summand. Instead, our ZK-STARK uses a single round of interaction to reduce all $s$ constraints to a single constraint that is a random linear combination of $P_1, \ldots, P_s$. This round of interaction completely removes the second summand of (2.1).

**Register-based encoding** The naïve computation performed by the prover can be recorded by an execution trace, a two-dimensional array with $c \cdot n$ rows and $w$ columns, in which each row represents the state of the computation at a single point in time$^5$ and each column corresponds to an algebraic register tracked over all $c \cdot n$ cycles. Prior systems, like [BBC$^+$16], encoded the full execution trace by a single Reed-Solomon codeword, leading to degree $n \cdot c \cdot w$; this degree is then multiplied by $d$ to account for application of the afore-mentioned “local map” to the codeword, resulting in the first summand of (2.1). Our ZK-STARK uses a separate Reed-Solomon codeword for each register$^6$, leading to $w$ many codewords, each of lower degree $n \cdot c$. At first glance this tradeoff may seem wasteful, because we now have to solve an RPT problem for each of these $w$ codewords. However, the interaction and use of randomness allowed by the IOP model once again come to our aid: it suffices to solve a single RPT problem, applied to a random linear combination of all $w$ codewords. The use of a single codeword per register also helps with reducing communication complexity.

Figure 2.2 compares the $d_{\text{max}}$ value of our ZK-STARK to that of the prior state of the art [BS08, BCGT13, CA15, BBC$^+$16] and shows a multiplicative reduction factor of 103–159 for the computations discussed in Section 2.3 and Figure 2.1.

<table>
<thead>
<tr>
<th></th>
<th>$d_{\text{old}}(1)$</th>
<th>$d_{\text{ZK-STARK}}(1)$</th>
<th>$d_{\text{old}}(1)/d_{\text{ZK-STARK}}(1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA2</td>
<td>6323922</td>
<td>41382</td>
<td>153</td>
</tr>
<tr>
<td>AES128+DM</td>
<td>39504</td>
<td>384</td>
<td>103</td>
</tr>
<tr>
<td>Rij160+DM</td>
<td>49996</td>
<td>464</td>
<td>108</td>
</tr>
<tr>
<td>DPM</td>
<td>78988</td>
<td>496</td>
<td>159</td>
</tr>
</tbody>
</table>

Figure 2.2: The maximal degree for old construction and for ZK-STARK.

---

$^6$For simplicity, the current description discusses the case of space bounded computations; the case of computations with large space also uses multiple codewords but the reduction is more complicated.
Chapter 3

Polynomial Constraints for Real-World Problems

3.1 overview

In this chapter we present the heart of the work, in which we use both known arithmetization techniques and several others developed in this work to implement PCP/IOP programs, and in this manner make PCP/IOP practical and efficient. The main part of this chapter presents our implementation of a PCP/IOP for the DNA profile matching problem, the problem we present on Chapter 1. Our solution utilizes the algebraic properties of AES, Fermat’s Little Theorem, linearized polynomials, and other algebraic methods. Thus, we detail many parts of our implementation (see Section 3.2 for details).

Afterwards, we present PCP/IOP implementations for several additional problems. However, we do not present the implementations for these problems in full; instead, we only elaborate on interesting techniques used and the efficiency of these implementations. A partial list of those programs is listed hereafter:

- **SHA2** The popular Secure Hash Algorithm 2 (SHA2) family [PUB12] requires the implementations of modular addition and cyclic shifts, which are not particularly “binary field friendly”, and is thus sub-optimal in this regard. Nevertheless, we construct a rather efficient AIR for it (first row of Table 2.1) using field-specific constraints. A notable example is a constraint system that “extracts” the $i$-th bit from $\alpha \in \mathbb{F}_2^t$ for any $i \in [t]$; this system uses only a pair of constraints of degree 2 (notice the number of constraints and their degree is independent of $t$); we believe this bit-extraction constraint set may be useful for other problems (see Section 3.3.1 for details).

- **DB matching — Merkle Tree & Hashchain** Similar to the DNA profile match problem, we want to prove that some value exists/does not exist in a database. Our approach provides a solution to the problem when the databases are implemented with either Merkle Trees or Hashchains. We focus our discussion on the Merkle Tree variant of this problem; nevertheless, both solutions are detailed thereafter (see Algorithm 2 and Figure 3.3 for a solution to the Hashchain variant of this problem).
A Merkle Tree [Mer88] is a tree in which each leaf is labeled with the hash of a data block, and each inner node is labeled with the cryptographic hash of the labels of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures. We have implemented two programs for this problem: Blacklist, which intends to prove that the given input value does not exist in the database, and Whitelist, which aims to prove that the input value exists in the database. In the former, we traverse the tree from each leaf to the root; when we reach a new leaf, we ask the prover the hash of its neighbor leaf. Note that we may trust the hash sent by the prover because the roothash is public and is thus comparable. See Algorithm 3 and Figure 3.9 for more implementation details. In the latter problem, we need only one path from a leaf to the root. See Algorithm 4 and Figure 3.10. For both programs one can choose either SHA2 or a function based on AES160+DM\(^1\) as the hash function.

- **PoW** A Proof-of-Work (PoW) system, also known as a Proof-of-Work protocol or Proof-of-Work function, first presented in [DN92], is a consensus mechanism. The system gives as output some piece of data which is difficult (costly, time-consuming) to produce but easy for others to verify and which satisfies certain requirements. Some applications of this idea can be used for showing how much processing power some computer has, or as a method to preventing email spam, requiring a proof of work on the email’s contents (including the To address), on every email. One popular system, used in Hashcash [Bac02], uses partial hash inversions to prove that work was done. For instance, the program receives as input a message. In each iteration, we append the output from the previous iteration to a randomly-sampled value, and evaluate some predetermined hash function on the concatenated input, such that after some number of iterations we get a hash value that begins with \(z\) binary zeros. To achieve this, roughly \(2^z\) hash computations need to be evaluated.

### 3.2 DNA profile matching

Let us describe the algebraic intermediate representation (abbrev. AIR, see Section 2.3) of the DNA profile match (DPM) program, that uses the Rijndael-160 based hash function. This exposition will show how we achieve the quantities that are specified in Tables 2.1 and 2.2 (i.e., the width \(w\), the cycles \(c\), etc.), by providing a “bottom up” description of our algebraic construction.

We first recount the Rijndael block cipher, but from an algebraic perspective. Then, we explain our AIR constraints for the Rijndael cipher (see Sections 3.2.1 and 3.2.3). Following that, we describe the AIR for the transformation from a block cipher to a cryptographic hash function (Section 3.2.4). Finally, we show our implementation of the AIR of the logic of the DPM program, that performs an exhaustive search to compare the loci pairs that are stored in the elements of a hashchain (Section 3.2.5).

The Advanced Encryption Standard (AES) instantiates Rijndael with 128-bit block size and 128-bit, 192-bit, or 256-bit key sizes. We denote by Rijndael-160 the cipher with 160-bit block size and 160-bit key size, and hence output (cipher-text) size of 160 bits. Assuming that Rijndael-160 is an ideal cipher

\(^{1}\)DM (Davies-Meyer) is detailed on Section 3.2.4 as a part of our DNA profile matching solution.
field elements are represented according to a standard basis consists of the following four steps, except for the last round that skips Step 3: 

Let $K$ be the field element that resides in the register $00$ during the first cycle. It should be noted that Rijndael with 192-bit (256-bit) block and key sizes can be used to build CRHF with a security parameter of 96 bits (128 bits), and that these stronger parameters entail a rather mild overhead in our algebraic construction (see Table 2.1). However, the STARK construction that we benchmark has 60 bits of security, and therefore the stronger hash functions will not provide better security with our benchmarked system.

