Fast Reductions from RAMs to Delegatable Succinct Constraint Satisfaction Problems* (full version)

Eli Ben-Sasson† eli@cs.technion.ac.il Technion

Alessandro Chiesa† alexch@csail.mit.edu MIT CSAIL

Daniel Genkin† danielg3@cs.technion.ac.il Technion

Eran Tromer‡ tromer@cs.tau.ac.il Tel Aviv University

February 17, 2012

Abstract

Succinct arguments for NP are proof systems that allow a weak verifier to retroactively check computation done by a more powerful prover. These protocols prove membership in languages (consisting of succinctly-represented very large constraint satisfaction problems) that, alas, are unnatural in the sense that the problems that arise in practice are not in such form.

For general computation tasks, the most natural and efficient representation is typically as random-access machine (RAM) algorithms, because such a representation can be obtained very efficiently by applying a compiler to code written in a high-level programming language. We thus study efficient reductions from RAM to other problem representations for which succinct arguments are known. Specifically, we construct reductions from the correctness of computation of a T-step non-deterministic random-access machine to:

1. (succinct) circuit satisfiability with $O(\log T)$ overhead, and
2. (succinct) algebraic constraint satisfaction with $O(\log^2 T)$ overhead.

On the latter problem representation, the best known Probabilistically Checkable Proofs can be directly invoked. Our constructions are explicit and do not hide large constants.

To attain these, we develop a set of tools (both unconditional and leveraging computational assumptions) for generically and efficiently structuring and arithmetizing the computation of random-access machines.

Keywords: delegation of computation; succinct arguments; random-access machines; probabilistically checkable proofs

*We thank Ohad Barta and Arnon Yogev for reviewing prior versions of this technical report.

†The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 240258.

‡This work was partially supported by the Check Point Institute for Information Security and by the Israeli Centers of Research Excellence (I-CORE) program (center No. 4/11).

Contents

Contents **2**

1 Introduction **4**
1.1 Motivation 4
1.2 Goals 5
1.3 Programming circuits: from bhRAM to circuit satisfaction 5
1.4 Programming polynomials: from bhRAM to algebraic constraint satisfaction 6

2 Summary of Results **6**

3 Roadmap **7**

4 Related Work **7**

5 Discussion of Results **7**
5.1 Defining SuccinctACSP 7
5.2 Results on Programming Polynomials 8
5.2.1 First route and then arithmetize 9
5.2.2 First delegate memory (and don't route) and then arithmetize 10
5.3 Results on Programming Circuits 11
5.4 Programming Other Models of Computation & Non-Uniformity 12
5.5 Tools 12
5.6 Open Problems 13

6 Definitions **15**
6.1 Levin Reductions 15
6.2 Graphs and Routing 16
6.3 Arithmetic in Finite Fields 17

7 Random-Access Machines **18**
7.1 Informal Discussion 18
7.2 Formal Definitions 19

8 A Generic Succinct Graph Coloring Problem **22**

9 From RAMs to Succinct GCPs **26**
9.1 A Levin Reduction 26
9.1.1 Step 1: From RAMs to computation graphs 27
9.1.2 Step 2: From computation graphs to (double) De Bruijn graphs 30
9.1.3 Step 3: From (double) De Bruijn graphs to succinct GCPs 34
9.1.4 Step 4: The Levin reduction 37
9.2 A Computational Levin Reduction 38
9.2.1 Step 1: From RAM to untrusted RAMs via Merkle trees 39
9.2.2 Step 2: From untrusted RAMs to succinct GCPs 40
9.2.3 Step 3: The computational Levin reduction 43
9.3 Tradeoff Between The Two Reductions 44

10 A Generic Succinct Algebraic Constraint Satisfaction Problem **45**

11 From Succinct GCPs to Succinct ACSPs **49**
11.1 Arithmetizing Double De Bruijn Graphs 50
11.1.1 An embedding and some lemmas for double extended De Bruijn graphs 51
11.1.2 The conversion of parameters for double extended De Bruijn graphs 54
11.1.3 The Levin reduction for double extended De Bruijn graphs 66

A Other Related Work **69**

B Routing on De Bruijn Graphs **70**
B.1 Butterfly Networks and Isomorphic Graphs of Interest . . . 70
B.2 Beneš Networks and Their Rearrangeability . . . 71
B.3 Routing Bit-Reversal Permutations . . . 75
B.4 Simulating Beneš Networks with Butterfly Networks . . . 76
B.5 De Bruijn Graphs and Their Rearrangeability . . . 77

C Circuits **79**
C.1 Transition Function . . . 79
C.2 Constraint Circuit for De-Bruijn Graphs . . . 81
C.3 Constraint Circuit for Cyclic Graphs . . . 85

D Untrusted RAM Lemmas **86**
D.1 Untrusted Memory Lemma . . . 86
D.2 Untrusted Input Lemma . . . 89
D.3 Untrusted Code Lemma . . . 91

E Finite Fields and Efficient Computation **94**
E.1 Irreducible and Primitive Polynomials . . . 94
E.2 Linear Maps and Sparse Polynomials . . . 95
E.3 Polynomial Evaluation . . . 97
E.4 Polynomial Interpolation . . . 98
E.4.1 Existence and uniqueness of low-degree extensions . . . 98
E.4.2 Complexity of the general case . . . 100
E.4.3 Interpolation over linear subsets . . . 100
E.4.4 Linearized interpolation . . . 101
E.5 A Canonical Embedding . . . 102
E.6 Some Useful Families of Polynomials . . . 103
E.7 Efficient Algebraic Computation . . . 105

References **109**

1 Introduction

1.1 Motivation

Cryptographic protocols that offer integrity and confidentiality guarantees for general computation are much needed, e.g., for attaining secure cloud computing and for reducing trust in execution platforms. Recent work in cryptography has focused, for example, on the problem of *delegation of computation* [BCCT11, DFH11, GLR11, CRR11, GGP10, CKV10, AIK10, CT10, Mic00, Kil92], *computing on encrypted data* [Gen09], and *non-interactive arguments of knowledge* [Gro10]. These protocols prove properties about circuits, or (in the case of PCP-based constructions) coloring problems on certain hypergraphs or algebraic constraint satisfaction problems.

This raises the question: how can we *generically and efficiently* convert problems that arise in practice (i.e., executing algorithms) to problems used by the underlying cryptographic protocols, such as the aforementioned? Clearly, if we are going to seriously take the idea of *using*, say, delegation schemes in practice, we must study and understand techniques for generically reducing natural problems that arise in practice to the requisite representations of the underlying protocols. Efficiency of the reduction is paramount since the envisioned applications deal with very large instances.

To restate more broadly, as cryptographers we wish to design *generic and practical* protocols for a variety of security goals. For example, in an ideal world, we would want practical non-interactive zero-knowledge proofs for any problem in NP, practical delegation protocols for any polynomial-time function, practical succinct arguments for any problem in NP, and so on.

Completeness. The notion of *completeness* allows for the construction of *one* protocol for a *whole* complexity class — an enormous design simplification. The choice of a specific complete problem is sometimes made for concreteness, but however other times the choice is *not arbitrary at all*: sometimes it is the case that only for certain complete problems that exhibit certain special properties we are able to design a desired protocol, and we simply do not know how to design similar protocols for many other (possibly much more interesting) problems, without having to go through potentially very expensive reductions.

Without loss of generality, but often with loss in efficiency. Thus, while the choice of a specific complete problem in principle comes without loss of generality, it may come with great loss in efficiency. Such a loss of efficiency may severely limit the applicability of a protocol, even if the protocol itself is quite efficient for certain special complete problems.

For example, in the case of zero-knowledge proofs, we know how to directly construct non-interactive zero-knowledge arguments for circuit satisfiability (e.g., [Gro10]), which is an NP-complete problem. As another example, we also know how to construct succinct arguments [Kil92] for problems for which PCPs with efficient verifiers are known [BSGH$^+$05, BSCGT12]; such PCP-friendly problems usually tend to be NP-complete problems with a strong combinatorial or algebraic flavor (e.g., coloring problems over certain graphs or whether there exist polynomials of a certain degree that vanish everywhere on a certain set).

However, the kinds of problems such as the ones mentioned in the previous paragraph are unnatural; e.g., while the circuit model may be convenient for highly structured computational tasks (such as evaluating Fourier transforms), it is not convenient in general; this is even more true for problems with strong combinatorial or algebraic structure.

In the real world, the problems that we are interested in arise in the form of *algorithms* written in high-level programming languages such as C or Java.

Random-access machines as a natural problem. Thus, we would want our cryptographic protocols to be generic and practical *for problems about random-access machines* [CR72, AV77].

More precisely, consider the following constraint satisfaction problem: the ***bounded-halting problem*** on a two-tape random-access machine M, denoted $\text{BHRAM}(M)$, is the language of all pairs (x, T) such that there is an auxiliary input w for which $M(x, w)$ accepts within T steps. (See Section 7 for formal definitions.)

Focusing on $\text{BHRAM}(M)$ comes at essentially no loss in efficiency, because existing compilers do a splendid job at reducing algorithms, expressed in a high-level programming language, to a sequence of basic instructions on a random-access machine. Hence, having generic and practical protocols for random-access machines gets us most of the way to having generic and practical protocols for any problem. That is, we suggest that BHRAM stands out as the NP-complete problem that best represents a natural starting point.

1.2 Goals

At the highest level, we investigate the following question. For various succinct arguments designed for some NP-complete problem:

> **Question:** Given a random-access machine M, what is the most efficient way to reduce $\textsc{bh}\mathrm{RAM}(M)$ to this NP-complete problem?

Ultimately, the hope is to develop a set of tools for reducing the correctness of computation of (non-deterministic) random-access machines to a variety of popular problems, i.e., problems that are the starting point in many cryptographic protocols. Achieving this hope would bestow great flexibility to protocols that benefit from these reductions, because our ability to easily "program" random-access machines (via compilers) would translate into the ability to easily *and efficiently* "program" less natural problem representations such as circuits or polynomials.

Naturally, investigating the above question will benefit from insights from complexity theory, where the study of reductions between different computational problems is a basic tool. Nonetheless, the reductions we seek are different from the traditional ones in two ways:

- Reductions need not have information-theoretic guarantees, but only **computational** ones. That is, computational reductions (when appropriately defined) will generally suffice. This relaxation may potentially greatly simplify reductions that with only information-theoretic techniques are too inefficient. (In this paper we will see an example of how computational guarantees offer an alternative, with different tradeoffs, to another unconditional reduction.)
- Reductions are between **uniform and succinct** models of computation. For example, to specify the question "is (x, T) in $\textsc{bh}\mathrm{RAM}(M)$?" we only need $|M| + |x| + \log(T)$ bits, but the constraint satisfaction problem it represents (i.e., the correct execution of $M(x, w)$, for some witness w, for at most T steps) has size that is $\Omega(T)$. The reductions we seek must be efficient in $|M| + |x| + \log(T)$ (and not T), and thus must implicitly convert a large yet succinctly-represented constraint satisfaction problem about random-access machines into a large yet succinctly-represented constraint satisfaction problem about, say, circuits or polynomials. This efficiency requirement is necessary, e.g., in delegation of computation where weak parties, and not just powerful ones, must be able to perform the reduction.

Thus, compared to traditional Levin reductions, our reductions are on the one hand easier to obtain in that we may usually settle for computational guarantees, but on the other hand are harder to obtain in that the succinctness requirement (of converting between large succinctly-represented constraint satisfaction problems) poses important additional technical constraints.

In this paper, we concentrate on *two concrete goals* that we feel are particularly urgent. We describe each goal in the following two subsections (Section 1.3 and Section 1.4 respectively).

1.3 Programming circuits: from $\textsc{bh}\mathrm{RAM}$ to circuit satisfaction

The problem of (Boolean) circuit satisfiability is a very popular one; its succinct version is:

Definition 1 (informal – see [Pap94] for precise definition)**.** *Let $\mathcal{C} = \{C_T\}_{T\in\mathbb{N}}$ be a Boolean circuit family (of* AND, OR, *and* NOT *gates) where the wiring information of each gate of the T-th circuit can be computed in* $\mathsf{polylog}(T)$ *time. Define* $\textsc{SuccinctCircSat}(\mathcal{C})$ *to be the language of all pairs (x, T) such that there exists w such that $C(x, w)$ accepts.*

Our first goal is the following:

> **Goal #1: Efficiently program circuits.**
> Namely, reduce $\textsc{bh}\mathrm{RAM}$ to $\textsc{SuccinctCircSat}$ with as tight a reduction as possible.

That is, we need to find an efficient transformation $M \mapsto \mathcal{C}^M$ such that $(x, T) \in \textsc{bh}\mathrm{RAM}(M)$ if and only if $(x, T) \in \textsc{SuccinctCircSat}(\mathcal{C}^M)$. In this case, the overhead of the reduction is naturally measured as the size of C_T^M divided by $S_M \cdot T$, where S_M is the "processor complexity" of M, i.e., the size of the Boolean circuit for the transition function of M.

Progress towards Goal #1 would greatly increase the applicability of protocols that rely on circuit satisfiability; for example, these include zero knowledge arguments such as [AF07] and [Gro10], delegation protocols such as [IKO07] and [IO11], and the SNARKs and proof-carrying data of [BCCT12].[1]

1.4 Programming polynomials: from bhRAM to algebraic constraint satisfaction

Probabilistically-checkable proofs (PCPs) [BFLS91], and the protocols that they enable (most notably succinct arguments [Kil92, Mic00, DCL08, BCCT11, DFH11, GLR11]), are perhaps the best example of powerful results for NP-complete problems that unfortunately rely on very unnatural choices of complete problems. Thus, their applicability (even if assumed that the PCP constructions were practical) remains very limited.

Unlike studying reductions to circuits, studying reductions to problems used in PCP constructions is not as well-defined a problem, because PCPs have been constructed for a variety of (unnatural) problems. Which problem should we pick as a target?

Thus our second goal actually consists of two subgoals: (1) distill a problem definition that captures the essential ingredients that makes an NP-complete problem "probabilistically checkable", and (2) construct as tight reductions as possible for this problem from bhRAM.

As a guide for which problem to pick, we choose to give preference to the kinds of problems for which *short* (i.e., quasilinear size) PCPs are known. This choice is motivated by the fact that a short PCP seems to be a prerequisite (though not a sufficient condition) for practicality of a PCP. Given this constraint, the relevant constructions essentially reduce to the work of Ben-Sasson and Sudan [BSS08] followed by by Ben-Sasson et al. [BSCGT12], whose short PCPs are constructed using algebraic (rather than combinatorial) techniques, and involve testing proximity and checking properties of univariate (rather than multivariate) polynomials. Thus, our second goal is the following:

> **Goal #2: Efficiently program PCP-friendly polynomials.**
> (1) Define a succinct algebraic constraint satisfaction problem, SuccinctACSP, for which (short) PCPs can be constructed.
> (2) Reduce bhRAM to SuccinctACSP with as tight a reduction as possible.

Analogously with Goal #1, we need to find an efficient transformation $M \mapsto \mathsf{par}^M$ such that $(x, T) \in \text{bhRAM}(M)$ if and only if $(x, T) \in \text{SuccinctACSP}(\mathsf{par}^M)$, where $\mathsf{par}^M = \{\mathsf{par}^M_T\}_{T \in \mathbb{N}}$ is a family of parameter choices for SuccinctACSP; that is, par^M_T describes the algebraic constraints representing the instance (x, T). In this case, the cost of the reduction is naturally measured as the size of the field $\mathbb{F}_T$ (induced by par^M_T) divided by $S_M \cdot T$, where again S_M is the "processor complexity" of M as a Boolean circuit.

A decoupling for tackling practical PCPs. The choice of a SuccinctACSP problem representation that is conductive towards reduction from bhRAM was tackled by Ben-Sasson et al. [BSCGT12], in conjuction with their study of practical PCP constructions. These two aspects are related in the sense that the choice of SuccinctACSP reflects a tradeoff between the PCP efficiency and the reduction efficienty. The concrete choice of [BSCGT12] provides a natural and flexible representation that *simultaneously* captures the essential ingredients that make a problem probabilistically checkable as well as general enough to make the problem easy to reduce to from more high-level problems such as bhRAM.

We shall thus focus on tight reductions to SuccinctACSP as defined in [BSCGT12] (and reviewed in Section 5.1).

2 Summary of Results

In this paper we develop tools, leveraging the expressive power of combinatorial and algebraic properties of graphs and polynomials, for tightly embedding generic computation of programs into the computation of circuits and into algebraic constraint satisfaction problems. These tools enable us to prove the following:

Theorem 1 (informal)**.** *There are (unconditional and computational) reductions:*

1. *from* bhRAM *to* SuccinctCircSat *with overhead* $O(\log T)$ *with small constants.*

[1] More precisely, the protocol in [AF07] does not need the succinctness of the reduction, because there the verifier is not weak, but can still benefit from the efficiency of the reduction.

2. from bhRAM *to* SuccinctACSP *with overhead $O(\log^2 T)$ with small constants.*

(Recall that SuccinctCircSat *is the satisfiability problem for succinctly-represented circuits, and* SuccinctACSP *is a satisfiability problem for succinctly-represented algebraic constraints, used by PCPs.)*

With the help of research assistants Ohad Barta and Arnon Yogev, we are in the process of implementing these reductions in code; an implementation report with further optimizations is forthcoming.

3 Roadmap

The rest of this paper is organized as follows.

After briefly reviewing prior work in Section 4, we discuss SuccinctACSP in Section 5.1, the second part of the theorem in Section 5.2 and the first part of the theorem in Section 5.3. We then discuss in Section 5.4 how our results can be used to obtain reductions from other models of computation and non-uniform models of computation. Finally, we pose some open questions in Section 5.6 (for example, relevant to fully-homomorphic encryption) that we believe are quite important to tackle for the cryptography community. Pointers to the appropriate technical sections are given during the corresponding high-level discussions.

4 Related Work

Prior work towards Goal #1. Leveraging the fact that multi-tape Turing machines can sort in quasilinear time [Sch78], Gurevich and Shelah [GS89, Theorem 2] showed that the class of languages non-deterministically accepted by a random-access machine in quasilinear time is equal to the class of languages non-deterministically accepted by a multi-tape Turing machine in quasilinear time. Combining this with the result of Pippenger and Fischer [PF79] that any computation that can be performed by a multi-tape Turing machine in T steps can be performed by an oblivious two-tape Turing machine in $\Theta(T \log T)$ steps, and the fact that the computation of an oblivious Turing machine (say with two tapes) in T' steps can be reduced to a circuit of size $\Theta(T')$, we deduce that there is an $\Omega(T \log^2 T)$ reduction from random-access machines to circuits. This reduction path, unfortunately, does not explicitly address the requirement of succinctness (though plausibly could me made to) and most importantly hides large constants, because it goes through (two) Turing machine reductions.

A better reduction was given by Robson [Rob91], who gave a $\Theta(T \log T)$ reduction from random-access machines to Boolean formulas. However, Robson also did not address the requirement of succinctness, and still hides large constants.

We seek an explicit, "lean", succinct solution to Goal #1, possibly by making the solution even simpler by leveraging computational assumptions.

Prior work towards Goal #2. Reductions from "non-algebraic" satisfaction problems to "algebraic" satisfaction problems are common in the PCP literature; this process is more widely known as *arithmetization*. However, such reductions are usually quite expensive; for example, [Har04, Chapter 5] shows how to arithmetize circuits, but with a *cubic* blow up. To the best of our knowledge, the only "tight" reduction is due to Ben-Sasson et al. [BSGH+05], but is from *Turing machines*. As discussed above, it is not practical to use the result of Gurevich and Shelah [GS89, Theorem 2] to reduce from random-access machines to Turing machines as a first step. So once again we seek a much more efficient solution.

In Appendix A, we discuss more related work.

5 Discussion of Results

We give a high-level discussion of our results; throughout, pointers to the appropriate technical sections are given.

5.1 Defining SuccinctACSP

SuccinctACSP is a class of succinct algebraic constraint satisfaction problems, parametrized by a list of parameters, for univariate polynomials over finite field extensions of GF(2). Each particular SuccinctACSP

problem simultaneously allows for the construction of short PCPs (as shown in [BSCGT12]) and is flexible enough to support (several) fast reductions from bhRAM, as we will show in this paper.