Notation. Let $g$ denote a primitive element of $F_{2^64}$, i.e., $<g> = F_{2^64}^*$. We assume throughout that field elements are represented according to a standard basis $(1, g, g^2, g^3, \ldots, g^{63})$, rather than a normal basis. We denote by $R_t \in F_{2^64}$ the content of the algebraic register $R$ at cycle $t$ of the execution (e.g., $K00$ is the field element that resides in the register $K00$ during the first cycle).

### 3.2.1 Rijndael overview

The input to the Rijndael cipher can be regarded as a plain-text array of $4n$ elements and a key array of $4n$ elements, such that each element resides in $F_{2^8}$. Rijndael executes in $n + 6$ rounds, where each round consists of the following four steps, except for the last round that skips Step 3:

- **SubBytes** Given byte $x$, compute $y = Mx^{-1} + b$, where $M \in F_{2^8}^{8 \times 8}$, $b \in F_{2^8}^{8 \times 1}$ are constants.

- **ShiftRows** For $i \in \{1, 2, 3, 4\}$, perform $i$ cyclic shifts (rightwards) of the $i$-th row of the plain-text matrix.

- **MixColumns** For $j \in 1, 2, \ldots, n$, multiply the $j$-th column of the plain-text matrix by the following constant circulant MDS matrix:

\[
\begin{bmatrix}
P0[j]_{t+1} \\
P1[j]_{t+1} \\
P2[j]_{t+1} \\
P3[j]_{t+1}
\end{bmatrix} = \begin{bmatrix}
g_0 & g_1 & 1 & 1 \\
1 & g_0 & g_1 & 1 \\
1 & 1 & g_0 & g_1 \\
g_1 & 1 & 1 & g_0
\end{bmatrix} \begin{bmatrix}
P0[j]_t \\
P1[j]_t \\
P2[j]_t \\
P3[j]_t
\end{bmatrix}.
\]

Here, $g_0$ is a specific field element of $F_{2^8}$ with $<g_0> = 51$, and $g_1 \triangleq g_0 + 1$ generates $F_{2^8}^*$. We note that MixColumns also can be defined as computing the following linear combinations:

\[
\begin{align*}
P0[j]_{t+1} &= g_0 \cdot P0[j]_t + g_1 \cdot P1[j]_t + P2[j]_t + P3[j]_t \\
P1[j]_{t+1} &= P0[j]_t + g_0 \cdot P1[j]_t + g_1 \cdot P2[j]_t + P3[j]_t \\
P2[j]_{t+1} &= P0[j]_t + P1[j]_t + g_0 \cdot P2[j]_t + g_1 \cdot P3[j]_t \\
P3[j]_{t+1} &= g_1 \cdot P0[j]_t + P1[j]_t + P2[j]_t + g_0 \cdot P3[j]_t
\end{align*}
\]

- **AddRoundKey** Key-Scheduler, which computed as follows:

  - Using $Rcon(t) \triangleq g_0^{t-1}$, the first column in the new key-matrix is computed according to:
\[
\begin{bmatrix}
K00_{t+1} \\
K10_{t+1} \\
K20_{t+1} \\
K30_{t+1}
\end{bmatrix}
= \begin{bmatrix}
\text{SubBytes}(K14_t) + K00_t + \text{Rcon}(t) \\
\text{SubBytes}(K24_t) + K10_t \\
\text{SubBytes}(K34_t) + K20_t \\
\text{SubBytes}(K04_t) + K30_t
\end{bmatrix}.
\]

- The other key elements are computed as: \(K[i, j]_{t+1} = K[i, j - 1]_{t+1} + K[i, j]_t\).
- The new key is added by combining each byte of the current plain-text with the corresponding byte of the key, using bitwise exclusive-OR.

### 3.2.2 Implementation technique of the Rijndael cipher

Per our complexity measures (cf. Section 2.3), we wish to construct an efficient representation of a hash function by using algebraic constraints. However, the Rijndael cipher is computed over \(\mathbb{F}_{2^8}\) with field operations modulo the irreducible polynomial \(x^8 + x^4 + x^3 + x + 1\), while the operations in our IOP system are over \(\mathbb{F}_{2^64}\), defined using a different primitive polynomial. The properties of finite fields entail that for any field \(\mathbb{F}_{pm}\) and \(k|m\), there exists a subfield \(\mathbb{F}_{pk}\). Therefore, there is an isomorphism between \(\mathbb{F}_{2^8}\) and a subfield of \(\mathbb{F}_{2^64}\).

**Isomorphism** We obtain such an isomorphism by mapping a primitive element of \(\mathbb{F}_{2^64}\) to a primitive element of the subfield with \(\mathbb{F}' \cong \mathbb{F}_{2^8}\), so that the mapping is implied by their powers. This is done by finding an element of \(\mathbb{F}_{2^64}\) with order \(2^8 - 1\).

Given \(F = \mathbb{F}_{2^m}\) and \(x \in F\), we need to find a primitive element of the subfield \(F' : |F'| = 2^k\) where \(k|m\), and \(F'\) is a field over known irreducible polynomial with degree \(k\), \(p(x)\).

Note, that \(x\) is not a primitive element in the field used on Rijndael. Actually, the multiplicative order of \(x\) is 51. So, we are using \(x + 1\), which its multiplicative order is 255, and therefore, it is a primitive element.

We start from \(g = x^{2m-1}\). The order of \(g\) is \(2^k - 1\), because \(g^{2^k-1} = x^{2^m-1} = 1\), but we can not know that \(g + 1\) is a root of the known irreducible polynomial \(p(x)\). However, we know that \(F' \setminus \{0\} = \{g^i\}_{i=0}^{2^k-2} = \{1, \ldots, 2^k - 1\}\) and one of them is a root of \(p(x)\). So we just need to go over all powers of \(g\), and check if \(ord(g^i) = 2^k - 1\), and if \(p(g^i + 1) = 0\), when we arrive such \(i\), we can say we have found a primitive element in \(F'\) over irreducible polynomial \(p(x)\). Let denote this power of \(g\) as \(g'\).

Now, we can define an isomorphism \(T : \mathbb{F}_{2^n} \leftrightarrow \mathbb{F}' : a^j \rightarrow (g^j)^j\), \(j = 1, \ldots, 255\), when \(a\) is the primitive element of \(\mathbb{F}_{2^n}\). We represent the isomorphism by a matrix \(M\) which its columns are \(g^0, g^1, \ldots, g^7\), so \(T(a) = M \cdot a\). When we have such an isomorphism, we can transform all constants needed for Rijndael to their representation at \(F'\), and then do all operators at \(F'\) since the transformation between fields save the operators.

We then transform all constants needed for Rijndael to their representation in \(F'\), and perform all the field operations in \(F'\). Importantly, this enables an efficient constraint for the SubBytes step of Rijndael, since we can represent the field inverse via a single multiplication in \(F'\). Specifically, by using an auxiliary element \(z \in F'\), the constraint \(y = x^{-1}\) can be represented via \(y \cdot z = x\). By contrast, a naïve implementation of the inverse operation would require auxiliary elements \(\{b_i\}_{i=0}^{7}\), booleanity constraints \(\bigcup\{b_i(b_i + 1)\}_{i=0}^{7}\), and a polynomial of degree 8 with 256 summands.

16
Rijndael steps  The full SubBytes S-box is defined according to \( x \rightarrow M \cdot x^{-1} + b \), where \( M \in \mathbb{F}_2^{8\times 8} \) and \( b \in \mathbb{F}_2^{8\times 1} \) are constants. Adding the constant \( b \) is a simple field addition in \( F' \), whereas the multiplication by the constant matrix \( M \) can be represented using a linear transformation \( T : \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n} \). Using algebraic properties, we have that any linear transformation can be represented by a linearized polynomial [LN97, Chapter 3.4]. We obtain the linearized polynomial \( C(x) = \sum_{i=0}^{7} c_i x^{2^i} \) by finding coefficients \( \{c_i\}_{i=0}^{7} \) that satisfy \( C(a_i) = b_i \), where \( (a_0, a_1, \ldots, a_7) \) is a basis for the domain of \( T \) and \( (b_0, b_1, \ldots, b_7) \) is a basis for the range of \( T \).