Informally, a choice par of parameters of SuccinctACSP consists of the following:

- A field size function $f\colon \mathbb{N} \to \mathbb{N}$, inducing a finite field family $\{\mathbb{F}_T\}_{T\in\mathbb{N}} := \{\mathrm{GF}(2^{f(T)})\}_{T\in\mathbb{N}}$.
- A family $\{H_T\}_{T\in\mathbb{N}}$ where each H_T is a subspace of $\mathbb{F}_T$.
- A family $\{N_T\}_{T\in\mathbb{N}}$, where each N_T is a set of affine functions.
- A family $\{P_T\}_{T\in\mathbb{N}}$, where each P_T is a constraint polynomial.
- A family $\{S_t\}_{t\in\mathbb{N}}$, where each S_t is a subset of $\mathbb{F}_T$.

A reduction to SuccinctACSP will concretely instantiate a choice of the above parameters.

A pair (x, T) is a member in SuccinctACSP(par) if it fulfills the following: there exists a low-degree assignment polynomial $A\colon \mathbb{F}_T \to \mathbb{F}_T$ that "colors" elements of the field $\mathbb{F}_T$ such that: (1) the constraint polynomial P_T at every element α of the subspace H_T, when given as input the colors in the "N_T-induced affine neighborhood" of α, is satisfied; and (2) the colors of the first $|x|$ elements of the set S_t are consistent with x.

Crucially, for each of the families in par, the T-th object must be able to be describable in time $\mathsf{polylog}(T)$. Additional properties that are essential in [BSCGT12] to construct short PCPs with an efficient verifier include: for example, H_T is a subspace (so that one may leverage the computational properties of *linearized polynomials* [LN97, Section 2.5]), and functions in N_T are affine, that is, have degree equal to 1 (because any degree greater than 1 would cause the composition of a neighbor function with the assignment polynomial to have degree that is at least quadratic).

From the perspective of this paper, what is important is that we have a lot of freedom in choosing what is (and how to use) the subspace H_T, what exactly are the affine functions in N_T (so that we may induce a variety of topologies for a corresponding "affine graph" over the field), what are the properties that the constraint P_T polynomial verifies, and so on. Indeed, in this paper, we shall exercise these degrees of freedoms in different ways for different reductions — and thanks to the decoupling provided by the interface of SuccinctACSP, we can do so without opening up the underlying PCP constructions.

Quantitatively, our goal is to ensure that f grows as slow as possible. Ultimately, as stated in Section 2, we will show two alternate reduction paths where $f(T) = O(T \log^2 T)$ with small constants. Ensuring that all of the relevant information can be computed in $\mathsf{polylog}(T)$ time will be a non-trivial task given the large degrees of the polynomials involved. We shall thus develop a set of computational tools for efficient arithmetization.

The class of problems SuccinctACSP is formally defined in Section 10.

5.2 Results on Programming Polynomials

Having introduced SuccinctACSP, we present two alternate reduction paths from bhRAM to SuccinctACSP with similar overhead but different tradeoffs; namely, we provide two different solutions to the second part of Theorem 1.

SuccinctGCP as a stepping stone. For modularity, we define a "stepping-stone" problem that we call SuccinctGCP; this problem is a class of succinct graph coloring problems (and hence the name) that, just like SuccinctACSP, is parametrized by a list of parameters. Thus each reduction path that we present consists of two steps, where the first step reduces bhRAM to SuccinctGCP and the second step reduces from there to SuccinctACSP (see Figure 1).

We design SuccinctGCP to be a graph coloring problem over a known (yet succinctly-represented) graph topology and with strong "local" properties: each vertex in the graph has a corresponding coloring constraint that can easily be computed from the identity of the vertex, and this coloring constraint only involves the colors of the vertex itself and its neighbors. The structure imposed in SuccinctGCP on the one hand still enables us, when given sufficiently expressive graph topologies, to tightly embed generic computation into the graph, and on the other hand provides key properties that are used in the next step, during the process of arithmetization.

The class of problems SuccinctGCP is formally defined in Section 8.

Proof strategy. Each of the reduction paths consists of two steps:

- **Step 1: "localizing" constraints.** In this step we translate the constraints that determine correctness of computation of a random-access machine to a graph coloring problem over a graph with some known topology, that is, some choice of parameters for SuccinctGCP. This is where we tame the global nature of the RAM's memory access operations, which can reach arbitrarily across memory and need to be consistent with values stored arbitrarily long ago; we capture these by localized graph constraints.

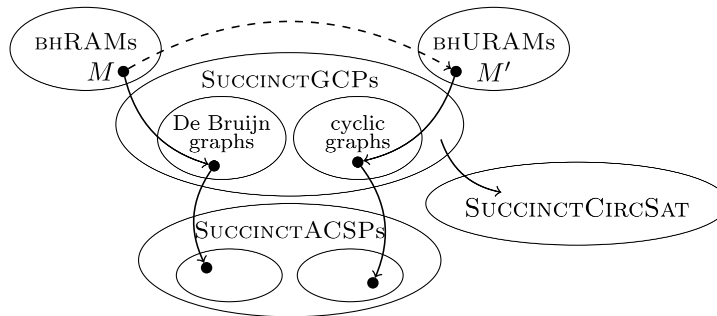


Figure 1: Summary of our reductions. The dashed arrow is the reduction that only has computational soundness. Starting from bhRAM, we present two alternative reduction paths: the one on the left is "first route and then arithmetize" and the one on the right is "first delegate memory (and don't route) and then arithmetize". Also, our SuccinctGCPs can be easily used to "program" Boolean circuits. A more detailed summary is in Figure 2.

- **Step 2: arithmetizing constraints.** In this step we translate the choices of parameters for SuccinctGCP obtained in Step 1 to corresponding algebraic constraints in a finite field, that is, to some choices of parameters for SuccinctACSP.

The two reduction paths (each with two steps) are outlined in Section 5.2.1 and Section 5.2.2, accompanied by Figure 1 and Figure 2.

5.2.1 First route and then arithmetize

The constraints that determine correct computation of a given random-access machine M can be divided into *code-consistency* constraints (i.e., did the machine correctly execute the specified operation?) and *memory-consistency* constraints (i.e., did the machine load a value equal to the last stored value at the given address?).

Step 1. Informally, we say that a graph is a *computation graph* for M if its vertices are named with "time steps" and its edges are divided into "time" and "memory" edges. The first task is to convert the correctness of computation of M to the problem of whether there exists a computation graph for M that can be colored with configurations of M in a way that some local constraint applied to the colors of the vertices of every edge is satisfied; this local constraint will be different depending on whether the given edge is a time or memory edge. Here, deducing a *local* constraint to apply to memory edges that implies the *global* property of memory consistency is a particularly tricky task; we do so by introducing a definition for what it means for a configuration to *precede in memory* (as opposed to precede *in time*) another configuration and prove that the ability to reorder vertices to satisfy this relation implies memory consistency. (See Figure 2(a).)

While phrasing the problem in terms of whether a valid computation graph exists is progress, we are by far not "ready" for arithmetization because the topology of a computation graph is still quite arbitrary: a memory edge can potentially point from any given vertex to any vertex of the "past" (intuitively, the last time step during which the same address was accessed). So, in a sense, while a computation graph "distills" code and memory into local constraints, these local constraints are still not "structured".

To address this problem and provide structure, we use routing networks [Ofm65], which can route any permutation with no congestion. Using routing techniques to structure constraint satisfaction problems for other other models of computation was already used in [BFLS91, PS94, BSS08, BSGH$^+$05]; we extend and adapt these to the case of random-access machines.

Specifically, our routing network uses De Bruijn graphs. These are graphs with $O(\log T)$ columns, each with T elements, that can route T elements from the first column to the last column, according to any given permutation. Here "route" simply means that intermediate columns receive a packet and either forward it to the "up neighbor" or "down neighbor" of the next column; these routing decisions can be efficiently computed given the desired permutation.

At high level, we show how to leverage the permutation properties of De Bruijn graphs, by routing code-consistency and memory-consistency constraints, to obtain a SuccinctGCP problem over De Bruijn graphs that encodes the correctness of computation of a random-access machine. (See Figure 2(c).) More concretely, the reduction works as follows:

1. The computation graph of a T-step random-access machine is "painted" on the first column of a *wrapped* De Bruijn graph, i.e., one where the first and last columns are identified.

2. To tackle memory consistency, we use the permutation properties of the De Bruijn graph to reorder the vertices of the computation graph according to the memory precedence permutation. Informally, we have routed the memory edges of the computation graph, so that now the graph structure is fixed; furthermore, the additional constraints we have to define to check a routing is valid are local, so that all the constraints we need to check now are indeed local.

3. Then, to check code consistency, we attach a second De Bruijn graph to the first, along its first column, and on this second De Bruijn graph we forward the information of each vertex in the computation graph to the next; in this way, when a vertex in the first column receives a packet, which allegedly is the configuration of the previous time step, it can check code consistency agains the locally-available configuration (of the current time step); this can be done by running the transition function of the random-access machine. Informally, we routed the time edges of the computation graph.[2]

Formalizing the above intuition involves many technical details, including some imposed by the requirement that all of the above must be able to be expressed succinctly.

Technical details for this step are given in Section 9.1.

Step 2. We now must arithmetize the SuccinctGCP problem on De Bruijn graphs obtained in the previous step. (See Figure 2(e).) Doing so consist of two subtasks:

1. *Graph Arithmetization.* We need to embed the De Bruijn graph into as small a subspace as possible by defining a map from vertices to field elements and corresponding affine "neighbor functions". The embedding map needs to be carefully engineered so that as few neighbor functions as possible are needed to express all the edges of the De Bruijn graph — the difficulty here lies in that we are only allowed to define *affine* neighbors.

2. *Constraint Arithmetization.* We need to convert the Boolean circuit encoding the local constraints of the SuccinctGCP problem into a polynomial constraint that will constrain the values of a candidate-witness low-degree polynomial A over the affine neighborhood of every element in the subspace.

   A naïve arithmetization will not work here, because it will result in a polynomial with too high a degree, and will make the reduction too inefficient. Instead, we leverage computational properties of linearized polynomials as well as developing new tools for carefully arithmetizing the constraint circuits into corresponding polynomials of small degree.

We thus obtain a choice of parameters of SuccinctACSP (corresponding the to the initial choice of random-access machine). Of course, many more details are needed in order actually specify the reductions; these details mostly have to do with keeping track of how expensive it is to convert, represent, and access the very large objects involved, together with enforcing consistency with the instance. (Indeed, not only do we need to ensure via constraints that some valid computation occurred, but we must also establish that this computation had something to do with the specific instance at hand.)

Technical details for this step are given in Section 11.1.

Efficiency. Each step incurs in a $O(\log T)$ overhead, for a total of $O(\log^2 T)$ overhead.

5.2.2 First delegate memory (and don't route) and then arithmetize

In the alternate reduction path, we avoid routing altogether by first modifying the random-access machine to not "trust" memory, and instead *delegate* its memory to an untrusted storage, so that dealing with memory consistency constraints will be easier. We do so via computational assumptions.

Step 1. The reduction works as follows:

[2]Unlike memory edges, time edges need not be routed to obtain a SuccinctGCP problem because they always follow the simple "+1 permutation". However, not routing them will result in a graph different from a De Bruijn graph, which cannot be arithmetized efficiently. So we do route time edges as well. Also see discussion in Section 5.3.

- We formalize the notion of a bounded-halting problem for random-access machines ***with untrusted memory***, and define a corresponding class that we call BHURAM; essentially, in order to "prove" membership of an instance in the class, it suffices to exhibit a transcript that is merely code consistent but not necessarily memory consistent.

  We then describe a computational reduction from BHRAM to BHURAM: we show how to transform a given random-access machine into a new random-access machine that (instead of assuming that when it loads a value from memory the value is equal to the value it last stored there) will dynamically maintain a Merkle tree [Mer89] over the memory, so that the machine can *itself* verify whether values returned from memory are "good" or not.

- Next, we show an unconditional reduction from BHURAM to a SUCCINCTGCP problem over cyclic graphs; now the reduction is considerably easier, because we do not have to worry about memory consistency (and in particular there is no need to perform routing).

Alternatively, we can think of the first step of the reduction as a computational reduction from random-access machines to log-space random-access machines, followed by an unconditional reduction to SUCCINCTGCP.

Technical details for this step are given in Section 9.2.

Step 2. We now must arithmetize the SUCCINCTGCP problem over cyclic graphs obtained in the previous step. Once again arithmetization must be done carefully to ensure that the final field size is not much larger than the cyclic graph itself. To do so, we re-use some of the techniques developed already for the De Bruijn case (thus demonstrating their flexibility), but we must also engineer other components of the reduction specifically for this case.

Note that while it is possible to reuse the second step of the reduction path described in the previous section (i.e., the arithmetization of SUCCINCTGCP over De Bruijn graphs), this would be an overkill resulting in a loss of $\log T$ factor in the reduction.

Technical details for this step will be given soon in a further version of this technical report. In the meantime, we can simply use the arithmetization of De Bruijn graphs, though we will "pick up" an extra (unnecessary) $\log T$ factor.

Efficiency. Once again each step incurs in a $O(\log T)$ overhead, for a total of $O(\log^2 T)$.

5.3 Results on Programming Circuits

We now discuss solutions to the first part of Theorem 1.

Fortunately, we have already done most of the work to obtain such solutions, because each first step of our BHRAM to SUCCINCTACSP reductions (respectively described in Section 5.2.1 and Section 5.2.2) *already* provides a convenient and generic method to tightly "program" non-deterministic Boolean circuits, i.e., a way to reduce BHRAM to SUCCINCTCIRCSAT.

Any SUCCINCTGCP problem easily induces a Boolean circuit for verifying the problem: namely, the Boolean circuit that verifies (in parallel) all of the coloring constraints, and then takes the conjunction of all of these verification results; the succinctness of SUCCINCTGCP ensures that the Boolean circuit can be represented succinctly. Thus the reduction from SUCCINCTGCP to SUCCINCTCIRCSAT is trivial. Therefore, to reduce from BHRAM to SUCCINCTCIRCSAT, we can either use either of the following:

- Use the first step of the reduction described in Section 5.2.1, which uses routing as the main tool for the reduction to SUCCINCTGCP. Then reduce to SUCCINCTCIRCSAT. In fact, because we do not intend to arithmetize the problem, we can simplify this step further by simply using a Beneš network for the routing [Lei92, Theorem 3.11], instead of a De Bruijn graph, thereby reducing the number of "columns" of the routing network to only $2 \log T$.

- Use the first step of the reduction described in Section 5.2.2, which uses delegation of memory (via Merkle trees) to reduce to SUCCINCTGCP. Then reduce to SUCCINCTCIRCSAT.

  This second option, as it relies on dynamically maintaining untrusted storage, is only computationally sound, but its advantage is that the non-deterministic circuit can be *generated and evaluated on-the-fly* as no routing is needed; in practice, this may constitute an big advantage since for long computations the space complexity of routing is likely to be a bottleneck.

Thus, our reductions at high level imply that:

> *A T-step non-deterministic computation on a random-access machine can be converted to a non-deterministic Boolean circuit with overhead $O(\log T)$ with small constants.*

Because the reduction from SuccinctGCP to SuccinctCircSat is trivial, there are no technical details to provide; all the required "work" is already done when working out the details for the reductions discussed in Section 5.2.

5.4 Programming Other Models of Computation & Non-Uniformity

While our reductions have only addressed bounded-halting problems on RAMs, we can easily address many other computation models, because the RAM model is powerful enough simply "evaluate" those computation models at a very small overhead.

For example, if one is interested in a large non-uniform Boolean circuit, then an elegant way to evaluate it would be to consider the machine M that takes as input a hash digest, and as a witness a (description of a) circuit; M then verifies that the hash of the circuit matches the input, and then evaluates the circuit. In this way the size of the combined machine and input is small, and the non-uniformity is "paid" once and for all by whoever evaluated the hash of the circuit to begin with (possibly during an expensive pre-processing phase).

5.5 Tools

The two essentially different reduction paths that we present in this paper (see Figure 1) demonstrate the flexibility of the problem definition of SuccinctACSP (and, in fact, also of the "stepping stone" problem SuccinctGCP too), because our reductions ultimately instantiate SuccinctACSP with very different choices of parameters. More details about these can be found in the appropriate appendices.

Working towards these constructions, we developed tools that we would like to briefly highlight here.

Efficient algebraic computation. As discussed, the arithmetization of constraints in both reduction paths is an especially delicate process, because one runs the risk of obtaining high-degree polynomials that are either not sparse or cannot be computed efficiently, or, worse, have a degree that is *too* high thereby forcing us to make the field size too large.

Building on the initial work of Ben-Sasson et al. [BSGH+05], we develop a set of algebraic tools that leverages the additive structure of subspaces of finite fields to enable and simplify various problems that arise during arithmetization; many of these insights leverage the computational properties of *linearized polynomials* [LN97, Section 2.5]

For example, essential in both arithmetizations is the ability to evaluate at a few points an "alternator polynomial": namely, a polynomial that, given two subspaces $V, W \subseteq \mathbb{F}$ with $V \subseteq W$, is equal to 1 on V but vanishes everywhere on $W - V$. Such a polynomial is useful in the construction of P_T (the final constraint polynomial) because it allows to "turn on" a given constraint on the subspace V but to specifically turn off the same constraint everywhere on $V - W$, where another constraint will be turned on. An alternator polynomial has degree $|W|$, which is very large; nonetheless, we show how to make a small arithmetic circuit for it, despite it not being sparse.

As another example, when embedding a graph G (such as a De Bruijn graph) defined by a SuccinctGCP problem into an affine graph $\mathbf{G}$ inside a field for the corresponding SuccinctACSP problem, the embedding is not "perfect", because being limited to only affine neighbor relations usually means that the out-degree of $\mathbf{G}$ is greater than the out-degree of G. This presents a problem because, when we attempt to convert a given constraint circuit C_T into a constraint polynomial P_T, we do not know a priori which of the affine neighbors of a given field element α is a "true" neighbor and thus do not know which inputs we should give to C_T. In order to recognize which of the neighbors are legitimate, we may need to learn some information about the *inverse image* of α under the embedding, because given its inverse image we can apply the neighbor function of the original graph G. We show how to do so using sparse representations of "projection polynomials", i.e., polynomials that are able to compute certain bits of the bit representation of a field element.

See Appendix E for more details.

Trading no-routing for computational soundness. Our second, alternate reduction path (described in Section 5.2.2) is not a standard reduction in that it is a *computational* reduction: we avoid the need for routing but settle for computational soundness.

More generally, we show how computational assumptions can be used not only to make memory consistency simpler, but also how to make "input consistency" and "code consistency" simpler. These additional reductions can be used to further improve the efficiency of our reductions by reducing their dependency on the input size and code size.

Furthermore, there are things that we *only* know how to do with computational reductions. For example, in [BCCT11] it is observed that these computational reductions (specifically, the one simplifying "input consistency") can be combined with a SNARK to yield simple memory delegation and streaming delegation schemes [CTY10, CKLR11]. As another example, in [BCCT12] the reduction from BHRAM to BHURAM plays an important role as a tool in the construction of CS proofs in the plain model. We find it an interesting open question to investigate the power of computational reductions to simplify further or obtain results that cannot be achieved with only unconditional ones.

See Appendix D for more details.

5.6 Open Problems

We believe that better understanding reductions from BHRAM to the problems used by cryptographic protocols is an important and exciting research direction. Our work leaves unanswered several intriguing questions:

Achieving $\log T$ overhead. Does a reduction from BHRAM to SUCCINCTACSP inherently require a $\log^2 T$ overhead, or can a reduction with $\log T$ overhead be constructed? Showing a reduction with only $\log T$ overhead would improve the practicality of PCPs.

Deterministic models. All the reductions that we have discussed in this paper are inherently non-deterministic; e.g., we reduce BHRAM(M) to the satisfiability of a certain circuit, *even* if M does not take into consideration any witness (i.e., even if M is a deterministic machine). Indeed, many of the techniques we used can be distilled to the paradigm of carefully choosing what extra information we might "put in the witness" because it is much faster to verify its goodness rather than to generate it from scratch.