While the degree of \( C(x) \) is 128, the degree of our constraint polynomial for the entire Rijndael-160 computation is in fact only 8. At the high-level, the degree reduction is achieved with 3 auxiliary field elements, using an AIR such as \( \{z + C_3(x), z' + C_2(z), z'' + C_1(z')\} \), represent a decomposition \( C(x) = C_1(C_2(C_3(x))) \). Per Section 2.4, this AIR is translated into a single constraint \((z = C_3(x)) \land (z' = C_2(z)) \land (z'' = C_1(z'))\), where the logical-AND is accomplished using the ALI protocol, i.e., random coefficients that are picked by the verifer and sent to the prover in the next round of interaction (this round is used simultaneously for zero-knowledge masking). For better efficiency, the exact implementation uses repeated squaring/quadrupling of the coefficients \( \{c_i\}_{i=0}^{7} \), rather than the polynomial composition \( C_1(C_2(C_3(x))) \).

The ShiftRows operation is implemented together with SubBytes, by simply placing the results of the SubBytes S-box in the appropriate registers for the next cycle (cf. Section 3.2.3). The MixColumns operations is implemented in a single cycle, using the linear combination that we described above to perform field additions and multiplications by the constant \( g_0 \). The AddRoundKey operation is done at the same cycle that we compute MixColumns, using the aforementioned efficient SubBytes S-box implementation.

3.2.3  State machine of the Rijndael cipher

The algebraic execution trace of the Rijndael-160 hash function is shown in the following table. We enforce boundary constraints on the first and last rows (i.e., cycle 0 and cycle \( T \)).

<table>
<thead>
<tr>
<th>P00 ··· P34</th>
<th>K00 ··· K34</th>
<th>INV1 ··· INV5</th>
<th>W11 ··· W53</th>
<th>F1</th>
<th>F2</th>
<th>RC</th>
<th>INVRC</th>
<th>STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P00 ··· P34</td>
<td>K00 ··· K34</td>
<td>INV1 ··· INV5</td>
<td>W11 ··· W53</td>
<td>F1</td>
<td>F2</td>
<td>RC</td>
<td>INVRC</td>
<td>STATE</td>
</tr>
<tr>
<td>P00 ··· P34</td>
<td>K00 ··· K34</td>
<td>INV1 ··· INV5</td>
<td>W11 ··· W53</td>
<td>F1</td>
<td>F2</td>
<td>RC</td>
<td>INVRC</td>
<td>STATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P00 ··· P34</td>
<td>K00 ··· K34</td>
<td>INV1 ··· INV5</td>
<td>W11 ··· W53</td>
<td>F1</td>
<td>F2</td>
<td>RC</td>
<td>INVRC</td>
<td>STATE</td>
</tr>
</tbody>
</table>

The 20 registers

\[
\begin{bmatrix}
P00 & P01 & P02 & P03 & P04 \\
P10 & P11 & P12 & P13 & P14 \\
P20 & P21 & P22 & P23 & P24 \\
P30 & P31 & P32 & P33 & P34
\end{bmatrix}
\]

contain the 160 bits of the plain-text;

17
\[(INV_{1t}P00_t + 1)(P00_t \land INV_{1t}) \land \\
(W11_t + INV_{1t}) \land (W12_t + W11_t) \land (W13_t + W12_t) \land \\
(P00_{t+1} + c_0 \cdot INV_{1t} + c_1 \cdot INV_{1t}^2 + c_2 \cdot W11_t + c_3 \cdot W11_t^2 \land \\
+ c_4 \cdot W12_t + c_5 \cdot W12_t^2 + c_6 \cdot W13_t + c_7 \cdot W13_t^2 + b)\]

Figure 3.1: Constraints polynomial for the Rijndael-160 SubBytes S-box.

The 20 registers

\[
\begin{bmatrix}
K00 & K01 & K02 & K03 & K04 \\
K10 & K11 & K12 & K13 & K14 \\
K20 & K21 & K22 & K23 & K24 \\
K30 & K31 & K32 & K33 & K34
\end{bmatrix}
\]

contain the 160 bits of the key.

Each of these 40 registers is a field element of \(F_{2^9}\) that resides in the subfield \(F' \cong F_{2^8}\), and hence contains only 8 bits of information. Per Section 3.2.1, this is done to support a native inversion operation for the SubBytes S-box. In each cycle, the registers \(INV_1, INV_2, INV_3, INV_4, INV_5\) are used primarily as the auxiliary field elements that compute the inverses for SubBytes.

For \(i \in \{1, \ldots, 5\}\), the registers \(W_i1, W_i2, W_i3\) store the repeated quadrupling that are used to compute the powers of \(INV_i\): \(W_i1 = INV_i, W_i2 = W_i1^4 = INV_i^4, W_i3 = W_i2^4 = W_i1^16 = INV_i^{64}\). Our constraints will then also square these registers, for example \(W_i3 \cdot W_i3 = INV_i^{128}\).

The registers \(F1, F2\) are inner flags that specify the current step in the Rijndael loop. Every round of Rijndael takes 4 steps, and our algebraic constraints use the values of \(F1, F2\) to enforce the requirements of the current step.

The register \(RC\) is used to compute \(Rcon(i)\) in round \(i\) of the Rijndael loop. The register \(INVRC\) is used for the inverse of \(RC\), in order to tell when to stop the Rijndael iterations. The register \(STATE\) is an external flag that specifies whether we compute the Rijndael cipher or some additional logic (i.e., \(STATE\) would be unnecessary for a single invocation of Rijndael-160).

We provide an excerpt of the algebraic constraints of a single SubBytes S-box in Figure 3.1.

Overall, the width of the computation is 65, per the above description. The Rijndael-160 cipher requires 11 rounds where each round consists of SubBytes, ShiftRows, MixColumns, and AddRoundKey (except for the last round that lacks MixColumns). Each round takes 5 cycles in our implementation, hence an entire invocation of Rijndael-160 takes 55 cycles. The prover needs to compute a total of \(55 \cdot 65 = 3575\) field elements for a single invocation of Rijndael-160.

### 3.2.4 From encryption to hash function: Davies-Meyer

The Rijndael-160 block cipher can be converted into a hash function by using the Davies-Meyer transformation: \(hash(B, K) = E_K(B) \oplus B\), where \(E\) is the Rijndael-160 cipher in our case. The resulting \(hash(B, K)\) is collision-resistant under the assumption that for any key \(K\) the function \(E_K(\cdot)\) is an independent random permutation, see [KL14, Theorem 6.5]. Let us also note that constructions with additional overhead can give a CRHF under milder assumptions (e.g., that \(E_K(\cdot)\) is a pseudo-random...
permutation), see for example [BÖS11, Table 3].

To implement the Davies-Meyer construction, the execution trace (cf. Section 2.3) requires saving the 160 bits of $B$ while computing the output of the Rijndael-160 cipher. As discussed, our implementation expands $B$ into 20 registers that contain elements of $\mathbb{F}_{2^{64}}$. Since each of these 20 registers holds only 8 bits of information, we can compress and save $B$ in 3 registers that hold 64 bits of information.

The compression method is quite simple: by treating $\mathbb{F}_{2^{64}}$ as an extension field of $\mathbb{F}_{2^8}$, we set a basis of the extension field and encode 8 registers of $B$ into an element of $\mathbb{F}_{2^{64}}$ by using them as the coefficients of the basis elements. It can in fact be proved that $(1, g_0, g_0^2, \ldots, g_0^7)$ is such a basis. Thus, the encoding is done as $B_0 = \sum_{k=0}^{7} p_k \cdot g_k^0$, where the values $p_k$ are taken from the registers $P_{ij}$.