This dependency on non-determinism is not surprising: it is known already that non-determinstic models of computation are quite robust under quasilinear time reductions [GS89, NRS94], but such efficient reductions are not believed to exist for the deterministic case. For example, while non-determinism makes Turing machines and random-access machines equivalent up to polylogarithmic factors, random-access machines are not believed to be simulatable, without non-determinism, within polylogarithmic factors on (even multi-tape) Turing machines.

Unfortunately, the dependency on non-determinism means that protocols that do not support non-determinism cannot benefit from our reductions. For example, the circuit-checking techniques of [GKR08] and the cut-and-choose techniques of [CKV10] do not support non-determinism, and it is not clear if they can be extended to do so. The best known reductions from deterministic random-access machines to deterministic circuits are at least $\Omega(T^2)$, which grows too quickly for large problem instances (which are precisely the ones that we are likely to outsource). Thus, the applicability of any protocol not supporting non-determinism is severely limited and it is an interesting open question to improve this situation or prove that such improvements are unlikely.

Reductions for FHE. Following the discussion of the previous paragraph, reductions to non-deterministic models do not seem to help with programming Boolean circuits for use with fully-homomorphic encryption [Gen09] either.

Specifically, even if we reduce from a deterministic algorithm (i.e., a deterministic random-access machine), the output Boolean circuit is still non-deterministic and a worker that only has access to encrypted values of the computation does not seem to be able to produce a valid witness for the Boolean circuit in quasilinear time. (Concretely, how could a worker find the routing? He does not know the memory accesses, because they are encrypted.) Thus, we do not know of a sub-quadratic solution for producing Boolean circuits that can be used for fully-homomorphic encryption.[3]

We believe it is an important open problem to either find quasilinear solutions for fully-homomorphic encryption, or prove that such a solution is unlikely to be found. Until then, the applicability of fully-homomorphic encryption may be limited to sufficiently structured problems or algorithms running with small memory.

[3] A natural attempt would be to convert the given RAM into an Oblivious RAM [GO96] in order to hide the information leaked by (logical) memory accesses, and then evaluate the transition function of the resulting "processor" under fully-homomorphic encryption in order to hide the registers and the Oblivious RAM's secrets. However, in order to efficiencly satisfy the processor's (oblivious) memory accesses, the evaluator needs to know their address, and these are produced in encrypted form. Thus, one needs an encryption scheme that lets the evaluator decrypt (correctly-computed) addresses and nothing else; no technique is known for this implausible-sounding goal.

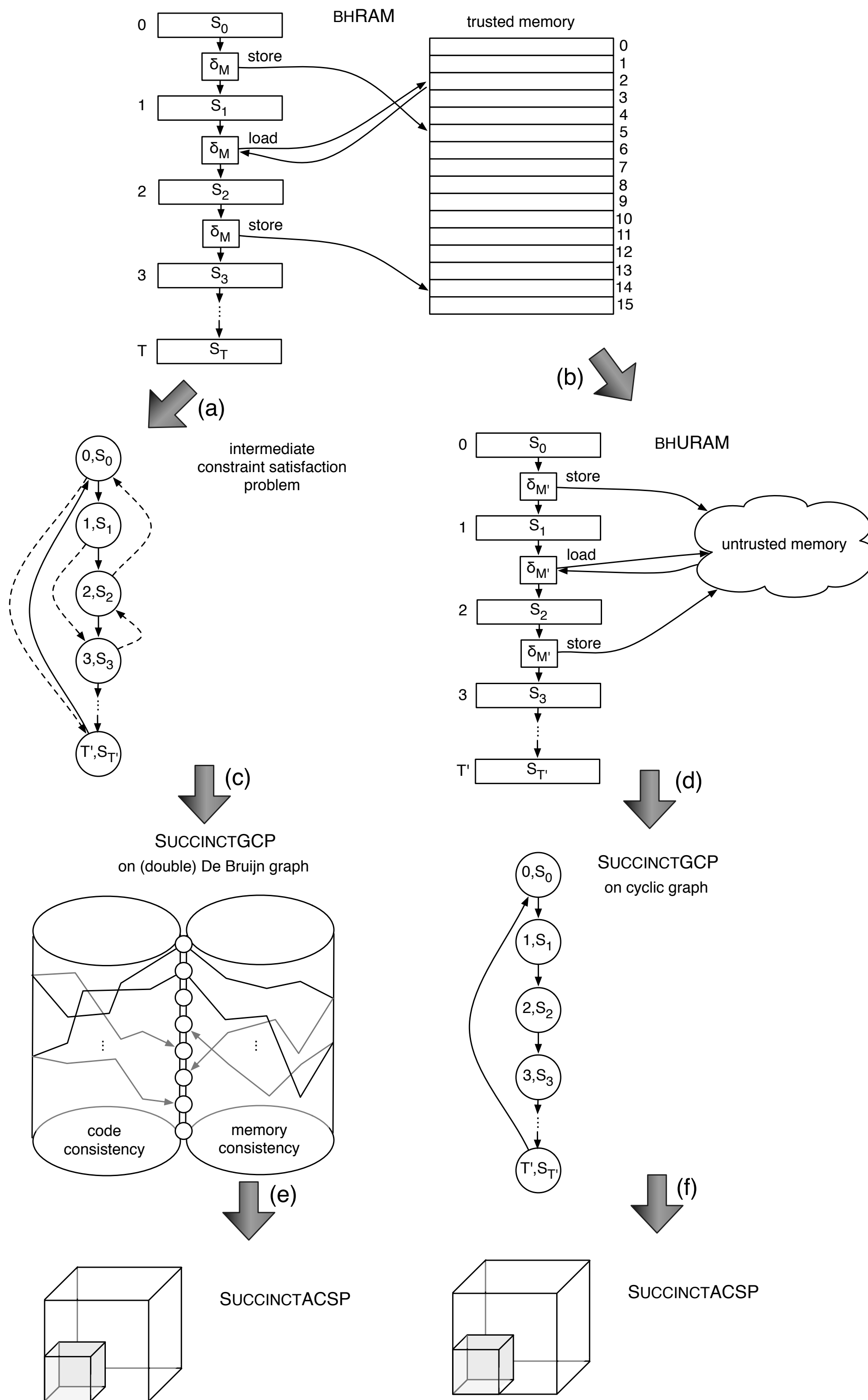


Figure 2: An "expanded" summary of our reductions. See Figure 1.

6 Definitions

We cover some basic definitions that we will use throughout the paper. Other definitions are given when appropriate.

Definition 6.1. *We say that a function $c\colon \mathbb{N} \to \mathbb{N}$ is a* **proper (complexity) function** *if it can be computed in time $O(|x| + c(|x|))$ and space $O(c(|x|))$ by some Turing machine. (See [Pap94, Definition 7.1].)*

If f is a Boolean function (a function over finite strings), we will denote by $[f]^{\mathsf{B}}$ a candidate Boolean circuit (of AND, OR, NOT, and constant gates) for computing f. Because in this paper we will be interested in carefully arithmetizing Boolean circuits, we need a sufficiently fine notion of the "degree" of a Boolean circuit; to this end, given a Boolean circuit $C\colon \{0,1\}^n \to \{0,1\}^m$, we say that C has *(multiplicative) degree* $\mathsf{D}\colon [n] \times [m] \to \mathbb{N}$ if $\mathsf{D}(i,j)$ bounds the degree of the i-th input variable in the j-th output bit of C when viewing C as an arithmetic circuit over $\mathbb{F}_2$. As a mnemonic (to remember which is the input and which is the output), we shall write $\mathsf{D}[i \to j]$ for $\mathsf{D}(i,j)$, and simply $\mathsf{D}[i]$ when C only has a single output bit.

We consider the following notion of uniformity for Boolean functions:

Definition 6.2. *Fix a Boolean function family $\{f_t\colon \{0,1\}^* \to \{0,1\}^*\}_{t\in\mathbb{N}}$. We say that $\{f_t\}_{t\in\mathbb{N}}$ is a* $(\mathsf{S},\mathsf{D},\mathsf{t},\mathsf{s})$**-uniform family of Boolean circuits** *if there exists a t-time s-space algorithm* FIND *such that $[f_t]^{\mathsf{B}} = \textsc{Find}(1^t)$ is a $\mathsf{S}(t)$-size $\mathsf{D}(t)$-degree Boolean circuit computing f_t.*

If f is a field function (a function over some field elements), we will denote by $[f]^{\mathsf{A}}$ a candidate arithmetic circuit for computing f. Given an arithmetic circuit C over variables $x_1, \dots, x_n$, we say that C has *(multiplicative) degree* $\mathsf{D}\colon [n] \to \mathbb{N}$ if $\mathsf{D}[i]$ bounds the degree of C in variable x_i.

We consider the following notion of uniformity for field functions:

Definition 6.3. *Fix a field family $\{\mathbb{F}_t\}_{t\in\mathbb{N}}$ and a function family $\{f_t\colon \mathbb{F}_t^{n(t)} \to \mathbb{F}_t\}_{t\in\mathbb{N}}$. We say that $\{f_t\}_{t\in\mathbb{N}}$ is a* $(\mathsf{S},\mathsf{D},\mathsf{t},\mathsf{s})$**-uniform family of arithmetic circuits** *(with respect to $\{\mathbb{F}_t\}_{t\in\mathbb{N}}$) if there exists a t-time s-space algorithm* FIND *such that $[f_t]^{\mathsf{A}} = \textsc{Find}(1^t)$ is a $\mathsf{S}(t)$-size $\mathsf{D}(t)$-degree $\mathbb{F}_t$-arithmetic circuit computing f_t.*

6.1 Levin Reductions

We first recall the standard definition of a Levin reduction, which is simply a Karp reduction together with efficient functions to "convert witnesses from both directions":

Definition 6.4. *Let L and L' be languages with respective polynomial-time-computable polynomially-balanced binary relations R_L and $R_{L'}$. A* **Levin reduction** *from L to L' is a triple of polynomial-time-computable functions (f_1, f_2, f_3) over finite strings such that the following conditions hold for every $x \in \{0,1\}^*$:*

1. *$x \in L$ if and only if $f_1(x) \in L'$;*
2. *for every $w \in \{0,1\}^*$, if $(x,w) \in R_L$ then $(f_1(x), f_2(x,w)) \in R_{L'}$; and*
3. *for every $w' \in \{0,1\}^*$, if $(f_1(x), w') \in R_{L'}$ then $(x, f_3(x,w')) \in R_L$.*

In this work we will consider Levin reductions that are in certain respects more general and in other respects more specific than the ones considered by the usual Definition 6.4. More precisely:

- We consider languages L whose instances have the form $x = (x, 1^t)$, where y will be an m-bit binary string and t a positive integer presented in binary (and no less than $\log m$). Loosely, the interpretation will be that x is in L if there is some 2^t-size witness w for which $(x,w) \in R_L$. We put *no restriction* on the magnitude of t relative $|y|$, and thus the relations of the languages that we consider will not necessarily be polynomially-balanced. Intuitively, that is because we want to capture long computations on small inputs.

- We only consider Levin reductions via the "identity mapping"; that is, we will only consider reductions where the "instance map" of the reduction (i.e., f_1) is the identity. This restriction will be mostly out of simplicity, rather than out of necessity (as the reductions will already contain enough complexity).

- We will only be interested in the "forward direction" of efficient witness conversion (i.e., f_2); namely, we will only care about being able to efficiently convert a witness for an instance in the language we are reducing from to the language we are reducing to.

Furthermore, we consider Levin reductions between *classes* of languages: a reduction from a class of languages $\mathcal{L}$ to another class of languages $\mathcal{L}'$ will also include an efficient function that takes as input (a description of) a language in $\mathcal{L}$ and generates (the description of) a language in $\mathcal{L}'$. Our motivation is to highlight the "succinct and uniform" nature of our reductions, across a number of languages with similar properties.

We concretize the foregoing discussion as a template definition for the notion of Levin reduction that we use, and we will instantiate it later for the specific cases that we consider. (See Definition 9.1 and Definition 11.1.)

Definition 6.5. *Let $\mathcal{L}$ and $\mathcal{L}'$ be classes of languages with polynomial-time-computable binary relations. A* **Levin reduction** *from $\mathcal{L}$ to $\mathcal{L}'$ is a pair of polynomial-time-computable functions (g_1, g_2) over finite strings such that the following conditions hold for every $L \in \mathcal{L}$ and $(x, 1^t) \in \{0,1\}^*$:*

1. *$g_1(L) \in \mathcal{L}'$;*
2. *$(x, 1^t) \in L$ if and only if $(x, 1^t) \in g_1(L)$;*
3. *for every $w \in \{0,1\}^*$, if w is a witness to "$(x, 1^t) \in L$" then $g_2(L, 1^{2^t}, w)$ is a witness to "$(x, 1^t) \in g_1(L)$".*

(Whenever L is given as input to the function g_1, it is understood that L denotes an appropriate description of the language.)

In fact, we will also consider a *computational* Levin reduction, but we will define it when we need it. (See for example Definition 9.24.)

6.2 Graphs and Routing

We use (a more general version of) the definition of *extended* (also, *wrapped*) De Bruijn graph given in [BSS08, Definition 5.4]:[4]

Definition 6.6. *Let κ and L be two positive integers. The (κ, L)* **extended De Brujn graph***, denoted $\mathrm{DB}(\kappa, L)$, is a directed 2-regular graph with L layers numbered $0, \ldots, L-1$, each containing 2^κ vertices identified by κ-bit strings. A vertex in layer $i \in \{0, \ldots, L-1\}$ with identifier $w \in \{0,1\}^\kappa$ has two neighbors in layer $(i + 1 \bmod L)$ with identifier $\mathrm{sr}(w)$ and $\mathrm{sr}(w) \oplus e_1$. In other words, the edge set is induced by the following two neighbor functions:*

$$\Gamma_1\big((i, w)\big) = \Big((i + 1 \bmod L), \mathrm{sr}(w)\Big) \ , \ \textit{and}$$
$$\Gamma_2\big((i, w)\big) = \Big((i + 1 \bmod L), \mathrm{sr}(w) \oplus e_1\Big) \ ,$$

where sr denotes the cyclic "shift right" bit operation.

Extended De Bruijn graphs, when "sufficiently wide", are rearrageable, that is, they can route any given permutation of the leftmost vertices.

Claim 6.7. *Let κ be a positive integer and $\pi \colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ a permutation. There exists a set S_π of 2^κ node-disjoint paths such that each vertex $(0, w)$ in $\mathrm{DB}(\kappa, 4\kappa - 1)$ is connected to $(0, \pi(w))$. Moreover, S_π can be found in $O(\kappa \cdot 2^\kappa)$ time and space.*

Details about the claim can be found in Appendix B; for now we simply explain how the claim follows from the material there.[5]

Proof. Because $\mathrm{DB}(\kappa, 4\kappa - 1)$ "contains three and a half" κ-dimensional De Bruijn graphs connected in tandem (except that the first and last column are identified, or "wrapped"), we can simply follow the optimizations discussed after Claim B.11 to obtain the desired set S_π. □

[4] Equivalent definitions have appeared in the same context. For example, the graph defined in [BSGH+05, Definition 4.1] is homomorphic via the mapping $(w, i) \mapsto (\mathrm{sr}^i(w^{\mathrm{r}}), i)$ to ours; also, the graph defined in [Spi95, Definition 4.3.2] is homomorphic via the mapping $(w, i) \mapsto (w^{\mathrm{r}}, i)$ to ours.

[5] While the routing properties of De Bruijn graphs are folklore, we have not been able to find explicit algorithms in the literature for routing; so, given that we are interested in an explicit implementation, we deduce an explicit routing algorithm over just *three and a half* De Bruijn graphs connected in tandem.

6.3 Arithmetic in Finite Fields

We represent elements of a finite field as polynomials modulo an irreducible polynomial of the appropriate degree. Specifically, to represent elements of $\mathbb{F}_q$, where $q = p^f$ and p is the characteristic of $\mathbb{F}_q$: we consider any *irreducible* polynomial I over $\mathbb{F}_p$ of degree f; I has a root x in $\mathbb{F}_q$ and thus $\mathbb{F}_q = \mathbb{F}_p(\mathsf{x})$, so that every element of $\mathbb{F}_q$ can be uniquely expressed as a polynomial in x over $\mathbb{F}_p$ of degree less than f. See [LN97, Section 2.5] for more details. In particular, this representation will allow us to perform field operations in time that is polylogarithmic in the field size and space that is logarithmic in the field size.

See Appendix E for more details and additional notation.

7 Random-Access Machines

We introduce our notion of random-access machines, which are a kind of register machines, beginning first with an informal discussion, followed by formal definitions.

7.1 Informal Discussion

We consider random-access machines with two inputs tapes, called tape A and B. Informally, a random-access machine M consists of a vector of k *registers* $r = (r_0, \ldots, r_{k-1})$, where each $r_i \in \{0,1\}^w$, and a vector of n *instructions* $\mathbb{P} = (I_0, \ldots, I_{n-1})$.

The machine always maintains a pointer to the next instruction to be executed; this pointer is known as the *program counter*, and we shall denote it by pc; of course, its initial value is 0. We assume that pc, like each register, is a w-bit string; in particular, it will always be the case that $n \leq 2^w$.

The machine also has random-access to *memory*, which we represent as a vector of memory cells $\mathbb{M}$; we assume that $\mathbb{M}$ consists of 2^w strings of w bits, i.e., $\mathbb{M} \in (\{0,1\}^w)^{2^w}$.

At every time step the instruction I_{pc} is executed; the instruction may modify memory, registers, or the program counter; if the program counter is not modified by the instruction, it is set to $\mathsf{pc}+1$, so that in the next time step instruction $I_{\mathsf{pc}+1}$ is executed. An instruction may also choose to read the next w bits from either input tape.

A special instruction denotes the end of the computation, and the machine outputs either accept or reject.

Instructions. A important question is which instructions we allow the machine to execute. We always consider a set of *basic* instructions:

instruction	arg. 1	arg. 2	arg. 3	semantics
nop				no operation (does nothing)
read_A	i			read the next w-bit string (or #) from tape A and put it into r_i
read_B	i			read the next w-bit string (or #) from tape B and put it into r_i
load	i	j		$r_j \leftarrow \mathbb{M}[r_i]$
store	j	x		$\mathbb{M}[r_j] \leftarrow x$
iseof	i	j		if $r_j = \#$, then $r_i \leftarrow 1$, else $r_i \leftarrow 0$
out	s			print s and halt
$A \in \mathbb{A}$	i	x	y	$r_i \leftarrow A(x,y)$

where in the above table $\mathsf{s} \in \{\mathsf{accept}, \mathsf{reject}\}$ and each of x and y is a register, the symbol "pc", or a constant in $\{0,1\}^w$. Note that reading from a tape is assumed to be in one direction only: that is, every time one reads the next w-bit string from (say) tape A, the pointer on tape A moves to the next w-bit string never moves back; when the end of the input on the tape has been reached, reading from the tape always returns the symbol #.

On top of the basic operations, we also consider a set $\mathbb{A}$ of *arithmetic* operations. For example, the set $\mathbb{A}$ may consist of the following operations:

instruction	arg. 1	arg. 2	arg. 3	semantics
add	i	x	y	$r_i \leftarrow x + y$
sub	i	x	y	$r_i \leftarrow x - y$
mul	i	x	y	$r_i \leftarrow x * y$
and	i	x	y	$r_i \leftarrow x \mathbin{\&} y$
or	i	x	y	$r_i \leftarrow x \mid y$
not	i	x		$r_i \leftarrow !\, x$
cmp	i	x	y	if $x < y$ then $r_i \leftarrow 1$ else if $x = y$ then $r_i \leftarrow 0$ else if $x > y$ then $r_i \leftarrow -1$
jl	i	L		if $r_i \geq 0$ then $\mathsf{pc} \leftarrow L$
jle	i	L		if $r_i > 0$ then $\mathsf{pc} \leftarrow L$
je	i	L		if $r_i = 0$ then $\mathsf{pc} \leftarrow L$
jne	i	L		if $r_i \neq 0$ then $\mathsf{pc} \leftarrow L$
jmp	L			$\mathsf{pc} \leftarrow L$

where in the above table L is an instruction number.

Universal computation. For universal computation (given enough memory), all of the above operations are not needed. For example, given $\mathbb{A}$ with only add and mul operations and the conditional branch je, we can carry out any computation.

Other architectures. Our notion of a random-access machine is fairly general, though there are other "architectures" that do not exactly fit our notion, which can be viewed as a reduced instruction set computer (or load-store architecture).

For example, in a *one instruction set computer* [Jon88][GL03], the single instruction usually has multiple simultaneous memory accesses, while our definition only envisages a single memory access per load or store instruction. The techniques that we develop for handling our notion of random-access machine can be extended to also handle architectures, e.g., having multiple memory accesses per instruction.

Alternatively, (in a "morally equivalent way") we can perform a "software reduction" of other architectures to our notion of random access machine by, e.g., striping multiple memory accesses across successive instructions. For example, if the single instruction of the computer is "subtract and branch if less than or equal to zero" (subleq), there are three simultaneous memory accesses, two loads and a write. More precisely, the instruction subleq $a\ b\ c$ means $\mathbb{M}[b] := \mathbb{M}[b] - \mathbb{M}[a]$ and if $\mathbb{M}[b] \leq 0$ go to c. We can then map the instruction subleq $a\ b\ c$ to the three "microcode" instructions $\mathsf{subleq}_1\ a$, $\mathsf{subleq}_2\ b$, and $\mathsf{subleq}_3\ c$ which all together perform what is needed by subleq $a\ b\ c$.

We note that such minimalist architectures are still Turing complete, and programs are written using *self-modifying code.*

7.2 Formal Definitions

We now turn to formalizing the above discussion. The goal of this section is to define the *bounded-halting problem for RAMs* with trusted memory and untrusted memory.

We begin with the definition of a random-access machine itself.

Definition 7.1. *Let w and k be positive integers and $\mathbb{A}$ a set of functions $A \colon \{0,1\}^w \times \{0,1\}^w \to \{0,1\}^w$. A two-tape* **random-access machine** *(RAM) with binary arithmetic unit is a tuple $M = \langle w, k, \mathbb{A}, \mathbb{P} \rangle$, where:*

- *w is the* register size*;*
- *k is the* number of registers*;*
- *$\mathbb{A}$ is the* arithmetic unit*; and*
- *$\mathbb{P} = (I_0, \ldots, I_{n-1})$, where $0 \leq n \leq 2^w$ and each I_i is a basic or $\mathbb{A}$ instruction, is the* program.

The state of a random-access machine is given by a configuration, which contains the current program counter, the next inputs from the two tapes, and the values of all the registers.

Definition 7.2. *Let $M = \langle w, k, \mathbb{A}, \mathbb{P} \rangle$ be a two-tape random-access machine. A* **configuration** *of M is a tuple*

$$S = \left[\mathsf{pc}, r_0, \ldots, r_{k-1}\right] \ ,$$

where pc *is the current instruction to be executed in the program $\mathbb{P}$ and $r_0, \ldots, r_{k-1}$ are the current w-bit values of the k registers.*

Note that the size of a configuration is $|S| = (1 + k)w$ bits.

Certain configurations are special in that they mark the beginning of a computation, or its ending (and, if so, whether the computation was accepting or not).

Definition 7.3. *Let $M = \langle w, k, \mathbb{A}, \mathbb{P} \rangle$ be a two-tape random-access machine and $S = [\mathsf{pc}, r_0, \ldots, r_{k-1}]$ a configuration of M.*

- *We say that S is* **initial** *for M if $\mathsf{pc} = 0$, and $r_0 = \cdots = r_{k-1} = 0^w$.*

  (I.e., the program counter points to the first instruction and all the k w-bit registers are zero.)

- *We say that S is* **final** *for M if $I_{\mathsf{pc}} = (\mathsf{out}, \mathsf{s})$ for some* s.

  (I.e., the program counter points to an out *instruction.)*

- *We say that a final configuration S is* **accepting** *for M if $I_{\mathsf{pc}} = (\mathsf{out}, \mathsf{accept})$.*

  (I.e., the program counter points to an out *instruction whose argument is* accept*.)*

  Otherwise, we say that S is **rejecting**.

A sequence of configurations starting in an initial configuration and ending in an accepting configuration is accepting.

Definition 7.4. *Let M be a two-tape random-access machine, and $\vec{S} = (S_0, \dots, S_{T-1})$ a sequence of configurations of M. We say that $\vec{S}$ is* **accepting** *for M if S_0 is initial for M and S_{T-1} is accepting for M.*

If a random-access machine has inputs on either tape, we must specify what it means for a sequence of configurations to be consistent with the two inputs. Intuitively, this simply means that, when the machine reads from the tape, the actual values of the inputs appear in the configurations.

Definition 7.5. *Let M be a two-tape random-access machine, $\vec{S} = (S_0, \dots, S_{T-1})$ a sequence of configurations of M, and x and w two input strings. We say that $\vec{S}$ is* **consistent** *with (x, w) if:*

(i) $\rho_0 \cdots \rho_{\ell-1}$ is a prefix of x, where $\rho_0 \cdots \rho_{\ell-1}$ is the (time-ordered) concatenation of all the w-bit strings ρ_i (not equal to #) read from tape A according to $\vec{S}$, and

(ii) $\sigma_0 \cdots \sigma_{\ell-1}$ is a prefix of w, where $\sigma_0 \cdots \sigma_{\ell-1}$ is the (time-ordered) concatenation of all the w-bit strings σ_i (not equal to #) read from tape B according to $\vec{S}$.

Given a configuration S and a memory vector $\mathbb{M}$, the configuration obtained by executing one step of computation is "implied" by S.

Definition 7.6. *Let M be a two-tape random-access machine, and let S and S' be two configurations of M.*

- *For any memory vector $\mathbb{M}$ for M, we say that S* **implies** *S' via $\mathbb{M}$, denoted $S \overset{\mathbb{M}}{\rightsquigarrow} S'$, if, by executing the instruction I_{pc}, M goes from configuration S and memory $\mathbb{M}$ to configuration S' (and possibly some other memory vector $\mathbb{M}'$).*

- *We say that S* **implies** *S' (without specifying a memory vector $\mathbb{M}$), denoted $S \rightsquigarrow S'$, if there exists a memory vector $\mathbb{M}$ for M such that $S \overset{\mathbb{M}}{\rightsquigarrow} S'$.*

Having defined implication from a given configuration to another one, we are now ready to define the transition function of a random-access machine. Intuitively, the transition function verifies "code consistency".

Definition 7.7. *Let M be a two-tape random-access machine. The* **transition function** *of M, denoted δ_M, is the Boolean function over pairs of configurations of M such that:*

$$\delta_M(S, S') \overset{\text{def}}{=} \begin{cases} 0 & \text{if } S \rightsquigarrow S' \\ 0 & \text{if } S \text{ is final for } M \text{ and } S' \text{ is initial for } M \\ 1 & \text{otherwise} \end{cases} .$$

In Appendix C.1 we give the circuit for the transition function.

We now specify under what conditions sequences of configurations are considered valid. We distinguish between the case where we only require "code consistency" and the case where we require both code consistency and "memory consistency". Intuitively, the former case is a weaker computation model where the memory is not trusted, and it is possibly up to the machine to ensure that the values from memory "make sense" (e.g., by maintaining a Merkle tree over the memory); the latter case is the "traditional" notion where the memory of the machine can be trusted.

Definition 7.8. *Let M be a two-tape random-access machine, and $\vec{S} = (S_0, \dots, S_{T-1})$ a sequence of configurations of M.*

- *We say that S is* **code-valid** *with M if $\delta_M(S_i, S_{i+1 \bmod T}) = 0$ for $i = 0, \ldots, T-1$.*
- *We say that S is* **all-valid** *with M if S is code-valid with M and, in addition, there exists a sequence $\mathbb{M}_0, \ldots, \mathbb{M}_{T-2}$ of memory vectors for M such that $\mathbb{M}_0[j] = 0^w$ for all $j \in \{0,1\}^w$ and $S_0 \overset{\mathbb{M}_0}{\rightsquigarrow} S_1 \overset{\mathbb{M}_1}{\rightsquigarrow} \cdots \overset{\mathbb{M}_{T-2}}{\rightsquigarrow} S_{T-1}$.*

We are finally ready to introduce bounded-halting problems for random-access machines. Intuitively, for a given machine M, $\textsc{bh}\mathrm{URAM}(M)$ is the set of all $(x, 1^t)$ such that, for some w, (x, w) is accepted by some computation of M with *untrusted* memory in time at most 2^t; similarly, $\textsc{bh}\mathrm{RAM}(M)$ is the set of all $(x, 1^t)$ such that, for some w, (x, w) is accepted by some computation of M with *trusted* memory in time at most 2^t.

In fact, for convenience, we shall require the length of the computation to be a power of 2; of course, this is without loss of generality (and incurs in at most a multiplicative factor of 2 in the length of the computation) as we can always pad the computation with enough nop's.

Definition 7.9. *Let M be a two-tape random-access machine.*

- **Bounded-halting problem for RAMs with untrusted memory.**

  The language $\textsc{bh}\mathrm{URAM}(M)$ consists of instances $(x, 1^t)$, where x is an m-bit binary string and t is an integer with $m \leq 2^t$ such that there exists a string w of length at most 2^t for which the following condition holds: there exists an accepting sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ of M that is code-valid for M and is consistent with (x, w).

- **Bounded-halting problem for RAMs (with trusted memory).**

  The language $\textsc{bh}\mathrm{RAM}(M)$ consists of instances $(x, 1^t)$, where x is an m-bit binary string and t is an integer with $m \leq 2^t$ such that there exists a string w of length at most 2^t for which the following condition holds: there exists an accepting sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ of M that is all-valid for M and is consistent with (x, w).

Furthermore, we denote by $\textsc{bh}\mathrm{URAM}$ the language of triples (M, x, t) such that $(x, 1^t) \in \textsc{bh}\mathrm{URAM}(M)$, and by $\textsc{bh}\mathrm{RAM}$ the language of triples (M, x, t) such that $(x, 1^t) \in \textsc{bh}\mathrm{RAM}(M)$.

8 A Generic Succinct Graph Coloring Problem

We define a family of graph coloring problems; roughly, each vertex in the graph has a *type* inducing a "local constraint" over colors in the *neighborhood* of the vertex, and membership of an instance in the language is determined by whether there exists an assignment of colors for every vertex in the graph, satisfying every local constraint, that "contains" the given instance. It is for this family of problems that we construct reductions from (bounded-halting problems for) random-access machines, and it is from this family we reduce to a family of algebraic constraint satisfaction problems (described in Section 10).

We begin with the formal definition of the class of problems; then we discuss the intuition behind the design of the definition, and explain why we believe that it is a great class of problems providing an "intermediate representation" when reducing from other "higher-level" classes of problems (such as bounded-halting problems on random-access machines).

Definition 8.1 (Succinct Graph Coloring)**.** Consider the following parameters:

1. a proper function associated with the family $\mathbf{G}$ in Parameter 2:
   (a) a regularity function $\alpha_{\mathbf{G}} \colon \mathbb{N} \to \mathbb{N}$,
2. a family $\mathbf{G} = \{G_t\}_{t\in\mathbb{N}}$ such that:
   (a) $G_t = (V_t, E_t)$ is an $\alpha_{\mathbf{G}}(t)$-regular directed graph;
3. a proper function associated with the family $\mathbf{C}$ in Parameter 4:
   (a) a cardinality function $\mathsf{c}_{\mathbf{C}} \colon \mathbb{N} \to \mathbb{N}$,
4. a family $\mathbf{C} = \{C_t\}_{t\in\mathbb{N}}$ such that:
   (a) $C_t = \{0,1\}^{\mathsf{c}_{\mathbf{C}}(t)}$ is a finite color set;
5. a proper function associated with the family $\mathbf{T}$ in Parameter 6:
   (a) a cardinality function $\mathsf{c}_{\mathbf{T}} \colon \mathbb{N} \to \mathbb{N}$,
6. a family $\mathbf{T} = \{T_t\}_{t\in\mathbb{N}}$ such that:
   (a) $T_t = \{0,1\}^{\mathsf{c}_{\mathbf{T}}(t)}$ is a finite vertex-type set;
7. four proper functions associated with the family $\mathbf{M}$ in Parameter 8:
   (a) a size function $\mathsf{S}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N}$,
   (b) a degree function $\mathsf{D}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N}$,
   (c) a time function $\mathsf{t}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N}$, and
   (d) a space function $\mathsf{s}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N}$;
8. a family $\mathbf{M} = \{M_t\}_{t\in\mathbb{N}}$ such that:
   (a) $M_t \colon V_t \to T_t$ gives the type for each vertex in V_t, and
   (b) there exists a $\mathsf{t}_{\mathbf{M}}$-time $\mathsf{s}_{\mathbf{M}}$-space algorithm $\textsc{Find}\mathbf{M}$ such that, for every $t \in \mathbb{N}$, $[M_t]^{\mathsf{B}} = \textsc{Find}\mathbf{M}(1^t)$ is a $\mathsf{S}_{\mathbf{M}}(t)$-size $\mathsf{D}_{\mathbf{M}}(t)$-degree circuit computing M_t;
9. four proper functions associated with the family $\mathbf{K}$ in Parameter 10:
   (a) a size function $\mathsf{S}_{\mathbf{K}} \colon \mathbb{N} \to \mathbb{N}$,
   (b) a degree function $\mathsf{D}_{\mathbf{K}} \colon \mathbb{N} \to \mathbb{N}$,
   (c) a time function $\mathsf{t}_{\mathbf{K}} \colon \mathbb{N} \to \mathbb{N}$, and
   (d) a space function $\mathsf{s}_{\mathbf{K}} \colon \mathbb{N} \to \mathbb{N}$;
10. a family $\mathbf{K} = \{K_t\}_{t\in\mathbb{N}}$ such that:
    (a) $K_t \colon T_t \times C_t^{1+\alpha_{\mathbf{G}}(t)} \to \{0,1\}$ is a coloring constraint, and
    (b) there exists a $\mathsf{t}_{\mathbf{K}}$-time $\mathsf{s}_{\mathbf{K}}$-space algorithm $\textsc{Find}\mathbf{K}$ such that, for every $t \in \mathbb{N}$, $[K_t]^{\mathsf{B}} = \textsc{Find}\mathbf{K}(1^t)$ is a $\mathsf{S}_{\mathbf{K}}(t)$-size $\mathsf{D}_{\mathbf{K}}(t)$-degree circuit computing K_t;
11. two functions associated with the family $\mathbf{W}$ in Parameter 12:

(a) a time function $\mathsf{t}_{\mathbf{W}} \colon \mathbb{N} \to \mathbb{N}$, and
(b) a space function $\mathsf{s}_{\mathbf{W}} \colon \mathbb{N} \to \mathbb{N}$;

12. a family $\mathbf{W} = \{W_t\}_{t \in \mathbb{N}}$ such that:
    (a) W_t is a subset of V_t, and
    (b) there exists a $\mathsf{t}_{\mathbf{W}}$-time $\mathsf{s}_{\mathbf{W}}$-space algorithm $\textsc{Find}\mathbf{W}$ such that $v_i := \textsc{Find}\mathbf{W}(1^t, i)$ is i-th vertex in W_t (under some canonical ordering of W_t) for every $t \in \mathbb{N}$ and $i \in \{1, \dots, |W_t|\}$;
13. two functions associated with the family $\mathbf{F}$ in Parameter 14:
    (a) a time function $\mathsf{t}_{\mathbf{F}} \colon \mathbb{N} \to \mathbb{N}$, and
    (b) a space function $\mathsf{s}_{\mathbf{F}} \colon \mathbb{N} \to \mathbb{N}$;
14. a family $\mathbf{F} = \{F_t\}_{t \in \mathbb{N}}$ such that:
    (a) $F_t \colon \{0,1\}^* \to \{0,1\}$ is a function, and
    (b) there exists a $\mathsf{t}_{\mathbf{F}}$-time $\mathsf{s}_{\mathbf{F}}$-space algorithm $\textsc{Comp}\mathbf{F}$ such that $\textsc{Comp}\mathbf{F}(1^t, \cdot)$ computes $F_t(\cdot)$ for every $t \in \mathbb{N}$.

The language $\textsc{Succinct}$GCP, with respect to a choice $\mathsf{par}_{\mathrm{sGCP}}$ of the above parameters, consists of instances $(x, 1^t)$, where x is a binary string and t is an integer with $|x| \leq 2^t$, such that there exists a coloring $C \colon V_t \to C_t$ for which the following two conditions hold:

(i) *Satisfiability of constraints.* Letting $\Gamma_{t,1}, \dots, \Gamma_{t,\alpha_{\mathbf{G}}(t)}$ be the $\alpha_{\mathbf{G}}(t)$ neighbor functions in G_t, for every vertex $v \in V_t$,

$$K_t\Big(M_t(v), C(v), \big(C \circ \Gamma_{t,1}\big)(v), \dots, \big(C \circ \Gamma_{t,\alpha_{\mathbf{G}}(t)}\big)(v)\Big) = 0 \ . \tag{1}$$

If so, we say that the coloring C *satisfies the coloring constraints* induced by K_t and M_t.

(ii) *Consistency with the instance.* For every index $i \in \{1, \dots, |W_t|\}$, letting v_i be the i-th vertex in W_t,

$$F_t\Big(x, \big(C(v_i)\big)_{i=1}^{|x|}\Big) = 0 \ .$$

If so, we say that the coloring C *is consistent* with the instance $(x, 1^t)$.

The above definition for succinct graph coloring problems is quite a mouthful; this is because the language itself encodes requirements ensuring that "large objects" (such as the function mapping a vertex in a large graph to its type, the function mapping a type and the colors in a neighborhood to 0 or 1, certain subsets of vertices, and so on) can be computed using very few resources by using succinct and functional representations of such objects (for example, efficiently constructible Boolean circuits). Ultimately, these requirements will directly affect similar requirements in the succinct algebraic constraint satisfaction problems of Section 10, and eventually enable the existence of an efficient PCP verifier for membership in the language. (See [BSCGT12] for more.)

Intuitive discussion. At high level, a choice of parameters $\mathsf{par}_{\mathrm{sGCP}}$ for $\textsc{Succinct}$GCP identifies a collection of infinite families of objects (one object for each $t \in \mathbb{N}$). When a specific instance $(x, 1^t)$ is considered for membership in $\textsc{Succinct}\mathrm{GCP}(\mathsf{par}_{\mathrm{sGCP}})$, the t-th element from each of these families in the collection is used to determine membership of the instance; candidate witnesses are colorings for the graph G_t. Despite the long definition, these objects interact in natural ways, so we now go over the parameters from Definition 8.1 in less formal terms, explaining some of the intuition behind the design of the definition.

- **Parameter 1 and Parameter 2.** The parameter $\mathbf{G} = \{G_t\}_{t \in \mathbb{N}}$ is a family of graphs, where the t-th graph is directed and $\alpha_{\mathbf{G}}(t)$-regular. A "succinct" representation of this family is provided by the $\alpha_{\mathbf{G}}(t)$ neighbor functions $\Gamma_{t,1}, \dots, \Gamma_{t,\alpha_{\mathbf{G}}(t)}$. We choose to not encode in the definition where to obtain these neighbor functions, as in all the settings we consider these will be almost trivial functions that can be very easily computed by a uniform algorithm that takes as input 1^t, a natural representation of a vertex, and the index of the desired neighbor. Intuitively, G_t will hold information about a computation whose correctness we wish to enforce.

- **Parameter 3 and Parameter 4.** The parameter $\mathbf{C} = \{C_t\}_{t \in \mathbb{N}}$ is a family of finite color sets, the t-th one of cardinality $\mathsf{c}_{\mathbf{C}}(t)$, representing the domain of the colorings we will consider. We assume without loss of generality that the colors in C_t are simply binary strings of length $\mathsf{c}_{\mathbf{C}}(t)$. A concise representation of this family will not matter for our purposes.