In order to feed the output digest into the next Rijndael-160 invocation (see Section 3.2.5), we require the output to be in the “uncompressed” form that spans 20 registers. Thus, for $i \in \{0, 1, 2\}$, we decompress $B_i$ after the Rijndael-160 cipher computation, and then add the uncompressed values to the cipher’s output. This is done by letting the prover supply 20 field element \{p_k\}_{k=0}^{19} nondeterministically (with $p_k = 0$ for $k \in \{20, 21, 22, 23\}$), and enforcing the algebraic constraint $B_i + \sum_{k=0}^{7} p_k+8i \cdot g_k^0 = 0$ for $i \in \{0, 1, 2\}$. However, this linear combination is unique only if the coefficients $p_k$ reside in $\mathbb{F}'$, and therefore the verifier must also validate $p_k \in \mathbb{F}'$ for $k \in \{0, \ldots, 19\}$. This is achieved using Fermat’s little theorem, which states that for any finite field $F$, $\forall x \in F : x^{|F|} = x$. In our case the constraints are $\bigcup \{p_k^{256} + p_k = 0\}_{k=0}^{19}$, and we again reduce their total degree to 8 by using repeated quadrupling.

The state machine thus requires 3 additional registers for every cycle:

```
B0  B1  B2
... ...
B0_t B1_t B2_t
B0_{t+1} B1_{t+1} B2_{t+1}
... ...
```

We show the crux of the algebraic constraints for compression and decompression in Figure 3.2.

```
<table>
<thead>
<tr>
<th>Constraint polynomial for compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_{0_{t+1}} + P00_t + x \cdot P10_t + x^2 \cdot P20_t + x^3 \cdot P30_t$</td>
</tr>
<tr>
<td>$+ x^4 \cdot P01_t + x^5 \cdot P11_t + x^6 \cdot P21_t + x^7 \cdot P31_t$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint polynomial for Fermat’s little theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(INV2_t + INV1^4) \land (INV3_t + INV2^4) \land (INV4_t + INV3^4) \land (INV1_t + INV4^4)$</td>
</tr>
</tbody>
</table>
```

Figure 3.2: Constraint polynomials for compression and decompression
The compression requires one cycle before each computation of the Rijndael-160 cipher, and the decompression requires two cycles afterwards. Thus, the number of cycles for the Rijndael-160 hash function is 58, the width is 68, and the total number of field elements that the prover needs to compute is \(58 \cdot 68 = 3944\).

### 3.2.5 DNA profile match (DPM)

The high-level pseudo-code of the DNA profile match is given as Program 1. The database records are assumed to reside in a hashchain of \(N\) elements, as illustrated in Figure 3.3. More precisely, the hashchain is computed using the Merkle-Damgard construction with Rijndael-160 as the compression function. The hash of the last element commits to the entire database, and is verified as a boundary constraint. Notice that we harness the power of nondeterminism to supply the values of the chain elements during the exhaustive search, which implies that for an arbitrary \(N\) the program can operate with only a small constant number of auxiliary variables. Due to the collision resistance of the underlying compression function and the Merkle-Damgard construction [Mer90, Dam89], this use of nondeterminism is secure.

An element of the chain is a DNA profile according to the Combined DNA Index System (CODIS) format; it is comprised of Short Tandem Repeat (STR) counts for 20 "core loci"; we use an 8 bit integer to record a single STR value, and we encode the integer by a single element of \(F\) (cf. Section 3.2.1). Since a single DNA profile requires 20 pairs of STR values (2 per loci), each record (profile) is stored in two consecutive elements of the hashchain. Thus, a database \(D(n)\) of \(n\) profiles requires \(N = 2n\) chain elements, and Program 1 consists of logic that alternates between odd and even elements.

To validate that the prover does not skip over some prefix of the chain in the exhaustive search, the total number of hash invocations \(N\) is also checked by the verifier as a boundary constraint. We also note that Program 1 increments its counter via a field multiplication with the generator \(g\) of \(F_{2^64}^\ast\), thereby avoiding integer arithmetics.

The register \(T\) stores the output, and is verified in the last cycle via a boundary constraint. The output can be either (1) "perfect match", meaning that an exact match between the input (i.e., the commitment \(cm_p\) that the prover decommits in ZK into 20 STR pairs) and a profile in the hashchain was found, or (2) "partial match", meaning that the exhaustive search found a profile in the hashchain such that \(\forall j \in \{1, \ldots, 20\}\) at least one STR value of its \(j\)th pair matches a value of the \(j\)th STR pair of the input, or (3) "no match".

Since the prover (but not the verifier) knows the decommitment of \(cm_p\), the first invocation of Rijndael-160 in Program 1 is executed with completely nondeterministic inputs (that occupy 40 registers), and the output is constrained to be equal to \(cm_p\). Due to the ZK guarantee of our proof system,
Program 1 DNA profile match

Explicit inputs: \( n, cm_t, cm_p \)

Nondeterministic inputs: \( \{\text{VAL}_{i,j}\}_{i \in \{1,2\}, j \in \{1,2,..,20\}}, \{W_j\}_{i \in \{0,1,2,..,n\}} \)

if \( cm_p \neq \text{hash160}() \{\text{VAL}_{1,j}\}_{j \in \{1,2,..,20\}}, \{\text{VAL}_{2,j}\}_{j \in \{1,2,..,20\}} \} \) then
    return false
end if

\( k \leftarrow 1, \ flag \leftarrow 0, \ h \leftarrow 0, \ T \leftarrow 0, \ N \leftarrow 2n \)

while \( k \neq g^N \) do
    Parse \( (L_1, R_1, L_2, R_2, \ldots, L_{10}, R_{10}) = W_j \{ j = \log_2 k \} \)
    if \( \text{flag} = 0 \) then
        \( T_1 \leftarrow \text{CheckPairs}(\text{VAL}_{1,1}, \text{VAL}_{1,2}, L_1, R_1, \text{VAL}_{2,1}, \text{VAL}_{2,2}, L_2, R_2, \ldots, \text{VAL}_{10,1}, \text{VAL}_{10,2}, L_{10}, R_{10}) \)
    else
        \( T_2 \leftarrow \text{CheckPairs}(\text{VAL}_{11,1}, \text{VAL}_{11,2}, L_1, R_1, \text{VAL}_{12,1}, \text{VAL}_{12,2}, L_2, R_2, \ldots, \text{VAL}_{20,1}, \text{VAL}_{20,2}, L_{10}, R_{10}) \)
        \( T \leftarrow \text{MatchingResult}(T_1, T_2, T) \)
    end if
    \( h \leftarrow \text{hash160}(h, W_j) \)
    \( k \leftarrow g \cdot k \)
    \( \text{flag} \leftarrow 1 - \text{flag} \)
end while

if \( cm_t \neq h \) then
    return false
else
    return \( T \)
end if

this implies semantic security (by contrast, comparing the database records to 40 explicit constants that commit to STR strings is not semantically secure because the same value may appear more than once, especially if Program 1 is executed multiple times with the same committed database \( cm_t \)). In each of the next invocations, those nondeterministic values should be compared against the current database record (i.e., 20 registers that are supplied nondeterministically by the prover). However, keeping the initial 40 values throughout the entire execution of Program 1 is inefficient. Since each of those 40 registers contains only 8 bits of information, we use the same technique as in Section 3.2.4 to compress this data into 5 auxiliary registers.

\[
\begin{array}{cccccc}
\text{L0} & \text{L1} & \text{L2} & \text{L3} & \text{L4} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\text{L0}_t & \text{L1}_t & \text{L2}_t & \text{L3}_t & \text{L4}_t \\
\text{L0}_{t+1} & \text{L1}_{t+1} & \text{L2}_{t+1} & \text{L3}_{t+1} & \text{L4}_{t+1} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\text{L0} & \text{L1} & \text{L2} & \text{L3} & \text{L4}
\end{array}
\]

After each invocation of Rijndael-160, we decompress these 5 registers in order to execute the comparison logic (cf. Figures 3.4,3.5,3.6,3.7) for STR pairs with the newly supplied database record. Here too we must verify the decompressed registers by using Fermat’s little theorem, and this requires two cycles as there are not enough temporary registers to compute the repeated quadrupling in a single cycle.
CheckPairs(VAL_{1,1}, VAL_{1,2}, L_1, R_1, VAL_{2,1}, VAL_{2,2}, L_2, R_2, \ldots, VAL_{10,1}, VAL_{10,2}, L_{10}, R_{10})

t ← 2
for \( j \in \{1, \ldots, 10\} \) do
    if (VAL_{j,1} = L_j) AND (VAL_{j,2} = R_j) then
        Continue
    end if
    if (VAL_{j,1} = R_j) AND (VAL_{j,2} = L_j) then
        Continue
    end if
    if (VAL_{j,1} = L_j) OR (VAL_{j,1} = R_j) OR (VAL_{j,2} = L_j) OR (VAL_{j,2} = R_j) then
        \( t \leftarrow 1 \)
    else
        return 0
    end if
end for
return \( t \)

Figure 3.4: CheckPairs subroutine of Program 1.

Program 1 also requires an extra register for the counter \( k = g^j \), two registers for handling the perfect/partial match constraints, three more registers that handle the possible states, and two additional auxiliary registers. Overall, the width of the witness is 81, the number of cycles is 62, and the total number of field elements that the prover computes is \( 81 \cdot 62 \cdot N = 5022 \cdot N \).

MatchingResult(\( T_1, T_2, T \))

if \( T_1 = T_2 = 2 \) then
    return 2
end if
if \( (T_1 = 0) \) OR \( (T_2 = 0) \) then
    return \( T \)
end if
if \( T = 0 \) then
    return 1
end if

Figure 3.5: MatchingResult subroutine of Program 1.
\[(\text{PAIR}_1 + \text{values}_{i,0}) \cdot (\text{PAIR}_1 + \text{values}_{i,1}) \cdot [\]
\[(\text{PAIR}_2 + \text{values}_{i,0}) \cdot (\text{PAIR}_2 + \text{values}_{i,1}) \cdot \text{NewFlag}_t \land \]
\[\text{LastFlag}_t \cdot (\text{NewFlag}_t + 1) \cdot [\]
\[(((\text{PAIR}_2 + \text{values}_{i,0}) \cdot \text{INVTMP}_1) + 1) \land \]
\[(((\text{PAIR}_2 + \text{values}_{i,1}) \cdot \text{INVTMP}_2) + 1)] \land \]
\[(\text{PAIR}_2 + \text{values}_{i,0}) \cdot (\text{PAIR}_2 + \text{values}_{i,1}) \cdot [\]
\[\text{LastFlag}_t \cdot (\text{NewFlag}_t + 1) \cdot [\]
\[(((\text{PAIR}_1 + \text{values}_{i,0}) \cdot \text{INVTMP}_3) + 1) \land \]
\[(((\text{PAIR}_1 + \text{values}_{i,1}) \cdot \text{INVTMP}_4) + 1)] \land \]
\[(\text{NewFlag}_t + \text{LastFlag}_t) \cdot [\]
\[(((\text{PAIR}_1 + \text{values}_{i,1}) \cdot \text{INVTMP}_4) + 1) \cdot (((\text{PAIR}_2 + \text{values}_{i,0}) \cdot \text{INVTMP}_1) + 1) \land \]
\[(((\text{PAIR}_1 + \text{values}_{i,1}) \cdot \text{INVTMP}_3) + 1) \cdot (((\text{PAIR}_2 + \text{values}_{i,1}) \cdot \text{INVTMP}_2) + 1)] \land \]
\[(\text{LastFlag}_t + 1) \cdot (\text{LastFlag}_t + X) \cdot \text{NewFlag}_t \]

Figure 3.6: Algebraic constraints for one pair in the CheckPairs subroutine.

\[\text{InnerMatch}_t \cdot (\text{InnerMatch}_t + 1) \cdot (\text{Match}_t + X) \land \]
\[\text{Match}_t \cdot (\text{Match}_t + 1) \cdot (\text{Match}_t + X) \land \]
\[\text{InnerMatch}_t \cdot (\text{InnerMatch}_t + 1) \cdot (\text{Match}_t + X) \cdot (\text{Match}_t + 1) \land \]
\[(\text{InnerMatch}_t + 1) \cdot (\text{InnerMatch}_t + X) \cdot (\text{Match}_t + 1) \cdot (\text{Match}_t + X) \cdot \text{Match}_{t+1} \]

Figure 3.7: Algebraic constraints for the MatchingResult subroutine.
\[ k_{\text{prev}} = \sum_{i=1}^{w} a_i x^i \]
\[ \bigcup_{i=1}^{w} \{ a_i (a_i + 1) \} \]
\[ a_1 b_1 + (a_1 + 1)(b_1 + 1) \]
\[ \bigcup_{i=2}^{w} \{ (a_{i-1} + 1)b_{i-1}(a_i + b_i + 1) + ((a_{i-1} + 1)b_{i-1} + 1)(a_i + b_i) \} \]
\[ k_{\text{new}} = \sum_{i=1}^{w} b_i x^i \]

Figure 3.8: Polynomial constraints for decrementing an integer.

3.3 SHA2

3.3.1 Bit Extraction

We start discussing in Section 3.1 the problem of bit extraction as a difficult step in SHA-256 implementation. Here we provide notation and several basic lemmas, which facilitate the bit extraction technique that our efficient SHA-256 hash function implementation is based on:

**Lemma 1.** The \( i \)th coefficient in the standard basis representation of \( y \in \mathbb{F}_{2^m} \) is 0 if and only if there exists \( w \in \mathbb{F}_{2^m} \) such that \( w^2 + w + \alpha_i \cdot y = 0 \), where \( \alpha_i \in \mathbb{F}_{2^m} \) is some field element that depends only on \( i \).

**Proof.** We define a \( \mathbb{F}_2 \)-linear function: \( f_i : \mathbb{F}_{2^m}^m \to \mathbb{F}_2, f_i(y) = \begin{cases} 0, & i \text{th coefficient of } y \text{ is 0} \\ 1, & i \text{th coefficient of } y \text{ is 1} \end{cases} \)

We use a field trace, and some of its properties. Trace of an element \( y \in \mathbb{F}_{2^m} \) is defined as \( Tr_{m|2}(y) \triangleq \sum_{i=0}^{m-1} y^{2^i} \). We recall the relevant properties for the proof:

- For any \( \mathbb{F}_2 \)-linear function \( f : \mathbb{F}_{2^m}^m \to \mathbb{F}_2 \), there exists a field element \( \alpha_f \in \mathbb{F}_{2^m} \) such that \( \forall y \in \mathbb{F}_{2^m} : f(y) = Tr_{m|2}(\alpha_f \cdot y) \).

- For every \( c \in \mathbb{F}_{2^m} \), the equation \( y^2 + y = c \) has solutions in \( \mathbb{F}_{2^m} \) if and only if \( Tr_{m|2}(c) = 0 \). Notice that if \( y_0 \) is a solution of \( y(y + 1) = c \) then \( y_0 + 1 \) is the other solution, since the field characteristic is 2.

From all above, we can deduce that the required \( \alpha_i \) exists.

Furthermore, from the first property we see that it is straightforward to pre-compute the constant \( \alpha_i \) by solving linear equations for the trace of basis elements.