- **Parameter 5 and Parameter 6.** The parameter $\mathbf{T} = \{T_t\}_{t \in \mathbb{N}}$ is a family of finite vertex type sets, the t-th one of cardinality $\mathsf{c}_{\mathbf{T}}(t)$, representing the possible types of vertices in each graph of the family. We assume without loss of generality that the types in T_t are simply binary strings of length $\mathsf{c}_{\mathbf{T}}(t)$. A concise representation of this family will not matter for our purposes.

- **Parameter 7 and Parameter 8.** The parameter $\mathbf{M} = \{M_t\}_{t \in \mathbb{N}}$ is a family of functions, where the t-th function in the family maps a vertex in G_t to its type. Intuitively, vertex types let us "mark" different sections of a graph, so that we may enforce different local constraints; thus vertex types are just a very useful degree of freedom that will let us easily express different constraints in different portions of the graph. To make sure that each of these functions can be found (and then evaluated) efficiently, the definition prescribes the existence of an algorithm $\textsc{Find}\mathbf{M}$ that for each t is able to output a small Boolean circuit that computes M_t.

- **Parameter 9 and Parameter 10.** The parameter $\mathbf{K} = \{K_t\}_{t \in \mathbb{N}}$ is a family of constraints, where the t-th constraint is a *single* condition that is required to hold for every vertex in the graph; the constraint takes as input the vertex type, the color of the vertex, and the colors of the vertices in the neighborhood of the vertex. Intuitively, depending on the vertex type, the constraint will enforce different properties among the colors of the vertex and those of its neighbors. With a careful use of vertex types, even if we only consider a single one enforced across the whole graph, we can express many properties on the graph. To make sure that each of the constraints can be found (and then evaluated) efficiently, the definition prescribes the existence of an algorithm $\textsc{Find}\mathbf{K}$ that for each t is able to output a small Boolean circuit that computes K_t.

At high level, the parameters described until now could, say, encode a variety of constraints to ensure "correct computation" (say, of Turing machine, or a random-access machine), but so far did not encode anything about whether this correct computation had anything to do with the instance $(x, 1^t)$ under consideration. The remaining parameters relate the computation to the instance at hand.

- **Parameter 11 and Parameter 12.** The parameter $\mathbf{W} = \{W_t\}_{t \in \mathbb{N}}$ is a family of subsets, where the t-th subset W_t is a subset of the vertices V_t in the graph G_t. Each subset W_t identifies on which part of V_t a (candidate witness) coloring must contain information related to the instance $(x, 1^t)$, which can be verified by the next parameter. As usual, we need an algorithm $\textsc{Find}\mathbf{W}$ to ensure that accessing elements of W_t can be done appropriately efficiently.

- **Parameter 13 and Parameter 14.** The parameter $\mathbf{F} = \{F_t\}_{t \in \mathbb{N}}$ is a family of "instance-consistency" functions. Intuitively, given the color of enough elements in W_t assigned by a (candidate witness) coloring for the graph, the function F_t, which can be computed by the algorithm $\textsc{Comp}\mathbf{F}$, verifies that the received colors are consistent with $(x, 1^t)$.

In summary, the "satisfiability of constraints" condition of the definition should be interpreted as saying that some computation (the particulars of which are captured by the specific choices of parameters) was performed correctly, while the other condition, "consistency with the instance", should be interpreted as saying that this computation was performed on the instance at hand.

Remark 8.2. How is a choice of parameters $\mathsf{par}_{\mathrm{sGCP}}$ for $\textsc{Succinct}$GCP (concisely) specified? All the "complexity functions" from Definition 8.1 (i.e., those specifying running times, sizes of arithmetic circuits, and so on) were chosen to be proper (see Definition 6.1), and thus each has an efficient algorithm that computes it (which, for simplicity, we denote with the same name as the function); moreover, every infinite family comes with an algorithm that computes the information about the family we are interested in. Thus, a choice of

parameters $\mathsf{par}_{\mathrm{sGCP}}$ can be specified as follows:

$$\mathsf{par}_{\mathrm{sGCP}} = \begin{pmatrix} \alpha_{\mathbf{G}}, \\ \mathsf{c}_{\mathbf{C}}, \\ \mathsf{c}_{\mathbf{T}}, \\ (\mathsf{S}_{\mathbf{M}}, \mathsf{D}_{\mathbf{M}}, \mathsf{t}_{\mathbf{M}}, \mathsf{s}_{\mathbf{M}}, \textsc{Find}\mathbf{M}), \\ (\mathsf{S}_{\mathbf{K}}, \mathsf{D}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathsf{s}_{\mathbf{K}}, \textsc{Find}\mathbf{K}), \\ (\mathsf{t}_{\mathbf{W}}, \mathsf{s}_{\mathbf{W}}, \textsc{Find}\mathbf{W}), \\ (\mathsf{t}_{\mathbf{F}}, \mathsf{s}_{\mathbf{F}}, \textsc{Comp}\mathbf{F}) \end{pmatrix} .$$

Remark 8.3. Our design of Definition 8.1 was inspired by related definitions that have appeared in [BSS08] and [BSGH$^+$05]. We briefly discuss here how our definition compares to the previous ones.

The definition of Ben-Sasson and Sudan [BSS08, Definition 5.6] also considers coloring problems over graphs; however, their definition is only specialized to De Bruijn graphs and (as is clear for its compactness!) does *not* require succinctness because their paper does not attempt to construct PCP verifiers that are efficient in time and space. (Note that, in the non-succinct case, the "consistency with the instance" requirement disappears.)

The later paper of Ben-Sasson et al. [BSGH$^+$05], which constructs efficient PCP verifiers, does indeed give a definition [BSGH$^+$05, Definition 4.3] for a succinct problem about coloring problems (similar to the previous one of Ben-Sasson and Sudan [BSS08, Definition 5.6]), but is specific to De Bruijn graphs and specific for the parameters obtained when reducing from bounded halting problems on Turing machines.

We consider a generalization of [BSGH$^+$05, Definition 4.3] that leaves unspecified degrees of freedom that we believe important for "supporting" a variety of reductions from other models of computation.

9 From RAMs to Succinct GCPs

We construct Levin reductions from bounded halting problems on random-access machines to certain choices of parameters of SuccinctGCP. Both of these classes of problems are "succinct" constraint satisfaction problems, where a candidate instance $(x, 1^t)$ must satisfy a very large set of constraints in order to be in a language. As already discussed in Section 6.1, we seek efficient Levin reductions that implicitly convert between the very large sets of constraints prescribed by the two languages.

This section consists of two parts:

- in Section 9.1, we construct a Levin reduction from bounded-halting problems on random-access machines with *trusted* memory to succinct GCPs; and
- in Section 9.2, we construct a computational Levin reduction from bounded-halting problems on random-access machines with *untrusted* memory to succinct GCPs.

From a high level, both reductions have similar asymptotic efficiency, but they involve very different techniques in the way the reduction is implemented.

9.1 A Levin Reduction

Given the "template" Definition 6.5, we give for convenience here the following specialized definition:

Definition 9.1. *Fix a class of random-access machines $\mathcal{P}_{\mathrm{RAM}}$. We say that a pair of polynomial-time-computable functions $(F_{\mathsf{p}}, F_{\mathsf{w}})$ is a* **Levin reduction** *from* bhRAM *to* SuccinctGCP *with respect to $\mathcal{P}_{\mathrm{RAM}}$ if the following three conditions are satisfied for every choice of random-access machine $M \in \mathcal{P}_{\mathrm{RAM}}$:*

1. $\mathsf{par}_{\mathrm{sGCP}} := F_{\mathsf{p}}(M)$ *is a choice of parameters for* SuccinctGCP*.*
2. *For every instance $(x, 1^t) \in \{0,1\}^*$, $(x, 1^t) \in \mathrm{bhRAM}(M)$ if and only if $(x, 1^t) \in \mathrm{SuccinctGCP}(\mathsf{par}_{\mathrm{sGCP}})$.*
3. *For every instance $(x, 1^t) \in \{0,1\}^*$, if $(w, \vec{S})$ is a witness to "$(x, 1^t) \in \mathrm{bhRAM}(M)$" then the coloring $C := F_{\mathsf{w}}(M, 1^{2^t}, (w, \vec{S}))$ is a witness to "$(x, 1^t) \in \mathrm{SuccinctGCP}(\mathsf{par}_{\mathrm{sGCP}})$".*

The two functions F_{p} and F_{w} are respectively called as the "parameter reduction" and the "witness reduction".

High-level idea. Our goal is to construct a Levin reduction from bounded-halting problems on random-access machines (with *trusted* memory) to succinct GCPs. Our strategy is to single out the constraints that need to be met in order for a computation of a random-access machine to be valid, accepting, and consistent with the given input; these constraints fall into two categories: constraints to ensure code consistency (i.e., the transition function of the machine correctly went from the current state to the next one) and constraints to ensure memory consistency (i.e., each memory load returns the last value that was written there). We express these constraints as edge constraints in a graph, which we then "structure" by using routing techniques. The resulting coloring problem can be then formalized as a succinct GCP problem.

We note that the idea of separately checking code-consistency constraints and memory-consistency constraints is fairly natural and, e.g., was already used by Gurevich and Shelah [GS89, Theorem 2] to show that the class of languages non-deterministically accepted by a random-access computer[6] in quasilinear time is equal to the class of languages non-deterministically accepted by a multi-tape Turing machine in quasilinear time; in [GS89, Theorem 2] the analogue of the technique of routing is the fact that multi-tape Turing machines can sort in quasilinear time [Sch78].

Our plan, step by step. We construct the Levin reduction in four steps:

- **Step 1 (Section 9.1.1).** For a given random-access machine M, sequence of configurations $\vec{S}$, and input strings (x, w), we show that the problem of whether $\vec{S}$ is accepting and all-valid for M and is consistent with (x, w) (recall Definition 7.4, Definition 7.8, and Definition 7.5) can be equivalently stated as whether a corresponding "computation graph" $G_M^{\vec{S}}$ that is valid, accepting, and consistent with (x, w) exists or not.

  Informally, the vertices of a computation graph $G_M^{\vec{S}}$ are the (time-stamped) configurations in $\vec{S}$; the validity of $G_M^{\vec{S}}$ can be determined by verifying, at every edge of the graph, a *local* constraint (which can

[6] See [AV77]; a random-access computer is essentially a random-access machine with a rich set of instructions.

be generated easily from M) that only takes as input the configuration of each of the two vertices of the edge.

However, different sequences of configurations for the same M determine different sets of edges allowable in a valid computation graph (if one exists).

- **Step 2 (Section 9.1.2).** For a given random-access machine M, sequence of configurations $\vec{S}$, and input strings (x, w), we show that the problem of determining whether a computation graph $G_M^{\vec{S}}$ that is valid, accepting, and consistent with (x, w) exists or not can be equivalently stated as whether a corresponding "computation routing" $R_M^{\vec{S}}$ that is valid, accepting, and consistent with (x, w) exists or not.

  Informally, the validity of $R_M^{\vec{S}}$ can be determined by verifying, at every vertex of a *fixed* routing network, a local constraint (which can be generated easily from M) that only takes as input information available at the vertex and its neighbors.

  Intuitively, we do so by leveraging the rearrangeability of De Bruijn graphs in order to "route" vertices of a computation graph $G_M^{\vec{S}}$.

- **Step 3 (Section 9.1.3).** For a given random-access machine M and input string w, we show how the problem of whether there exists a second input string w, a sequence of configurations $\vec{S}$, and a computation routing $R_M^{\vec{S}}$ (for M with respect to $\vec{S}$) that is valid, accepting, and consistent with (x, w) exists or not can be formalized as a problem of deciding whether $(x, 1^t)$ is in $\textsc{Succinct}\text{GCP}(\mathsf{par}_{\text{sGCP}})$, for an appropriate choice of $\mathsf{par}_{\text{sGCP}}$ depending on M.

  Informally, we do so by noting that figuring out which constraint to apply at a given vertex is a simple operation that can easily be made to depend on a properly abstracted "vertex type"; we can then "bunch up" all the requisite constraints into a universal constraint that needs to be enforced at every vertex of the graph.

- **Step 4 (Section 9.1.4).** We conclude by writing down explicitly the two functions F_{p} and F_{w} of the Levin reduction and explaining why they are correct.

We now proceed to describe each of the four steps in detail. Throughout, it will be useful to have the definitions from Section 7 in mind.

9.1.1 Step 1: From RAMs to computation graphs

We begin by defining what it means for a timestamp-configuration pair (τ, S) to "precede in memory" another timestamp-configuration pair (τ', S'). Intuitively, this will be so when both configurations correspond to memory operations, and the second configuration has a memory dependency on the first one. (The definition is in fact slightly more complicated, because we shall eventually consider memory accesses that are somewhat canonical; see Definition 9.7.)

Definition 9.2. *Let M be a two-tape random-access machine, $S = [\mathsf{pc}, r_0, \ldots, r_{k-1}]$ and $S' = [\mathsf{pc}', r'_0, \ldots, r'_{k-1}]$ two configurations of M, and τ and τ' non-negative integers. We say that (τ, S)* **precedes** *(τ', S')* **in memory***, denoted $(\tau, S) \overset{\text{m}}{\prec} (\tau', S')$, if pc and pc' each point to a store or load instruction and at least one of the following three conditions is satisfied: letting r_j and $r'_{j'}$ be the two memory addresses respectively accessed and i and i' the two registers respectively read/written,*

(i) S and S' correspond to accesses to the same memory address (i.e., $\tau < \tau'$ and $r_j = r'_{j'}$) and the accesses are consistent (i.e., if both S' and S' are load instructions, then the value loaded by both S and S' is the same; if S is a store instruction and S' is a load instruction, then the value loaded in S' is the same as the value stored in S).

(ii) S and S' correspond to accesses to different memory addresses (i.e., $r_j < r'_{j'}$); moreover, if pc' points to a load instruction then the loaded value is 0 (i.e., $r_{i'} = 0$).

(iii) S and S' correspond to accesses to different memory address where the second configuration is the first in the computation and thus it must access the first memory cell (i.e., $\tau' = 0$ and $r'_{j'} = 0$); moreover, if pc' points to a load instruction then the loaded value is 0 (i.e., $r_{i'} = 0$).

Next, for a random-access machine M and sequence of configurations $\vec{S}$, we define its "computation graph" $G_M^{\vec{S}}$. Intuitively, it is a graph where the edges consist of all the configurations, and the edges consist of "time edges" (connecting successive configurations in time) and "memory edges" (allegedly, connecting configurations that are successive "in memory").

Definition 9.3. *Let M be a two-tape random-access machine and $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations for M. A* **computation graph** *for M with respect to $\vec{S}$ is a triple $G_M^{\vec{S}} = (V, E_{\mathsf{t}}, E_{\mathsf{m}})$ such that $(V, E_{\mathsf{t}} \cup E_{\mathsf{m}})$ is a graph satisfying the following two conditions:*

(i) the set of vertices V is equal to $\{(\tau, S_\tau)\}_{\tau \in \{0,\ldots,T-1\}}$; and

(ii) the set of edges E_{t} is equal to $\{((\tau, S_\tau), (\tau + 1 \bmod T, S_{\tau+1 \bmod T}))\}_{\tau \in \{0,\ldots,T-1\}}$.

Intuitively, a computation graph $G_M^{\vec{S}}$ is valid if its time edges respect the transition function and its memory edges are successive "in memory"; it is accepting if $\vec{S}$ is also, and it is consistent with the input strings if $\vec{S}$ is also.

Definition 9.4. *Let M be a two-tape random-access machine, $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations for M, (x, w) two input strings, and $G_M^{\vec{S}}$ a computation graph for M with respect to $\vec{S}$.*

- *We say that $G_M^{\vec{S}}$ is* **accepting** *if $\vec{S}$ is accepting for M. (Recall Definition 7.4.)*
- *We say that $G_M^{\vec{S}}$ is* **consistent** *with (x, w) if $\vec{S}$ is consistent with (x, w). (Recall Definition 7.5.)*
- *We say that $G_M^{\vec{S}}$ is* **valid** *if it satisfies the following conditions:*
  - *(i) For every edge $((\tau, S_\tau), (\tau + 1 \bmod T, S_{\tau+1 \bmod T})) \in E_{\mathsf{t}}$, $S_\tau \rightsquigarrow S_{\tau+1 \bmod T}$. (Recall Definition 7.6.)*
  - *(ii) For every edge $((\tau, S), (\tau', S')) \in E_{\mathsf{m}}$, $(\tau', S') \overset{\mathrm{m}}{\prec} (\tau, S)$. (Recall Definition 9.2.)*
  - *(iii) For every vertex (τ, S_τ) such that the instruction in S_τ is a memory instruction, the in-degree and out-degree of (τ, S_τ) in E_{m} is equal to 1.*

We show that in a valid computation graph $G_M^{\vec{S}}$ every vertex whose configuration points to a memory instruction can be reached and can reach the "initial" vertex $(0, S_0)$.

Lemma 9.5. *Let M be a two-tape random-access machine, $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations for M, and $G_M^{\vec{S}}$ a computation graph for M with respect to $\vec{S}$. Suppose that $G_M^{\vec{S}}$ is valid. Then, for any $(\tau, S) \in V$ such that pc in S points to a memory instruction,*

(i) there exists a path in E_{m} from (τ, S) to $(0, S_0)$ and

(ii) there exists a path in E_{m} from $(0, S_0)$ to (τ, S).

Proof. We only prove (i); a similar argument can be made to prove (ii).

Let (τ, S) and (τ', S') be vertices in V such that both S and S' point to a memory instruction. Let $S = [\mathsf{pc}, r_0, \ldots, r_{k-1}]$ and $S' = [\mathsf{pc}', r'_0, \ldots, r'_{k-1}]$, and let r_j and $r'_{j'}$ be the two memory addresses respectively accessed and i and i' the two registers respectively read/written by the two memory instructions pointed to by pc and pc'.

For the purpose of this proof, we write $S' \prec S$ if either "$\tau' < \tau$ and $r'_{j'} \leq r_j$" or "$r'_{j'} < r_j$". Let X be the set of those S_τ in $\vec{S}$ pointing to a memory instruction.

Note that the relation $\prec$ is well-founded with respect to X.[7] Assume by way of contradiction that this is not the case, i.e., that there exists $(k, \widetilde{S}) \in V$ such that $\widetilde{S}$ contains a memory instruction and there is no path from $(0, S_0)$ to $(k, \widetilde{S})$ using only E_{m} edges. Let P be the set of all vertices $(k', \widetilde{S}')$ for which there exists a path from $(k', \widetilde{S}')$ to $(k, \widetilde{S})$ using only edges in E_{m}. Let $(k', \widetilde{S}')$ be a minimal element of P. Since $(0, S_0) \notin P$, we have that $(k', \widetilde{S}') \neq (0, S_0)$. By Item (iii) of Definition 9.4 there exists $(k'', \widetilde{S}'')$ such that $((k'', \widetilde{S}''), (k', \widetilde{S}')) \in E_{\mathsf{m}}$.

[7] Recall that a binary relation is *well-founded* with respect to a set X if every non-empty subset of X has a minimal element with respect to the relation.

By Item (ii) of Definition 9.4 we obtain that $(k'', \widetilde{S}'') \overset{\mathrm{m}}{\prec} (k', \widetilde{S}')$ and therefore, since $(k', \widetilde{S}') \neq (0, S_0)$, we have that $\widetilde{S}'' \prec \widetilde{S}$ as defined above.

To complete the proof, we distinguish two cases:

- *Case 1:* $(k'', \widetilde{S}'') \in P$. Since $\widetilde{S} \prec \widetilde{S}'$, we have reached a contradiction to the fact that $(k', \widetilde{S}')$ is a minimal element of P.
- *Case 2:* $(k'', \widetilde{S}'') \notin P$. Since $((k'', \widetilde{S}''), (k', \widetilde{S}')) \in E_{\mathsf{m}}$ and since $(k', \widetilde{S}') \in P$, we conclude that there exists a path from $(k'', \widetilde{S}'')$ to $(k, \widetilde{S})$, which is a contradiction to $(k'', \widetilde{S}'')$ not being in P.