3.3.2 Integer Decrement

We show here how an operation that seems simple, like integer decrement, can be complicated and inefficient in an algebraic system. In Figure 3.8, we can see how an algebraic constraint can decrement
(by 1) an integer $k \in [2^w - 1]$. The input is assumed to be encoded in the field element $k_{\text{prev}}$, and the output will be encoded in the field element $k_{\text{new}}$.

First, we employ an “unpacking” technique, with the help of $w$ auxiliary field elements $\{a_i\}_{i=1}^w$. We extract the bit representation of $k$ by forcing each field elements $a_i$ to satisfy $a_i \in \{0, 1\}$, and then require $\sum_{i=1}^w a_i x^i$ to be equal to the input field element. Note that the values $\{x^i\}_{i=1}^w$ are constants that correspond a standard basis representation of the underlying field $(2^m)$, where $m \geq w$.

The decrement is done by using $w$ more auxiliary field elements $\{b_i\}_{i=1}^w$. We enforce for $i = 1, 2, 3, \ldots$ that each “output” bit $b_i$ is the opposite of the “input” bit $a_i$, until the first time that $b_i = 0$ holds. Afterwards, we constrain $b_i$ to be equal to $a_i$. The high-level constraints that achieve this task are

$$b_1 \neq a_1,$$

$$\forall i \in \{2, 3, \ldots, w\} : \text{if } ((a_{i-1} = 0) \text{ and } (b_{i-1} = 1)) \text{ then } b_i \neq a_i, \text{ otherwise } b_i = a_i.$$

Translated into a set of polynomials that would all evaluate to 0 if and only if the above high-level constraints hold, we obtain the algebraic constraints of Figure 3.8.

For example, if $k_{\text{prev}}, k_{\text{new}}, \{a_i\}_{i=1}^w, \{b_i\}_{i=1}^w$ are elements of $(2^w)$ for $w = 8$, and $k_{\text{prev}} = 01011000$ according to the standard basis representation $\{x^i = 2^i\}_{i=1}^w$, then it must hold that $k_{\text{new}} = 01010111$.

It can be observed that decrementing $k_{\text{prev}} = 0$ will cause a wrap-around, but our boundary constraints will ensure that this is never a concern.

---

**Program 2 Hashchain exhaustive search (blacklist)**

**Explicit inputs:** VALUE, N, HEADHASH

**Nondeterministic inputs:** $\{W_i\}_{i \in \{0,1,2,\ldots,N\}}$

```plaintext
k ← 1, h ← 0
while $k \neq g^N$ do
  if VALUE = $W_j$ then
    return false
  end if
  $j = \log_g k$
  $h \leftarrow \text{hash}(h, W_j)$
  $k \leftarrow g \cdot x$
end while
if HEADHASH = $h$ then
  return true
else
  return false
end if
```

25
Figure 3.9: Exemplary Merkle tree with \( \text{HEIGHT} = 4 \) and \( n = 14 \).
Figure 3.10: Illustration of a Merkle tree membership query.

Program 4  Merkle tree membership query (whitelist)

Explicit inputs: \( \text{VALUE, ROOTHASH} \)

Nondeterministic inputs: \( L, \{W_i, D_i\}_{i \in \{0,1,2,\ldots,L-1\}} \)

\[
\begin{align*}
k &\leftarrow 1, h \leftarrow \text{VALUE} \\
\text{while } k \neq g^L \text{ do} \\
&\quad \text{if } D_k = 0 \text{ then} \\
&\qquad h \leftarrow \text{hash}(h, W_j) \{j = \log_g k\} \\
&\quad \text{else} \\
&\qquad h \leftarrow \text{hash}(W_j, h) \\
&\quad \text{end if} \\
&\quad k \leftarrow g \cdot k \\
\text{end while} \\
\text{if } \text{ROOTHASH} = h \text{ then} \\
&\quad \text{return } \text{true} \\
\text{else} \\
&\quad \text{return } \text{false} \\
\text{end if}
\end{align*}
\]
Chapter 4

From AIR to ZK-STARK

The purpose of this chapter is to detail our IOP constructions, specifically about AIR, which is our focus in this thesis. The existence of a ZK-IOP system for $NEXP$ was already established in [BCGV16, BCF$^+$16]; thus, we focus on concrete efficiency and on a description of the construction realized in code.

For the purposes of the current discussion, an instance of a computational integrity statement, denoted $x$, is specified by (i) a transition relation over a space of machines states and (ii) a set of boundary constraints (like inputs and outputs). A witness to the integrity of $x$ is a valid execution of the computation, given by an execution trace, a sequence of machine states that adheres to both boundary and transition constraints of the computation. Casting CI statements like (**) in this format is a straightforward application of the Cook-Levin Theorem.

![Figure 4.1: The reduction from an AIR instance to a pair of RPT problems, solved using the FRI protocol.](image)

4.1 Overview

Our process has 4 parts (see Figure 4.1):

1. The starting point is a natural algebraic intermediate representation (AIR) of $x$ and $w$, denoted $x_{AIR}, w_{AIR}$. By “algebraic” we mean that states of the execution trace are represented by sequences of elements in a finite field $\mathbb{F}$, and the transition relation is specified by a set of polynomials over variables representing the “current” and “next” step of the execution trace.

2. We reduce the AIR representation into a different one, in which states of the execution trace are “placed” on nodes of an affine graph, so that consecutive states are connected by an edge in that graph. Informally, an affine graph is a “circuit” that has “algebraic” topology. The process of “placing” machine states on nodes of a circuit resembles the process of placement and routing.
which is commonly used in computer and circuit design, although our design space is constrained by algebra rather than by physical reality. We refer to the transformation as the **algebraic placement and routing** (APR) reduction, and the resulting representation is an APR instance/witness pair \((x_{\text{APR}}, w_{\text{APR}})\). This reduction is deterministic on the verifier side, i.e., involves no verifier-side randomness and no interaction; as such, it also has perfect completeness and perfect soundness (The prover uses randomness to obtain zero-knowledge; however, this neither affects completeness nor soundness).

3. The APR representation is used to produce, via a 1-round IOP, a pair of instances of the Reed-Solomon proximity testing (RPT) problem. The instances are defined now by the parameters of the RS code. The two codes resulting from the reduction are over the same field \(\mathbb{F}\) but have different evaluation domains \((L, L_{\text{cmp}})\) and different code rates \((\rho, \rho_{\text{cmp}})\). The witness in this case is a pair of purported codewords \((f(0), g(0))\). The verifier’s randomness in the 1-round IOP is used (among other things) to “link” the numerous constraints of the transition relation into a single (random) one. We thus refer to this step as the **algebraic linking IOP (ALI)** protocol.

4. Finally, for each of the two functions (oracles) \(f(0), g(0)\), we invoke the fast RS IOP of proximity (FRI) protocol from [BBHR18a], and this completes our reduction.

### 4.2 Algebraic Intermediate Representation (AIR)

An algebraic execution trace of a program running for \(T\) steps is represented by a \(w \times T\) array in which each entry is an element of a finite field. A single row describes the state of the computation at a certain point in time, and a single column tracks a “register” and its contents over time. It is straightforward to reduce (instance,witness) pairs for a relation that defines a language \(L \in \text{NTimeSpace}(\hat{T}(n), \hat{w}(n))\) to AIR instances with algebraic execution traces of size \(T(n) = O(\hat{T}(n))\) and width \(w(n) = O(\hat{w}(n))\), see, e.g., [Per17] for examples of such reductions.

**Definition 4.2.1 (Algebraic internal representation (AIR))** The relation \(R_{\text{AIR}}\) is the set of pairs \((x, w) = (x_{\text{AIR}}, w_{\text{AIR}})\) satisfying:

1. **Instance Format:** the instance \(x\) is a tuple \(x = (\mathbb{F}, T, \mathcal{P}, C, \mathcal{B})\) where
   - \(\mathbb{F}\) is a finite field
   - \(T\) is an integer representing a bound on running time
   - \(w\) is an integer representing state width
   - \(\mathcal{P} \subset \mathbb{F}[X_1, \ldots, X_w, Y_1, \ldots, Y_w]\) is a set of constraints. The degree of \(\mathcal{P}\) is \(\text{deg}(\mathcal{P}) \triangleq \max_{P \in \mathcal{P}} \text{deg}(P)\)
   - \(\mathcal{B}\) is a set of boundary constraints, where each boundary constraint is a tuple \((i, j, \alpha)\) for \(i \in [T], j \in [w], \alpha \in \mathbb{F}\)

2. **Witness Format:** The witness \(w\) is a set of functions \(w_1, \ldots, w_w : [T] \to \mathbb{F}\); we say \(w\) satisfies the instance if and only if
(a) For all boundary constraints \((i, j, \alpha)\) we have \(w_j(i) = \alpha\), and

(b) For all \(t \in [T - 1]\) and for all \(P \in \mathcal{P}\) we have \(P(w[t], w[t + 1]) = 0\) where \(w[t] = (w_1(t), \ldots, w_w(t))\).

Finally, \(R_{AIR}\) is the set of all pairs \((x, w)\) such that \(w\) satisfies \(x\), and \(L_{AIR} \triangleq \{ x \mid \exists w, (x, w) \in R_{AIR} \}\).

Our reductions will use AIRs in which \(\mathbb{F}\) is invariably a binary field, and we shall further assume the witness size and degree are “binary friendly”.

### 4.3 Realization considerations

Our focus is on the DPM program that requires small space, hence the former system is more efficient in this case.

For the hash function used to construct Merkle trees as commitments to oracles, we use the Davies-Meyer construction instantiated with AES128. This gives an estimated collision resistance parameter of 64 bits.

Using the notation detailed on [BBHR18b], we set \(R = 3\) and \(\lambda = 60\), leading to security error of at most \(err \leq 2^{-60}\). The binary field we use has \(|\mathbb{F}| = 2^{64}\), i.e., \(n = 64\). The degree of our constraint system is 8 thus \(d = 3\). We fix the ZK parameter to \(k = 1\); this ensures zero knowledge for \(t \geq 8\), and this is obtained for dataset sizes \(n \geq 2^3\) because \(t = \lceil n \cdot 62 \rceil\). For all smaller datasets, setting \(k = 3\) would suffice for ZK but since our focus is on large datasets we did not implement this.

The repetition parameter for the number of FRI-COMMIT is \(s = 2\), and for the \(FRI - QUERY\) we have \(\ell_{f(0)} = 9\) and \(\ell_{g(0)} = 22\).

#### Realizing time and witness-size compression

Let \(|D(n)|\) denote the bit-length of a dataset \(D(n)\) that contains \(n\) profiles (each profile is 40 bytes long); let \(CC(n)\) denote the communication complexity of the ZK-STARK for \(D(n)\), i.e., the total number of bits communicated between prover and verifier; similarly, let \(T_C(n)\) denote the time needed to naively verify \(C\) by executing it on \(D\) with \(n\) entries, and let \(T_V(n)\) denote the time required by \(V\) to verify it, (both measured on a fixed physical computer.)

Consider a computation \(C\) which requires auxiliary confidential input \(D\) that varies in size, like the DPM example. Any ZK-system \(S = (P, V)\) for \(C\) induces a pair of rate measures for time and witness-size, respectively:

\[
\rho_{time}(n) = \frac{T_V(n)}{T_C(n)}, \quad \rho_{size}(n) = \frac{CC(n)}{|D(n)|}
\]

(4.1)

The rate measures (and thresholds defined next) depend on \(C\) and the system \(S\), so the notation \(\rho_{time}^{(S,C)}\) would be more precise, but we prefer notational simplicity and assume \(C\) and \(S\) are known.

A rate value smaller than 1 indicates compression, meaning verification in \(S\) is more efficient than naïve verification. In fully scalable ZK systems verifier complexity is poly-logarithmic in prover complexity. Therefore eventually, for large enough \(n\), the system achieves compression. Our main claim here is that we exhibit, for the first time, time and witness-size compression for a ZK-STARK for a
large-scale sequential computation. Define the compression threshold to be the smallest value \( n_0 \) such that for all \( n \geq n_0 \) the rate is less than 1,

\[
\theta_{\text{time}} = \min \{ n_0 \mid \forall n \geq n_0 \ \rho_{\text{time}}(n) < 1 \}; \quad \theta_{\text{size}} = \min \{ n_0 \mid \forall n \geq n_0 \ \rho_{\text{size}}(n) < 1 \} \quad (4.2)
\]

Figure 4.2 shows the rate measures for the DPM problem on a double logarithmic scale. The time compression threshold is at \( \theta_{\text{time}} = 2.8 \times 10^5 \) and the witness-size threshold is \( \theta_{\text{size}} = 9 \times 10^3 \). The largest database for which we could generate a proof during our tests is \( n_{\text{max}} = 2^{20} \approx 1.14 \times 10^6 \) DNA profiles; larger databases require more disk space and RAM than was available to us. Each profile occupies 40 bytes so \( |D_{n_{\text{max}}}| \approx 43 \) megabytes. The time-rate for \( n_{\text{max}} \) is \( \rho_{\text{time}}(n_{\text{max}}) = 1/6 \) and the witness-size rate is \( \rho_{\text{size}}(n_{\text{max}}) = 1/100 \). This figure also demonstrates that compression will improve if supported by stronger hardware than that on which our tests were executed. (see Section 4.4 for more measurements.)

Figure 4.2: The time (left) and witness-size (right) rate functions of the DPM benchmark ZK-STARK as a function of (i) number of entries (\( n \)) in the database (upper horizontal axis) and (ii) number of multiplication gates (lower horizontal axis).
4.4 Measurements of the ZK-STARK for the DNA profile match

In this section we provide additional raw measurements for proving complexity and verification complexity for the DPM discussed in Chapter 1. We used two separate machines to measure performance: a strong server for the prover, and a “standard” laptop for the verifier.

4.4.1 Prover

Figure 4.3 shows the running time required to generate the ZK-STARK proofs, both in absolute terms (left) and as a multiplicative overhead in running time over naïve computation (middle). On the right we plot the size occupied by the all IOP oracles and Merkle trees used by the prover. Due to space limitations (called the “RAM threshold”), the actual space used by the prover was lower than plotted there but saving space required larger running time. Specifically, notice that at \( n = 2^{14} \approx 16,000 \) the total space (plotted on right) passes the RAM threshold, corresponding to (and explaining) the jump in proving time which is noticed on the middle plot at the same value of \( n \). This jump is due to re-computing parts of the proof oracles on demand, which is required to operate within RAM limits.

The code of the prover (written in C++) has been optimized for large instances and running times, hence it uses multi-threading (MT) intensively. Our use of MT seems empirically quite efficient; in particular, disabling MT incurs a slowdown factor close to \( \times 2 \). The relatively large multiplicative overhead noticeable on the middle plot for small instance sizes is likely explained by the overhead that the use of MT introduces. However, since prover execution time is measured in fractions of a second for these short executions we leave further optimizations of it to future work.

Figure 4.3: On the left we plot, on a double-logarithmic scale, prover running time as a function of the number of entries (\( n \)) in the DNA profile database. On the middle we plot the ratio between proving time to naïve execution time; the horizontal axis is logarithmic (\( \log n \)) and the vertical one measures ratio. On the right we plot, on a double-logarithmic scale, the total size of all oracles and their commit trees (Merkle trees), generated by the prover during execution. See text inline for an explanation for the “phase transitions” seen in the middle plot.