□

We can now use the previous lemma to show that in a valid computation graph $G_M^{\vec{S}}$ there is a *single* cycle passing through all the vertices whose configurations point to a memory instruction.

Lemma 9.6. *Let M be a two-tape random-access machine, $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations for M, and $G_M^{\vec{S}}$ a computation graph for M with respect to $\vec{S}$. Suppose that $G_M^{\vec{S}}$ is valid. Then there exists a cycle K in $G_M^{\vec{S}}$, using only edges in E_{m}, that goes through $(0, S_0)$ and contains all those (τ, S) for which the program counter in S points to a memory instruction.*

Proof. By Lemma 9.5, since $G_M^{\vec{S}}$ is valid, we know that every (τ, S) for which pc in S points to a memory instruction appears on some cycle K_τ that goes through $(0, S_0)$ and (τ, S). By definition of a computation graph, $(0, S_0)$ has only in degree and out degree equal to 1 in E_{m}. Thus, only one E_{m} cycle can go through $(0, S_0)$ and therefore there must exist a single cycle K such that $K = K_\tau$ for all τ. □

We now focus only on those random-access machines that satisfy the very minor requirement of accessing the first memory cell in the first computation step (which we can certainly assume without loss of generality).

Definition 9.7. *Let M be a random-access machine. We say that M is* **memory-well-behaved** *if M always accesses the first memory cell in the first step.*

We finally have the tools to prove that a computation graph $G_M^{\vec{S}}$ is valid (as well as accepting, and consistent with input strings x and w) if and only if $\vec{S}$ is all-valid for M (as well as accepting, and consistent with input strings x and w), as long as M is memory-well behaved.

Claim 9.8. *Let M be a memory-well-behaved two-tape random-access machine, $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations for M, and (x, w) two input strings. Then $\vec{S}$ is all-valid and accepting for M and consistent with (x, w) if and only if there exists a computation graph $G_M^{\vec{S}}$ for M with respect to $\vec{S}$ that is valid, accepting and consistent with (x, w).*

Proof. We prove in **(1)** one direction and in **(2)** the other direction.

(1) Assume that $\vec{S}$ is all-valid and accepting for M and consistent with (x, w). Let $G_M^{\vec{S}} = (V, E_{\mathsf{t}}, E_{\mathsf{m}})$ be the computation graph where we construct E_{m} as follows. For every node $(\tau, S) \in V$ such that the program counter in S points to a memory instruction:

- If S is the first configuration accessing a memory cell a_1 such that $\tau \neq 1$, let (τ', S') be the last configuration accessing a memory cell $a_2 < a_1$ such that all the cells between a_2 and a_1 are not accessed in $\vec{S}$, and add the edge $((\tau, S), (\tau', S'))$ to E_{m}.
- If $\tau = 1$, let (τ', S') be the last configuration in $\vec{S}$ containing a memory instruction, and add the edge $((\tau, S), (\tau', S'))$ to E_{m}.
- If S is not the first configuration accessing a memory cell a, let (τ', S') with $\tau' < \tau$ be another configuration accessing a such that all configurations between S' and S in $\vec{S}$ do not access a, and add the edge $((\tau, S), (\tau', S'))$ to E_{m}.

We now argue that $G_M^{\vec{S}}$ is all-valid, accepting and consistent with (x, w). Indeed, since $G_M^{\vec{S}}$ is a computation graph for M with respect to $\vec{S}$ by the first two items of Definition 9.4 we obtain that $G_M^{\vec{S}}$ is accepting and consistent with (x, w). Regarding the all-valid requirement, first, notice that since $G_M^{\vec{S}}$ is a computation graph, it immediately satisfies the first requirement of the all-valid section of Definition 9.4. Next, if $((\tau, S), (\tau', S')) \in E_{\mathsf{m}}$ then (τ', S') precedes (τ, S) in memory. In addition, since the cases above do not overlap and each configuration S that contains a memory instruction has a case corresponding to it, we obtain that the degree in E_{m} of each node (τ, S) such that S contains a memory instruction is 1. This fulfills the last two requirements of the all-valid section of Definition 9.4.

(2) Conversely, assume that $G_M^{\vec{S}}$ is valid, accepting and consistent with (x, w). By the first two sections of Definition 9.4 we obtain that $\vec{S}$ is accepting and consistent with (x, w). By definition of E_{t}, we have that, for every $0 \leq \tau \leq T$, $S_\tau \rightsquigarrow S_{\tau+1 \bmod T}$. Thus, the only thing that is left to prove is that the memory of the random-access machine behaves as defined in the code of the machine (i.e., that every load instruction indeed receives the value that was stored in that memory cell at the last store to it).

Assume by way of contradiction that there exists a configuration containing a load instruction in $\vec{S}$ such that the value received by this load in not the last value stored in that cell by the previous store or not 0 (if this is the first load that accesses that cell). Let S_τ be the first such configuration. By Lemma 9.6 there exists a cycle K through $(0, S_0)$ in $G_M^{\vec{S}}$ such that K contains only edges from E_{m} and every load or store instruction in $\vec{S}$ is present of this cycle.

Furthermore, K is the only E_{m} cycle in $G_M^{\vec{S}}$. Let (r, S_r) be the vertex following (τ, S_τ) in K, since the in degree of (τ, S_τ) in E_{m} is 1, (r, S_r) is the only vertex that has an E_{m} edge to (τ, S_τ). Finally, we distinguish between several cases, according to (r, S_r):

- If (r, S_r) contains a store instruction for the same cell that (τ, S_τ) accesses, since the value loaded in (τ, S_τ) is not the value stored in (r, S_r), we obtain that the edge $((r, S_r), (\tau, S_\tau))$ is not memory ordered which is a contradiction to the first part of Definition 9.2.
- If (r, S_r) contains a load instruction for the same cell that (τ, S_τ) accesses and the value loaded in (τ, S_τ) is not the value loaded in (r, S_r), then the edge $((r, S_r), (\tau, S_\tau))$ is not memory ordered which is a contradiction to the first part of Definition 9.2.
- If (r, S_r) contains a memory instruction that accesses a different cell than (τ, S_τ) accesses, since the value loaded in (τ, S_τ) is not 0 we obtain that the edge $((r, S_r), (\tau, S_\tau))$ is not memory ordered which is a contradiction to the second or third part of Definition 9.2 (based on whether $r = T$ or not).

□

Witness reduction. Note that not only does Claim 9.8 tell us an equivalence between the two decision problems, but its proof tells us how deduce a "witness" too. Specifically, given M, $\vec{S}$, and (x, w), we can construct the graph $G_M^{\vec{S}}$ by deciding which edges should go in E_{m} by following the three bullet points in part **(1)** of the proof. (As for the reverse direction, part **(2)** of the proof gives us that, but we are not interested in "going back".)

9.1.2 Step 2: From computation graphs to (double) De Bruijn graphs

Recall that extended De Bruijn graphs (see Definition 6.6), when "sufficiently wide", are rearrangeable (see Claim 6.7). At high level, we wish to leverage the rearrangeability property De Bruijn graphs to ensure that the edges of a computation graph do not depend on the particular sequence of configurations under consideration.

Looking back at Definition 9.4, we note that the two sets of edges E_{t} and E_{m} of a computation graph are "treated differently" in the sense that exactly which edges are allowed in each of those sets in a valid computation graph is different. More precisely, the edges in E_{t} (i.e., the "time edges") induce a permutation on the vertices consisting of a configuration being mapped to the successive configuration in time; the edges in E_{m} (i.e., the "memory edges") induce a permutation on the vertices consisting of a configuration being mapped to the successive configuration in memory (according to Definition 9.2).

We shall route each of the two permutations with an extended De Bruijn graph.

We thus begin by defining a "double" extended De Bruijn graph, which is simply a graph with two extended De Bruijn graphs side by side, connected at the 0-th column.[8]

Definition 9.9. *Let κ and L be two positive integers. The (κ, L)* **double extended De Bruijn graph***, denoted $\mathrm{DDB}(\kappa, L)$, is a 3-regular directed graph consisting of the Cartesian graph product of the directed two-cycle and $\mathrm{DB}(\kappa, L)$. In other words, the vertex set V of $\mathrm{DDB}(\kappa, L)$ consists of vertices $v = (b, i, w)$, where $b \in \{0,1\}$, $i \in \{0, \dots, L-1\}$, and $w \in \{0,1\}^\kappa$, and the edge set E is induced by the following three neighbor functions:*

- $\Gamma_1\big((b, i, w)\big) = (b, i+1 \bmod L, \mathrm{sr}(w))$*, i.e., one kind of De Bruijn edges;*
- $\Gamma_2\big((b, i, w)\big) = (b, i+1 \bmod L, \mathrm{sr}(w) \oplus e_1)$*, i.e., the other kind of De Bruijn edges; and*
- $\Gamma_3\big((b, i, w)\big) = (b \oplus 1, i, w)$*, i.e., edges between corresponding vertices of the two De Bruijn graphs.*

Remark 9.10. A (κ, L) double extended De Bruijn graph is simply two (κ, L) extended De Bruijn graphs "side by side", connected with bi-directional edges at corresponding vertices. (Compare with Definition 6.6, where single De Bruijn graphs are defined.) While the only edges between the two graphs that we will need are the edges between the two 0-th columns, for convenience we still define edges between the two graphs at every node, in order to ensure that the graph is 3-regular.

Given $t \in \mathbb{N}$, let G_t be the graph $\mathrm{DDB}(\kappa, L)$ where $\kappa = t$ and $L = 4t - 1$, i.e., the $(t, 4t-1)$ double extended De Bruijn graph. By Claim 6.7 we can route on G_t any *two* permutations of 2^t elements.

We now define the analogue of a computation graph (Definition 9.3) for double extended De Bruijn graphs, which we call a "computation routing"; intuitively, it is a coloring $R_M^{\vec{S}}$ of the graph G_t such that the 0-th columns of both extended De Bruijn graphs contain the sequence of configurations $\vec{S}$ (whose length we now take without loss of generality to be a power of 2) and other columns may contain any configuration.

Definition 9.11. *Let M be a two-tape random-access machine and $\vec{S} = (S_0, \dots, S_{2^t-1})$ a sequence of configurations for M. We say that a coloring $R_M^{\vec{S}}$ of G_t is a* **computation routing** *for M with respect to $\vec{S}$ if, for every $b \in \{0,1\}$, $i \in \{0, \cdots, L-1\}$, and $w \in \{0,1\}^\kappa$ it holds that*

$$(s, \tau, S) = R_M^{\vec{S}}\big((b, i, w)\big) \in \{0,1\} \times \{0,1\}^\kappa \times \mathcal{S}$$

where $\mathcal{S}$ is the set of all possible configurations of M.

Intuitively, the bit s denotes whether at the given node the routing of the packet, consisting of a "timestamp" τ and configuration S, is done by forwarding the packet "straight ahead" or "diagonally".

Analogously to the notion of a valid computation graph (Definition 9.4), we now define a valid computation routing; intuitively, we need to ensure that, in the columns other than the 0-th one, routing constraints are respected, and that between the last and 0-th columns the appropriate code and memory consistency checks are performed (as well as ensuring that the two 0-th columns are consistent with each other).

Definition 9.12. *Let M be a two-tape random-access machine, $\vec{S} = (S_0, \dots, S_{2^t-1})$ a sequence of configurations for M, (x, w) two input strings, and $R_M^{\vec{S}}$ a computation routing for M with respect to $\vec{S}$.*

- *We say that $R_M^{\vec{S}}$ is* **accepting** *if $\vec{S}$ is accepting. (Recall Definition 7.4.)*
- *We say that $R_M^{\vec{S}}$ is* **consistent** *with (x, w) if $\vec{S}$ is consistent with (x, w). (Recall Definition 7.5.)*
- *We say that $R_M^{\vec{S}}$ is* **valid** *if it satisfies the following conditions:*
  - *(i)* The routing "packets" are initialized correctly. *For every $b \in \{0,1\}$ and $w \in \{0,1\}^\kappa$, it holds that $(s, w, S_w) = R_M^{\vec{S}}((b, 0, w))$ for some s and one of the following conditions holds:*

[8] We note here that one of the two permutations to be routed is always "nice": namely the permutation induced by the "time edges" E_t is always the "$+1 \bmod T$" permutation that, because of its strong locality, does not need an entire permutation network to be routed. (In fact, we could even avoid routing it by adding "fixed" $+1 \bmod T$ edges in the first column of the "memory edge" De Bruijn graph, without bringing in another De Bruijn graph.) However, we still route both permutations, each on a De Bruijn graph, because later on, during arithmetization, we will need a "well-structured" graph.

– $(R_M^{\vec{S}} \circ \Gamma_1)((b,0,w)) = (1,w,S_w)$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_2)((b,0,w))\right)_0 = 1$ *or,*

– $(R_M^{\vec{S}} \circ \Gamma_2)((b,0,w)) = (0,w,S_w)$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_1)((b,0,w))\right)_0 = 0$.

(ii) The routing constraints are respected. *For every* $b \in \{0,1\}$, $i \in \{1,\ldots,L-2\}$, *and* $w \in \{0,1\}^\kappa$, *letting* $(s',\tau',S) = R_M^{\vec{S}}((b,i,w))$, *one of the following conditions holds:*

– $(R_M^{\vec{S}} \circ \Gamma_1)((b,i,w)) = (1,\tau',S)$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_2)((b,i,w))\right)_0 = 1$ *or,*

– $(R_M^{\vec{S}} \circ \Gamma_2)((b,i,w)) = (0,\tau',S)$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_1)((b,i,w))\right)_0 = 0$.

(iii) Code consistency is maintained. *For every* $w \in \{0,1\}^\kappa$, *letting* $(s',\tau',S) = R_M^{\vec{S}}((0,L-1,w))$, *one of the following holds:*

– $(R_M^{\vec{S}} \circ \Gamma_1)((0,L-1,w)) = (1,\tau'+1 \bmod 2^t, S')$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_2)((0,L-1,w))\right)_0 = 1$ *and* $S \rightsquigarrow S'$ *or,*

– $(R_M^{\vec{S}} \circ \Gamma_2)((0,L-1,w)) = (0,\tau'+1 \bmod 2^t, S')$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_1)((0,L-1,w))\right)_0 = 0$ *and* $S \rightsquigarrow S'$.

(iv) Memory consistency is maintained. *For every* $w \in \{0,1\}^\kappa$, *letting* $(s',\tau',S) = R_M^{\vec{S}}((1,L-1,w))$, *one of the following holds:*

– $(R_M^{\vec{S}} \circ \Gamma_1)((1,L-1,w)) = (1,\tau'',S')$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_2)((1,L-1,w))\right)_0 = 1$ *and if both* S *and* S' *contain memory instructions we require that* $(\tau',S) \overset{\mathrm{m}}{\prec} (\tau'',S')$ *and otherwise we require that* $(\tau''=\tau' \ \wedge \ S'=S)$, *or*

– $(R_M^{\vec{S}} \circ \Gamma_2)((1,L-1,w)) = (0,\tau'',S')$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_1)((0,L-1,w))\right)_0 = 0$ *and if both* S *and* S' *contain memory instructions we require that* $(\tau',S) \overset{\mathrm{m}}{\prec} (\tau'',S')$ *and otherwise we require that* $(\tau''=\tau' \ \wedge \ S'=S)$.

(v) Column-0 consistency is maintained. *For any* $w \in \{0,1\}^\kappa$ *it holds that*

$$(R_M^{\vec{S}} \circ \Gamma_3)((1,0,w)) = (s,w,S_w) \ \textit{and} \ (R_M^{\vec{S}} \circ \Gamma_3)((0,0,w)) = (s',w,S_w)$$

for some $s,s' \in \{0,1\}$.

It is easy to see that any two routing on G_t of the same packets of the form $\{(w,S_w)\}_{w\in\{0,1\}^\kappa}$ can be viewed as a computation routing $R_M^{\vec{S}}$ satisfying Item (i), Item (ii), and Item (v) of Definition 9.12.

We now formally establish the connection between computation graphs and computation routings.

Claim 9.13. *Let* M *be a two-tape random-access machine and let* $\vec{S} = (S_0,\ldots,S_{2^t-1})$ *a sequence of configurations for* M, *and* (x,w) *two input strings. Then there exists a computation graph* $G_M^{\vec{S}}$ *for* M *with respect to* $\vec{S}$ *that is valid, accepting, and consistent with* (x,w) *if and only if there exists a computation routing* $R_M^{\vec{S}}$ *for* M *with respect to* $\vec{S}$ *that is valid, accepting, and consistent with* (x,w).

Proof. We prove in **(1)** one direction and in **(2)** the other direction.

(1) Assume that $G_M^{\vec{S}}$ is a computation graph for M with respect to $\vec{S}$ that is valid, accepting and consistent with (x,w). We construct a valid computation routing $R_M^{\vec{S}}$ for M with respect to $\vec{S}$ as follows:

1. First, $R_M^{\vec{S}}$ colors the zero layers G_t such that $R_M^{\vec{S}}((1,0,w)) = R_M^{\vec{S}}((0,0,w)) = (0,w,S_w)$.

2. For any edge $((\tau,S),(\tau',S')) \in E_{\mathsf{m}}$, the coloring $R_M^{\vec{S}}$ of G_t is updated so it expresses the routing of the node $(1,0,\tau)$ to the node $(1,0,\tau')$.

3. For any node $(1,0,\tau)$ not routed in the previous case, the coloring $R_M^{\vec{S}}$ of G_t is updated so it expresses the routing of the node $(1,0,\tau)$ to itself.

4. For any edge $((\tau,S),(\tau+1 \bmod 2^t,S')) \in E_{\mathsf{t}}$, the coloring $R_M^{\vec{S}}$ of G_t is updated so it expresses the routing of the node $(0,0,\tau)$ to the node $(0,0,\tau+1 \bmod 2^t)$.

We now argue that $R_M^{\vec{S}}$ is a valid, accepting and consistent with (x,w) computation routing.

By the first two parts of Definition 9.12 it follows that $R_M^{\vec{S}}$ is accepting and consistent with (x,w).

Regarding the validity requirement, by Lemma 9.6 the edges in E_{m} form a cycle such that every configuration S that contains a load instruction is on the cycle. In addition, we notice that a cycle induces a permutation on its nodes. Also, we notice that Item 3 above also defines a permutation but on the nodes "skipped" by part Item 2. Therefore Item 2 and Item 3 define a permutation on the nodes of the form $(1,0,\tau)$ for some $\tau \in \{0,\ldots,2^t-1\}$. Next we look on the edges used in Item 4 of the construction above. We notice that this requirement also induces a permutation on the nodes of the form $(0,0,\tau)$ for some $\tau \in \{0,\ldots,2^t-1\}$. Because G_t can route any permutation from layer 0 to layer 0 by setting the nodes in the other layers of the graph to be a "straight ahead" or a "diagonal", the two permutations defined by Item 2, Item 3, and Item 4 above can be routed. This fulfills the requirement posed in Item (i) and Item (ii) of Definition 9.12.

Next, by Item 4 of the construction above we have that $(0,0,\tau)$ is routed to $(0,0,\tau+1 \bmod 2^t)$ and, by definition of E_{t}, we have that, for all τ, $S_\tau \rightsquigarrow S_{\tau+1 \bmod 2^t}$. Thus, the requirement of Item (iii) of Definition 9.12 is fulfilled.

We turn our attention to Item 2 and Item 3 of the construction above. Since $G_M^{\vec{S}}$ is a valid computation graph we have that if $(1,0,\tau)$ is routed to $(1,0,\tau')$ then $(\tau,S_\tau) \overset{\mathrm{m}}{\prec} (\tau',S_{\tau'})$ if both S_τ and $S_{\tau'}$ contain memory instructions and $S_\tau = S_{\tau'}$ otherwise. Therefore Item (iv) of Definition 9.12 is also fulfilled. Regarding the requirement posed in Item (v) of Definition 9.12, we notice that Item 1 in the construction of $R_M^{\vec{S}}$ colors all the nodes in layer zero in a way that respects the requirement posed in Item (v) of Definition 9.12. In addition, since all the other items in the construction do not update the configuration part of the colors of layer 0 we obtain that $R_M^{\vec{S}}$ fulfills the requirement of Item (v) of Definition 9.12.

Thus we get that $R_M^{\vec{S}}$ is a valid computation routing.

(2) Conversely, assume that $R_M^{\vec{S}}$ is a valid computation routing. Let $G_M^{\vec{S}} = (V, E_{\mathsf{t}}, E_{\mathsf{m}})$ be the graph that is defined as follows

- $V = \{(\tau, S_\tau) : R_M^{\vec{S}}((0,0,\tau)) = (0,\tau,S_\tau) \text{ and } \tau \in \{0,\ldots,2^t-1\}\}$,
- $E_{\mathsf{t}} = \{((\tau,S_\tau),(\tau',S_{\tau'})) : (0,0,\tau) \text{ is routed to } (0,0,\tau')\}$, and
- $E_{\mathsf{m}} = \{((\tau,S_\tau),(\tau',S_{\tau'})) : (1,0,\tau) \text{ is routed to } (1,0,\tau') \text{ and } S_\tau, S_{\tau'} \text{ both contain a memory instruction}\}$.

We now argue that $G_M^{\vec{S}}$ is a computation graph that is valid, accepting and consistent with (x,w).

We begin by proving that $G_M^{\vec{S}}$ is a computation graph. The first item of Definition 9.3 immediately follows from the definition of V. As for the second item, we notice that by the first two items in Definition 9.12 we obtain that the configuration and timestamp part of the node's color remains unchanged throughout the different layers. These constraints, combined with the constraints of Item (iii) of Definition 9.12, imply that that every $(0,0,\tau)$ is routed to $(0,0,\tau+1 \bmod 2^t)$ and therefore every (τ,S_τ) is routed to $(\tau+1 \bmod 2^t, S_{\tau+1 \bmod 2^t})$. Thus we have obtained that $G_M^{\vec{S}}$ is indeed a computation graph.

Next, we show that $G_M^{\vec{S}}$ is valid, accepting and consistent with (x,w).

Notice that if $R_M^{\vec{S}}$ is accepting and consistent with (x,w) then so is $\vec{S}$. Therefore $G_M^{\vec{S}}$ is accepting and consistent with (x,w) if $R_M^{\vec{S}}$ is. This fulfills the acceptance and consistency requirements as stated in the first two items of Definition 9.4. Next we turn our attention to the validity requirement of Definition 9.4. Indeed, by the first two items in Definition 9.12 we obtain that each node on layer i is routed to exactly one node in layer $i+1 \bmod L$ and that the configuration and time stamp part of the node's color remains unchanged throughout the layers. Moreover, those items also enforce that the routing respects the edge relation of the underlaying graph. Next, Item (iii) and Item (v) of Definition 9.12 requires that every $(0,0,\tau)$ is routed to $(0,0,\tau+1 \bmod 2^t)$ and that $S_\tau \rightsquigarrow S_{\tau+1 \bmod 2^t}$. We thus obtain that $E_{\mathsf{t}} = \{((\tau,S_\tau),(\tau+1,S_{\tau+1 \bmod 2^t})) : (\tau,S_\tau),(\tau+1 \bmod 2^t, S_{\tau+1 \bmod 2^t}) \in V \text{ and } S_\tau \rightsquigarrow S_{\tau+1 \bmod 2^t}\}$. This fulfills the requirements posed by Item (i) of the valid part of Definition 9.4.

Next, as in the previous case, the first two items in Definition 9.12 enforce that each node on layer i is routed to exactly one node in layer $i+1 \bmod L$, that the configuration and time stamp parts of the nodes color remain unchanged throughout the layers and that the routing respects the edge relation of the underlying graph. In addition, any routing done in Item (iv) of Definition 9.12 induces a permutation on the nodes of the form $(1,0,\tau)$ in $G_M^{\vec{S}}$. Furthermore, we require that each configuration that does not contain any memory instruction will be

routed to itself. Thus we obtain that the routing also induces a permutation on the nodes of the form $(1, 0, \tau)$ such that S_τ contains an instruction that accesses the memory. This fulfills the requirement of Item (iii) of Definition 9.4. Item (iv) and Item (v) of Definition 9.12 requires that if $(1, 0, \tau)$ is routed to $(1, 0, \tau')$ and both S_τ and $S_{\tau'}$ contain memory instructions then $(\tau, S_\tau) \overset{\text{m}}{\prec} (\tau', S_{\tau'})$. Therefore, for every edge $((\tau, S_\tau), (\tau', S_{\tau'})) \in E_{\mathsf{m}}$ we have that $(\tau, S_\tau) \overset{\text{m}}{\prec} (\tau', S_{\tau'})$. This fulfills the requirement of Item (ii) of Definition 9.4. □

In light of Claim 9.8 and Claim 9.13, we deduce the following:

Claim 9.14. *Let M be a memory-well-behaved two-tape random-access machine, $\vec{S} = (S_0, \ldots, S_{2^t-1})$ a sequence of configurations for M, and (x, w) two input strings. Then $\vec{S}$ is all-valid and accepting for M and consistent with (x, w) if and only if there exists a computation routing $R_M^{\vec{S}}$ for M with respect to $\vec{S}$ that is valid, accepting, and consistent with (x, w).*

Witness reduction. Note that not only does Claim 9.13 tell us an equivalence between the two decision problems, but its proof tells us how deduce a "witness" too. Specifically, given $G_M^{\vec{S}}$, we can construct a computation routing $R_M^{\vec{S}}$ by deducing the two permutations (for code consistency and memory consistency) by following part **(1)** of the proof. (As for the reverse direction, part **(2)** of the proof gives us that, but we are not interested in "going back".)

9.1.3 Step 3: From (double) De Bruijn graphs to succinct GCPs

We are now ready to discuss succinct GCPs. We note that determining whether a computation routing is valid or not (see Definition 9.12) involves checking different constraints that depend on the vertex (this is what distinguishes constraints (i) through (v) of Definition 9.12), the color of the vertex, and the colors of the neighbors of the vertex. As there are not many different constraints that need to be enforced, we can "bundle up" these into a universal constraint that needs to be enforced at every vertex of the graph — and this is the essence of the of the satisfiability requirement of a succinct GCP problem.

We now discuss then how to derive the parameters of the succinct GCP language corresponding to a given machine M. Of course, there are more details that need to be addressed than discussed in the previous paragraph; specifically, we need to be explicit about the "cost" of creating all the necessary objects, and we need to discuss how consistency with the input string x is achieved.

Thus, using using the template provided in Definition 8.1, we present the following construction:

Construction 9.15. Fix a two-tape random-access machine

$$M = \langle w, k, \mathbb{A}, \mathbb{P} \rangle \ .$$

Construct a choice of parameters

$$\mathsf{par}_{\text{sGCP}} = \Big((\alpha_{\mathbf{G}}, \mathbf{G}), (\mathsf{c}_{\mathbf{C}}, \mathbf{C}), (\mathsf{c}_{\mathbf{T}}, \mathbf{T}), (\mathsf{S}_{\mathbf{M}}, \mathsf{D}_{\mathbf{M}}, \mathsf{t}_{\mathbf{M}}, \mathsf{s}_{\mathbf{M}}, \mathbf{M}), (\mathsf{S}_{\mathbf{K}}, \mathsf{D}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathsf{s}_{\mathbf{K}}, \mathbf{K}), (\mathsf{t}_{\mathbf{W}}, \mathsf{s}_{\mathbf{W}}, \mathbf{W}), (\mathsf{t}_{\mathbf{F}}, \mathsf{s}_{\mathbf{F}}, \mathbf{F}) \Big)$$

for SuccinctGCP as follows:

1. **Constructing Parameter 1.** Define the (proper) regularity function $\alpha_{\mathbf{G}} \colon \mathbb{N} \to \mathbb{N}$ by

$$\alpha_{\mathbf{G}}(t) := 3 \ .$$

2. **Constructing Parameter 2.** Define the graph family $\mathbf{G} = \{G_t\}_{t \in \mathbb{N}}$ by

$$G_t := \text{DDB}(t, 4t - 1) \ ,$$

(Recall Definition 9.9, the definition of a double extended De Bruijn graph.) Note that indeed each G_t is $\alpha_{\mathbf{G}}(t)$-regular. Intuitively, G_t is the graph for a computation routing (Definition 9.11) of a computation of length at most 2^t.

3. **Constructing Parameter 3.** Define the cardinality function $\mathsf{c}_{\mathbf{C}} \colon \mathbb{N} \to \mathbb{N}$ by

$$\mathsf{c}_{\mathbf{C}}(t) := 1 + t + (1 + k)w \ .$$

4. **Constructing Parameter 4.** Define the finite color set family $\mathbf{C} = \{C_t\}_{t\in\mathbb{N}}$ by

$$C_t = \{0,1\}^{\mathsf{c}_{\mathbf{C}}(t)} \ .$$

We have set C_t to be the set of colors of a computation routing. Indeed, recall from Definition 9.11 that a coloring of a computation routing provides for each vertex a bit, a timestamp, and a configuration; thus $\mathsf{c}_{\mathbf{C}}(t) = 1 + t + (1+k)w$, since a timestamp has length t and a configuration has length $(1+k)w$.

5. **Constructing Parameter 5.** Define the cardinality function $\mathsf{c}_{\mathbf{T}} \colon \mathbb{N} \to \mathbb{N}$ by

$$\mathsf{c}_{\mathbf{T}}(t) := 1 + \lceil \log(4t-1) \rceil + t \ .$$

6. **Constructing Parameter 6.** Define the finite vertex-type set family $\mathbf{T} = \{T_t\}_{t\in\mathbb{N}}$ by

$$T_t := \{0,1\}^{\mathsf{c}_{\mathbf{T}}(t)} \ .$$

We have set T_t to be the set of vertices of $G_t = \mathrm{DDB}(t, 4t-1)$, when written in binary.

7. **Constructing Parameter 7.** Define the following (four) proper functions:

$$\begin{aligned} \text{a size function } \mathsf{S}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N}\colon \ & \mathsf{S}_{\mathbf{M}}(t) := O(t) \ , \\ \text{a degree function } \mathsf{D}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N}\colon \ & \mathsf{D}_{\mathbf{M}}(t)[i] := 1 \text{ for } i = 1, \dots, 1 + \lceil \log(4t-1) \rceil + t \ , \\ \text{a time function } \mathsf{t}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N}\colon \ & \mathsf{t}_{\mathbf{M}}(t) := O(t) \ , \\ \text{a space function } \mathsf{s}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N}\colon \ & \mathsf{s}_{\mathbf{M}}(t) := O(t) \ . \end{aligned}$$

8. **Constructing Parameter 8.** Define the family $\mathbf{M} = \{M_t \colon V_t \to T_t\}_{t\in\mathbb{N}}$ by

$$M_t\big((b,i,w)\big) := (b,i,w) \ .$$

In other words, the type of a vertex $v = (b,i,w)$ is simply given by the vertex itself. While the definition of M_t is simple, the definition requires some explanation. At the intuitive level, M_t should "pass on" whichever information is needed to figure out what constraint to apply to the given vertex.

Thus, having Definition 9.12 in mind (which is the definition specifying the constraints we need to enforce): in the first column of each De Bruijn graph we need to initialize the packets correctly and to do so we need to know w for the given vertex; furthermore, in the last column of each De Bruijn graph we need to know whether to check code consistency or memory consistency, and thus we need to know b for the given vertex; finally, to know if we are in the first or last column, we need to know i for the given vertex. Thus, overall, we need to know the identity of the vertex itself.[9]

9. **Constructing Parameter 9.** See next item.

10. **Constructing Parameter 10.** Define the family $\mathbf{K} = \{K_t \colon T_t \times C_t^{1+\alpha_{\mathbf{G}}(t)} \to \{0,1\}\}_{t\in\mathbb{N}}$ to be the function

$$K_t\big(\theta, c_0, c_1, \dots, c_{\alpha_{\mathbf{G}}(t)}\big) \tag{2}$$

that interprets the vertex type θ as a vertex $v \in V_t$, and verifies depending on v the appropriate constraint(s) in the first and third bullet of Definition 9.12.

As for a circuit computing K_t, see Section C.2, where we give explicit circuits.

11. **Constructing Parameter 11.** Define the following two functions:

$$\begin{aligned} \text{a time function } \mathsf{t}_{\mathbf{W}} \colon \mathbb{N} \to \mathbb{N}\colon \ & \mathsf{t}_{\mathbf{W}}(t) := O(t) \ , \\ \text{a space function } \mathsf{s}_{\mathbf{W}} \colon \mathbb{N} \to \mathbb{N}\colon \ & \mathsf{s}_{\mathbf{W}}(t) := O(t) \ . \end{aligned}$$

[9] There are of course some ways to "compress" this information into fewer bits, but we will not bother with these optimizations, as they will not affect the reduction much.

12. **Constructing Parameter 12.** Define the vertex subset family $\mathbf{W} = \{W_t\}_{t \in \mathbb{N}}$ by

$$W_t := \big\{(0, 2i, 0)\big\}_{1 \le i \le 2^{t-1}-1} \ .$$

In other words, we set W_t to be the even-numbered vertices in the 0-th column of the 0-th extended De Bruijn graph in $G_t = \mathrm{DDB}(t, 4t-1)$ (except the 0-th one). This is so because ultimately we will be interested in random-access machines that are memory-well-behaved (see Definition 9.7), so that the first memory cell is accessed in the first time step, and input-well-behaved, so that they then read and store in memory the whole input (see Definition 9.16); in particular, in order to "observe" the input in the configurations we examine the first sufficiently-many even-numbered configurations (after the initial one).

Therefore, we can simply set $(0, 2i, 0) := \textsc{Find}\mathbf{W}(1^t, i)$, and thus both $\mathsf{t}_{\mathbf{W}}(t)$ and $\mathsf{s}_{\mathbf{W}}(t)$ have the above claimed values.

13. **Constructing Parameter 13.** Define the following (three) proper functions:

$$\text{a time function } \mathsf{t}_{\mathbf{F}} \colon \mathbb{N} \to \mathbb{N} \colon \ \mathsf{t}_{\mathbf{F}}(t) := O(t + (1+k)w) \ ,$$
$$\text{a space function } \mathsf{s}_{\mathbf{F}} \colon \mathbb{N} \to \mathbb{N} \colon \ \mathsf{s}_{\mathbf{F}}(t) := O(t + (1+k)w) \ .$$

14. **Constructing Parameter 14.** Define the extraction function family $\mathbf{F} = \{F_t \colon \{0,1\}^* \to \{0,1\}\}_{t \in \mathbb{N}}$ by

$$F_t(x, c_1, \ldots, c_{|x|}) := \text{"0 if and only if } x = \rho_1 \cdots \rho_{|x|}\text{"} \ ,$$

where $c_i = (s_i, \tau_i, S_i)$, $S_i = [\mathsf{pc}_i, r_0^i, \ldots, r_{k-1}^i]$, and $\rho_i = r_j^i$ if the instruction pointed to by $\mathsf{pc} - 1 \bmod 2^t$ is a read from the input tape into register j (else, ρ_i is the empty string). Note that indeed the size of ρ is w bits. In other words, the extraction function F_t, on input x and all the reads from tape A in order, checks that the concatenation of the reads is equal to x.

Furthermore, $\textsc{Comp}\mathbf{F}(1^t, c)$ is trivial because F_t is merely a comparison between two strings.

As we need to check input consistency, we shall restrict (without loss of generality) our attention to random-access machines that read the whole input and store it into memory, without ever accessing the input tape again. This ensures that the input bits appear at "canonical" times of the computation.

Definition 9.16. *Let M be a random-access machine. We say that M is* **input-well-behaved** *if M, starting from the second step, reads all of the input on A and stores it into memory, and then never reads from the input again.*

In sum, a random-access machine is well-behaved if it is both memory-well-behaved and input-well-behaved.

Definition 9.17. *Let M be a random-access machine. We say that M is* well-behaved *if it is both memory-well-behaved and input-well-behaved.*

We now give the equivalence between finding computation routings and membership in $\textsc{Succinct}\mathrm{GCP}(\mathsf{par}_{\mathrm{sGCP}})$ for parameters $\mathsf{par}_{\mathrm{sGCP}}$ constructed as above.

Claim 9.18. *Let M be a well-behaved two-tape random-access machine, x an input string, and $t \in \mathbb{N}$. There exists another input string w, a sequence of configurations $\vec{S} = (S_0, \ldots, S_{T-1})$ for M with $T \le 2^t$, and a computation routing $R_M^{\vec{S}}$ for M with respect to $\vec{S}$ that is valid, accepting, and consistent with (x, w) if and only if $(x, 1^t) \in \textsc{Succinct}\mathrm{GCP}(\mathsf{par}_{\mathrm{sGCP}})$, where $\mathsf{par}_{\mathrm{sGCP}}$ are constructed from M following Construction 9.15.*

Proof. We prove in **(1)** one direction and in **(2)** the other direction.

(1) Assume there exists another input string w, a sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ for M, and a computation routing $R_M^{\vec{S}}$ for M with respect to $\vec{S}$ that is valid, accepting, and consistent with (x, w). We need to prove prove that $(x, 1^t) \in \textsc{Succinct}\mathrm{GCP}(\mathsf{par}_{\mathrm{sGCP}})$.

We now argue that $R_M^{\vec{S}}$ itself is in fact a witness for $(x, 1^t) \in \textsc{Succinct}\mathrm{GCP}(\mathsf{par}_{\mathrm{sGCP}})$.

Following Definition 8.1, we show that the following two conditions hold:

- *Satisfiability of constraints.* The routing $R_M^{\vec{S}}$ satisfies the routing constraints induced by K_t and M_t, i.e., for every $v \in V_t$,

$$K_t\Big(M_t(v), R_M^{\vec{S}}(v), \big(R_M^{\vec{S}} \circ \Gamma_{t,1}\big)(v), \ldots, \big(R_M^{\vec{S}} \circ \Gamma_{t,\alpha_{\mathbf{G}}(t)}\big)(v)\Big) = 0 \ , \quad (3)$$

  where, for $i = 1, \ldots, \alpha_{\mathbf{G}}(t)$, $\Gamma_{t,i}$ is the i-th neighbor function of $G_t = \mathrm{DDB}(t, 4t-1)$. And, indeed, since $R_M^{\vec{S}}$ is a valid computation routing by Item 10 of Construction 9.15 the above equation holds. Thus the coloring $C := R_M^{\vec{S}}$ satisfies the coloring constraints induced by K_t and M_t, as required by Item (i) of Definition 8.1.

- *Consistency with the instance.* The routing $R_M^{\vec{S}}$ is consistent with the instance $(x, 1^t)$, i.e., for every index $i \in \{1, \ldots, |W_t|\}$, letting v_i be the i-th vertex in W_t, $F_t(x, R_M^{\vec{S}}(v_1), \ldots, R_M^{\vec{S}}(v_{|x|})) = 0$.

  Because M is well-behaved, all the input x will be read into memory at the start of the computation right after accessing the first memory cell, so instruction 2 through $2|x| + 1$ will consist of a read_A followed by a store, repeated $|x|$ times. And, indeed, we have defined the i-th vertex of W_t to be the $(2i)$-th vertex in the first column of one De Bruijn graph in V_t, and F_t to verify the appropriate read_A information from its configuration. Thus, once again the same coloring $C := R_M^{\vec{S}}$ is consistent with the instance $(x, 1^t)$, as required by Item (ii) of Definition 8.1.

(2) Suppose that $(x, 1^t) \in \textsc{Succinct}\text{GCP}(\mathsf{par}_{\text{sGCP}})$, and let C be a coloring witnessing this. We need to prove that there exist another input string w, a sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ for M, and a computation routing $R_M^{\vec{S}}$ for M with respect to $\vec{S}$ that is valid, accepting, and consistent with (x, w).

For $i = 0, \ldots, 2^t - 1$, if $C(0, i, 0) = (0, i, S_i')$, define $S_i := S_i'$ and let $S_i = [\mathsf{pc}_i, r_0^i, \ldots, r_{k-1}^i]$. By Item 10 of Definition 9.15 and by the construction of $\vec{S}$, we know that C is a valid and accepting computation routing for M with respect to $\vec{S}$.