Prover machine specifications

- CPU (2 units) : Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz (24 cores, 2 threads per core)
- RAM : 768GB DDR4 (32GB \( \times 24 \), Speed: 2666 MHz)
- SWAP : 3.2TB NVMe (1.6TB \( \times 2 \))
- Operating System : Red Hat Enterprise Linux 7 (3.10.0-693.5.2.el7.x86_64)
4.4.2 Verifier

Figure 4.4 gives the ZK-STARK verifier running time ($T_V$) on the left, and communication complexity on the right. The verifier is non-adaptive, which means it’s complexity can be measured even for databases sizes $n$ that are too large to generate a proof for. The values for which an actual proof was generated are indicated by full circles in both plots.

The ZK-STARK verifier is comprised of two sub-verifiers. The first is the ZK-STIK verifier that verifies proofs in the “pure” but unrealistic IOP model. The second is the sub-verifier that checks (only) consistency of values residing in various Merkle trees with previously committed Merkle tree roots. In both plots of Figure 4.4 the bottom line indicates the complexity of the ZK-STIK verifier, both for time (left) and communication complexity (right). As evident from the Figure, the ZK-STIK complexity is small relative to overall complexity. Moreover, as oracle size increases, the ratio of STIK/STARK complexity grows smaller. This is because as oracles grow larger, the relevant Merkle trees grow deeper and hence there are more authentication paths, and each is of larger length.

We executed the verifier in single thread mode; the tests run by it are amenable to parallelization and faster execution time. However, since verification time is already quite small we leave these further optimizations to future work. Similarly, we point out that the additional memory consumption required by the verifier is negligible, compared to the communication-complexity. In particular, when the verifier is executed on weaker machines than the one reported here (see specification below), verification complexity does not increase significantly.

![Figure 4.4: Verifier running time (left) and communication complexity (right) as a function of the number of entries ($n$) in the DNA database size; for both plots the horizontal axis is logarithmic and the vertical one is linear. In both plots, the lower line measures the complexity of the ZK-STIK verifier, and the upper line measures that of the ZK-STARK verifier. We stress that verifier measurements are performed for values of $n$ that are (significantly) larger than those for which a proof can be generated; this is possible because our verifier is non-adaptive, thus, we tested it with randomly generated “proofs”. Full circles indicate values of $n$ for which proofs were generated.](image)

Verifier machine specifications

- Model: Lenovo ThinkPad W530 Laptop
- CPU: Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz (4 cores, 2 threads per core)
- RAM: 32GB DDR3 (8GB ×4, Speed: 1600 MT/s)
- Operating System: Arch Linux (4.13.12-1-ARCH)
Chapter 5

Conclusion and Open Questions

5.1 Using arithmetization for making PCPs/IOPs practical

In this work, we described how PCPs/IOPs can be used for verifying computation on a given dataset, without relying on any external trusted party and in a Zero-Knowledge setting. We focused on arithmetization, which is the encoding of a transition function of a computation model as an algebraic problem. We saw how a set of constraints, called AIR, can be built, and later used in ZK-STARK. We started with implementing cryptographic primitives, such as Rijndael, SHA2, etc. We can see now the importance of using algebraic attributes in our ZK-STARK, to reduce the degree of the polynomial constraints, thus achieving better performance. We can see the difference between “algebraic friendly” functions such as Rijndael, and functions which are not, e.g. SHA2.

On top of that we constructed a layer of various data structures such as Merkle-Tree, HashChain, POW. Afterwards, we could give solutions to real-world scenarios, like DPM. This part of our work can show that PCP/IOP may be used in practical scenarios, and not only in theoretical research.

5.2 New directions

Following our aforementioned results and contributions, research can continue this work in several directions: first, developing a new hash-function being thoroughly assessed by cryptanalysts, who consider it as secure as any other standard hash function. This function should be built especially for ZK-STARK, which is, in our configuration, more efficient than any other hash-function. We started examining such functions, based on principles of Rijndael, just with field elements in $\mathbb{F}_2^{64}$, and we got very efficient proofs with roughly 1000 field elements in total. There are already some STARK-Friendly (or semi-friendly) hash functions, such as The MARVELlous family of Cryptographic Primitives [AD18], HadesMimic [AGR+16] etc. A survey of these functions is detailed in [GKK+19]. Second, implementing basic opcodes of any simple processor (e.g., PDP – 11) in ZK-STARK. This allows anyone to easily build programs based on PCPs/IOPs. Of course, these programs cannot be the most efficient. Therefore, we can wrap some previously implemented primitives or functions as new opcodes.
Bibliography


לראות בניית אילוח שמתלע ביט באנדקס מוטיס, הפתרון הנazıיב היה פוליוס מדרגה 2,IZED 64 פוליומיס מדרגה 2 וandez סיקור בבכותרת אלבטה - 20 תארים בנייה של אורות צהוב ע"וי 2 פוליומיס מדרגה 2 וendez הסיקור בכרך לראות ספרון במקום פעיל ע"וי גיבוב. הליך היבטועים של בניית ההכחה וידוא של פונקציה כética צהוב ע"וי המוני ע"וי. נראת שימת פונקציה גיבוב מצורה, אך הנה מורכבות על שימת בדואת סופים, לעומת אאות, נוכל להראותquiry שימת פונקציה גיבוב מצורה,شف הכנסים על צהוב ע"וי הפונקומים מדרגה 2 פולינומים מדרגה 64 כמות הצעדים במחוזנית ניתן לראות שימוש בפונקציות (2). 2 פולינומים מדרגה 64

לראות בנייה של פונקציה גיבוב מובססת שדות סופיים, בוחר והמספר העדדים כיון הקט התרחבה בניית שדות סופיים שמטרתו שדה של הפונקומים, אבל עדין מחוסכים מחוץ לשנייה על צהוב. בנוכם כל אג רשתות ומ Ive בשדות. הבנייה הבולתי העדד - למעלה, הארצות לכל 20 רופי-瞍ים לנד 3 רופי-瞍ים. בככ יבכים
לשתהו ביביאר 17 הרופי-瞍ים לגבוב השיבוים אגרים.

למרות שהמקרה היי בניין פתרון מובסס מחסנית פונקציוניסים לכותת פתרון ביוות מחסנית האמצעי. המקרה מחוור האופטייל地毯 כל הגיוו. המקרה התחל ממוסכים פרימיטיבים קריפטוגרפיים פונקציות גיבוב, חוו בחרה היבטועו פונקציה כולה בטבעת החינוך האלגרביות של הפונקציה, פעולות השביה פונקציה מבני צהוב, ולאחר מכך הריכב על פתרון שלימי שלושים מענה לרצות ביוות

iii49+ 1
למשל, כדי לשמש את הכותרת:Aיתונאסטרטגיפיקאי, we will use the following definition:

1. **Axioms and Premises**:
   - The axioms and premises are the foundational truths of the theory.

2. **Theorem Statement**: The theorem statement is as follows:

3. **Proof**: The proof is constructed as follows:

   - ...

4. **Conclusion**: The conclusion is as follows:

   - ...

The presented work is a continuation of the previous work, and it focuses on the following aspects:

1. **Methodology**: The methodology used in this work is as follows:

   - ...

2. **Results**: The results obtained in this work are as follows:

   - ...

3. **Discussion**: The discussion on the results is as follows:

   - ...

The work is concluded with a summary, which highlights the main findings and contributions of this research.

Technion - Computer Science Department - M.Sc. Thesis - MSC-2020-19 - 2020
תקציר


אריתמטיזציה לטובה הוכחות הנויות
לויידן בbestosברות בפורוטוקול יшир וא
אינטראקטיבי

חיבור על מחקר

לשם מליי חלקי של הדרישות לכבלי התוכנה
منهجי למדעי בנידיע המחשב

יגון חורש

הנהל לטכנולוגיות --- מכון טכנולוגי לישראל
אדר אתשים'ה ייפח מרציה 2020
אריתמטייזציה לטובת הוכחות הניתנות
לمواد בשתיברות בפורמה של יישר או
אינטראקטיבי

יין חורש