Regarding the consistency requirement, consider the string $w = \sigma_1 \cdots \sigma_{2^t-1}$ where for any $i \in \{0, \cdots, 2^t - 1\}$ it holds that $\sigma_i = r_j$ if $\mathsf{pc}_i - 1$ points to a read_B j instruction and $\sigma_i' = \varepsilon$ otherwise. Notice that by construction it holds that C is consistent with w. As for consistency with the input x, since C is a witness to $(x, 1^t) \in \textsc{Succinct}\text{GCP}(\mathsf{par}_{\text{sGCP}})$ it must be the case that C is consistent with x also.

Thus we have obtained that C is a computation routing with respect to $\vec{S}$ that is valid, accepting, and consistent with (x, w). ☐

In light of Claim 9.14 and Claim 9.18, we deduce the following:

Claim 9.19. *Let M be a well-behaved two-tape random-access machine, x an input string, and $t \in \mathbb{N}$. There exists another input string w, and a sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ for M that is all-valid, accepting, and consistent with (x, w) if and only if $(x, 1^t) \in \textsc{Succinct}\text{GCP}(\mathsf{par}_{\text{sGCP}})$, where $\mathsf{par}_{\text{sGCP}}$ are constructed from M following Construction 9.15.*

9.1.4 Step 4: The Levin reduction

We show that the parameter conversion discussed in Section 9.1.4 yields a Levin reduction (according to Definition 9.1) from bhRAM to $\textsc{Succinct}\text{GCP}$ with respect to any class of random-access machines $\mathcal{P}_{\text{RAM}}$ (specifically, any random-access machine as defined in Section 7.2).

More precisely, consider the following definitions:

- Define $F_{\mathsf{p}} \colon \{0,1\}^* \to \{0,1\}^*$ to be the function that, on input a choice of random-access machine $M \in \mathcal{P}_{\text{RAM}}$, first modifies M to ensure it is well-behaved (see Definition 9.17) and then performs the parameter conversion described in Construction 9.15. More precisely, F_{p} works as follows:

$$M = \langle w, k, \mathbb{A}, \mathbb{P} \rangle \ \stackrel{F_{\mathsf{p}}}{\longmapsto} \ \mathsf{par}_{\text{sGCP}} = \begin{pmatrix} \alpha_{\mathbf{G}}, \\ \mathsf{c}_{\mathbf{C}}, \\ \mathsf{c}_{\mathbf{T}}, \\ (\mathsf{S}_{\mathbf{M}}, \mathsf{D}_{\mathbf{M}}, \mathsf{t}_{\mathbf{M}}, \mathsf{s}_{\mathbf{M}}, \textsc{Find}\mathbf{M}), \\ (\mathsf{S}_{\mathbf{K}}, \mathsf{D}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathsf{s}_{\mathbf{K}}, \textsc{Find}\mathbf{K}), \\ (\mathsf{t}_{\mathbf{W}}, \mathsf{s}_{\mathbf{W}}, \textsc{Find}\mathbf{W}), \\ (\mathsf{t}_{\mathbf{F}}, \mathsf{s}_{\mathbf{F}}, \textsc{Comp}\mathbf{F}) \end{pmatrix} ,$$

where the mapping is done by following the definitions of the various new complexity functions and algorithms (for SuccinctGCP) based on the old ones (for M).

- Define $F_{\mathsf{w}} \colon \{0,1\}^* \to \{0,1\}^*$ as follows: for every $M \in \mathcal{P}_{\mathrm{RAM}}$, $t \in \mathbb{N}$, $w \in \{0,1\}^*$, and $\vec{S}$,

  $F_{\mathsf{w}}(M, 1^{2^t}, (w, \vec{S})) \equiv$
  1. Route $\vec{S}$ on the graph $G_t := \mathrm{DDB}(\kappa, L)$, where $\kappa = t$ and $L = 4\kappa - 1$, by computing the routing for the code-consistency permutation and the memory-consistency permutation, and let C be the resulting coloring.
  2. Output C.

We prove the following theorem:

Theorem 9.20. *The pair of functions $(F_{\mathsf{p}}, F_{\mathsf{w}})$ is a Levin reduction from* bhRAM *to* SuccinctGCP *with respect to any class of random-access machines $\mathcal{P}_{\mathrm{RAM}}$.*

We divide the proof of Theorem 9.20 into three claims:
- in Claim 9.21, we explain why both F_{p} and F_{w} are polynomial-time computable;
- in Claim 9.22, we show the "completeness" and "soundness" of F_{p}; and
- in Claim 9.23, we show that F_{w} produces good witnesses.

Claim 9.21. *The functions F_{p} and F_{w} are polynomial-time computable.*

Proof. The efficiency of F_{p} easily follows by inspection of how the parameters are converted in Construction 9.15. (Essentially, all the new functions and algorithms are "easy combinations" of previous functions and algorithms, and thus not hard to write down.) Also note that ensuring that M is well-behaved is tantamount to small changes to the program of M. The efficiency of F_{w} easily follows from the fact that it can run in time that is polynomial in 1^{2^t}, which is plenty of time for computing the routing of $\vec{S}$ on G_t. (Indeed, recall Claim 6.7.) □

Claim 9.22. *For every choice of machine $M \in \mathcal{P}_{\mathrm{RAM}}$ and for every instance $(x, 1^t) \in \{0,1\}^*$, $(x, 1^t) \in$ bhRAM(M) if and only if $(x, 1^t) \in$ SuccinctGCP$(F_{\mathsf{p}}(M))$.*

Proof. Follows directly from Claim 9.19 and Definition 7.9 (which says that $(x, 1^t) \in$ bhRAM(M) if and only if there is another input string w, and a sequence of configurations $\vec{S} = (S_0, \dots, S_{2^t-1})$ for M that is all-valid, accepting, and consistent with (x, w)). □

Claim 9.23. *For every choice of machine $M \in \mathcal{P}_{\mathrm{RAM}}$ and for every instance $(x, 1^t) \in \{0,1\}^*$, if $(w, \vec{S})$ is a witness to "$(x, 1^t) \in$ bhRAM(M)" then $F_{\mathsf{w}}(M, 1^{2^t}, (w, \vec{S}))$ is a witness to "$(x, 1^t) \in$ SuccinctGCP$(F_{\mathsf{p}}(M))$".*

Proof. The claim follows immediately from the fact that in the proof of the "completeness" direction of the statement from Claim 9.22 (or, more precisely, from the proof of Claim 9.18, which implies Claim 9.19, and in turn Claim 9.22), we have constructed, starting from a valid pair $(w, \vec{S})$ and according to F_{p}, a valid coloring C for $(x, 1^t)$: by routing $\vec{S}$ on G_t. □

9.2 A Computational Levin Reduction

In this section we shall construct a *computational* Levin reduction.

Definition 9.24. *Fix a class of random-access machines $\mathcal{P}_{\mathrm{RAM}}$. We say that a pair of polynomial-time-computable functions $(F_{\mathsf{p}}, F_{\mathsf{w}})$ is a* **computational Levin reduction** *from* bhRAM *to* SuccinctGCP *with respect to $\mathcal{P}_{\mathrm{RAM}}$ if the following three conditions are satisfied for every choice of random-access machine $M \in \mathcal{P}_{\mathrm{RAM}}$:*

1. *$F_{\mathsf{p}}(M, s)$ is a choice of parameters for* SuccinctGCP *for every $\kappa \in \mathbb{N}$ and $s \in \{0,1\}^\kappa$.*

2. *For every instance $(x, 1^t) \in \{0,1\}^*$, if $(x, 1^t) \in$ bhRAM(M) then $(x, 1^t) \in$ SuccinctGCP$(F_{\mathsf{p}}(M, s))$ for every $\kappa \in \mathbb{N}$ and $s \in \{0,1\}^\kappa$.*

3. *For every polynomial-size circuit family* $\{C_\kappa\}_{\kappa\in\mathbb{N}}$ *and every non-negative* c *there exists* $K \in \mathbb{N}$ *such that, for every* $\kappa \geq K$,

$$\Pr_{s\leftarrow\{0,1\}^\kappa}\left[\begin{array}{c}(x,1^t)\notin \text{BHRAM}(M)\\ C \textit{ is witness to } (x,1^t)\in \text{SUCCINCTGCP}(F_{\mathsf{p}}(M,s))\end{array}\middle| (x,t,C)\leftarrow C_\kappa(s)\right]\leq \frac{1}{\kappa^c} \ .$$

4. *For every instance* $(x,1^t)\in\{0,1\}^*$, *if* w *is a witness to "*$(x,1^t)\in \text{BHRAM}(M)$*" then* $F_{\mathsf{w}}(M,1^{2^t},w,s)$ *is a witness to "*$(x,1^t)\in \text{SUCCINCTGCP}(F_{\mathsf{p}}(M,s))$*" for every* $\kappa\in\mathbb{N}$ *and* $s\in\{0,1\}^\kappa$.

The two functions F_{p} *and* F_{w} *are respectively called as the "parameter reduction" and the "witness reduction".*

Motivation. In Section 9.1 we have already constructed a Levin reduction from bounded-halting problems on random-access machines to succinct GCPs, even without relying on any computational assumptions; in that reduction we leveraged routing techniques to to enforce both code and memory consistency on a fixed graph structure (namely, a routing network).

We now show that routing techniques are *not necessary*, provided that one is willing to make computational assumptions: we construct a *computational Levin reduction*, following Definition 9.24, from BHRAM to SUCCINCTGCP on cyclic graphs, avoiding routing networks altogether.

High-level idea. Using collision-resistant hash functions, we observe that a random-access machine can in fact check memory consistency *by itself*, with only a small computational overhead: the machine can simply dynamically maintain a Merkle tree over memory. With this in mind, the first step of the reduction is to modify a given machine into one that checks its own memory. This step is where we use computational assumptions.

Next, we are left to only enforce code consistency. But code consistency is a constraint that is much more "well-behaved" than memory consistency, and is in fact simple enough that no routing is needed to enforce it on a fixed graph structure. Thus, as a second step, we construct an information-theoretic reduction from bounded-halting problems for random-access machines with *untrusted* memory to succinct GCPs on the much simpler cyclic graphs.

Our plan, step by step. We construct the Levin reduction in four steps:

- **Step 1 (Section 9.2.1).** We show how, for a given random-access machine M, the problem of deciding whether an instance $(x,1^t)$ is in BHRAM(M) can be "computationally reduced" to the problem of deciding whether the same instance $(x,1^t)$ is in BHURAM(M_s), where M_s is a random-access machine that can be easily derived from M when given a seed s for a collision-resistant hash function h_s.

- **Step 2 (Section 9.2.2).** We show how the problem of deciding whether an instance $(x,1^t)$ is in BHURAM(M) can be reduced to the problem of deciding whether the same instance $(x,1^t)$ is in SUCCINCTGCP$(\mathsf{par}_{\text{sGCP}})$ for an appropriate choice of $\mathsf{par}_{\text{sGCP}}$ depending on M, and where the underlying graph is very simple: it is a cyclic graph.

- **Step 3 (Section 9.2.3).** We conclude by writing down explicitly the two functions F_{p} and F_{w} of the Levin reduction and explaining why they are correct.

We now proceed to describe each of the three steps in detail. Throughout, it will be useful to have the definitions from Section 7 in mind.

Remark 9.25. The results we discuss in this section also hold for an adversarially-chosen machine M (possibly based on the seed s). For simplicity, our presentation will focus on the case where M is chosen upfront.

9.2.1 Step 1: From RAM to untrusted RAMs via Merkle trees

Recall from Definition 7.9 that in order to verify whether a given instance $(x,1^t)$ is in BHRAM(M) for a given random-access machine M via a witness w, we need to run the machine M "correctly" on input (x,w) and check whether it accepts within at most t steps. Here correctness means two things: code consistency, namely the state of the machine progresses according to the transition function, and memory consistency, namely accessing a certain address in memory results in the last value that was stored there.

However, we could simply not bother to enforce memory consistency if M were to maintain its own memory, by performing appropriate consistency checks when retrieving values from memory that are "not trusted". So now we discuss how to modify M in a way that it does maintain its own memory.

Securing untrusted storage via Merkle trees. The observation here is simple: given a seed s for a collision-resistant hash, we can modify M into a new machine M_s that dynamically maintain a Merkle tree over the untrusted memory. The machine M will always store in its local state a root of the Merkle tree; then, every time M wishes to retrieve a value from memory, it will also retrieve appropriate authentication paths to check that indeed the value claimed from memory is a good one; every time M wishes to store a value in memory, it will update the authentication path information and the locally-stored root.

Of course, Merkle trees are known as a valuable tool to secure untrusted storage, but here we see that they have the surprising application of implying computational Levin reductions from random-access machines to weaker models of computation, so that further reductions may be simplified.

We do not present here the details of the transformation from M to M_s, but the formal statement and its proof are given in Appendix D.1.

Remark 9.26 (More Untrusted Components)**.** When reasoning about the correctness of a random-access machine, there are three components that may be larger than one is willing to pay: its memory, its code, and its input.

As discussed above, we have showed that we can without loss of generality "forget" about memory, and only worry about code consistency, which involves reasoning about states that are of fixed size. The use of collision-resistant hashes turns out to be a simple paradigm that can *also* be applied to the input of the machine or the code of the machine as well (or any combination of these); we discuss these additional cases, as well as the memory one, in Appendix D.

9.2.2 Step 2: From untrusted RAMs to succinct GCPs

We give an information-theoretic Levin reduction from bounded-halting problems on random-access machines with untested memory to succinct GCP problems on cyclic graphs. As discussed, because now we do not have to worry about memory consistency, we only need to enforce code consistency, which we can do over a cyclic graph because we only need to correctly enforce progress of the transition function of the machine, which is a very local constraint that only involves the current configuration and the next.

We begin with the definition of a cyclic graph (in a slightly more general form than what we will need, as the subsequent arithmetization will be able to handle this more general form):

Definition 9.27 (Cyclic Graph)**.** *Let α and β be positive integers with $\alpha \leq \beta$. The (α, β)* **cyclic graph***, denoted $\textsc{Cyc}(\alpha, \beta)$, is a directed graph with β vertices labeled by the integers $0, \ldots, \beta - 1$. Each vertex $v \in \{0, \ldots, \beta - 1\}$ has α neighbors, with labels $\Gamma_1(v) = v + 1 \bmod \beta, \ldots, \Gamma_\alpha(v) = v + \alpha \bmod \beta$.*

We now discuss how to derive the parameters of the succinct GCP problem corresponding to a given machine M. As usual, there are more details that need to be addressed than the high-level intuition; specifically, we need to be explicit about the "cost" of creating all the necessary objects, and we need to discuss how consistency with the input string x is achieved.

Thus, using using the template provided in Definition 8.1, we present the following construction:

Construction 9.28. Construct a choice of parameters

$$\mathsf{par}_{\mathrm{sGCP}} = \Big((\alpha_{\mathbf{G}}, \mathbf{G}), (\mathsf{c}_{\mathbf{C}}, \mathbf{C}), (\mathsf{c}_{\mathbf{T}}, \mathbf{T}), (\mathsf{S}_{\mathbf{M}}, \mathsf{D}_{\mathbf{M}}, \mathsf{t}_{\mathbf{M}}, \mathsf{s}_{\mathbf{M}}, \mathbf{M}), (\mathsf{S}_{\mathbf{K}}, \mathsf{D}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathsf{s}_{\mathbf{K}}, \mathbf{K}), (\mathsf{t}_{\mathbf{W}}, \mathsf{s}_{\mathbf{W}}, \mathbf{W}), (\mathsf{t}_{\mathbf{F}}, \mathsf{s}_{\mathbf{F}}, \mathbf{F})\Big)$$

for $\textsc{Succinct}$GCP as follows:

1. **Constructing Parameter 1.** Define the (proper) regularity function $\alpha_{\mathbf{G}} \colon \mathbb{N} \to \mathbb{N}$ by
$$\alpha_{\mathbf{G}}(t) := 1 \enspace .$$
2. **Constructing Parameter 2.** Define the graph family $\mathbf{G} = \{G_t\}_{t \in \mathbb{N}}$ by
$$G_t := \textsc{Cyc}(1, 2^t) \enspace .$$
In other words, we chose a cyclic graph with 2^t vertices, each vertex "pointing" to the next one. Note that, indeed, G_t is $\alpha_{\mathbf{G}}(t)$-regular.

3. **Constructing Parameter 3.** Define the cardinality function $\mathsf{c}_{\mathbf{C}} \colon \mathbb{N} \to \mathbb{N}$ by

$$\mathsf{c}_{\mathbf{C}}(t) := (1+k)w \ .$$

That is, we set the number of bits of a color to be equal to the number of bits in a configuration of M. (Recall Definition 7.2.)

4. **Constructing Parameter 4.** Define the finite color set family $\mathbf{C} = \{C_t\}_{t \in \mathbb{N}}$ by

$$C_t = \{0,1\}^{\mathsf{c}_{\mathbf{C}}(t)} \ .$$

That is, we define the set of colors to be equal to the set of possible configuration of M.

5. **Constructing Parameter 5.** Define the cardinality function $\mathsf{c}_{\mathbf{T}} \colon \mathbb{N} \to \mathbb{N}$ by

$$\mathsf{c}_{\mathbf{T}}(t) := 1 \ .$$

6. **Constructing Parameter 6.** Define the finite vertex-type set family $\mathbf{T} = \{T_t\}_{t \in \mathbb{N}}$ by

$$T_t := \{0,1\}^{\mathsf{c}_{\mathbf{T}}(t)} \ .$$

That is, we define the set of types to be just $\{0,1\}^{\mathsf{c}_{\mathbf{T}}(t)} = \{0,1\}$; intuitively, the type will denote whether the vertex is the last one in the cyclic graph or not.

7. **Constructing Parameter 7.** Define the following (four) proper functions:

$$\begin{aligned} \text{a size function } \mathsf{S}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N} \colon \ & \mathsf{S}_{\mathbf{M}}(t) := O(t) \ , \\ \text{a degree function } \mathsf{D}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N} \colon \ & \mathsf{D}_{\mathbf{M}}(t)[i] := 1 \text{ for } i = 1, \ldots, t \ , \\ \text{a time function } \mathsf{t}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N} \colon \ & \mathsf{t}_{\mathbf{M}}(t) := O(t) \ , \\ \text{a space function } \mathsf{s}_{\mathbf{M}} \colon \mathbb{N} \to \mathbb{N} \colon \ & \mathsf{s}_{\mathbf{M}}(t) := O(t) \ . \end{aligned}$$

8. **Constructing Parameter 8.** Define the family $\mathbf{M} = \{M_t \colon V_t \to T_t\}_{t \in \mathbb{N}}$ by

$$M_t(v) := \begin{cases} 1 & \text{if } v = 2^t - 1 \\ 0 & \text{otherwise} \end{cases} \ .$$

That is, M_t sets the type of a vertex v in $\textsc{Cyc}(1, 2^t)$ to be equal to 1 if and only if v is the last vertex in the graph.

9. **Constructing Parameter 9.** See next item.

10. **Constructing Parameter 10.** Define the family $\mathbf{K} = \{K_t \colon T_t \times C_t^{1+\alpha_{\mathbf{G}}(t)} \to \{0,1\}\}_{t \in \mathbb{N}}$ by

$$K_t\big(\theta, c_0, c_1, \ldots, c_{\alpha_{\mathbf{G}}(t)}\big) \tag{4}$$

where K_t is the function that, interpreting the vertex type θ as a bit and the two colors c_0 and c_1 as configurations, checks that $\delta_M(c_0, c_1) = 0$ and, moreover, that (a) if $\theta = 1$ (i.e., we currently are at the last vertex of the graph) then c_0 is an accepting configuration of M, and (b) if c_0 is a final configuration of M then $\theta = 1$. (Recall from Definition 7.7 that δ_M is the transition function of M.)

See Section C.3 for a circuit computing K_t.

11. **Constructing Parameter 11.** Define the following two functions:

$$\begin{aligned} \text{a time function } \mathsf{t}_{\mathbf{W}} \colon \mathbb{N} \to \mathbb{N} \colon \ & \mathsf{t}_{\mathbf{W}}(t) := O(t) \ , \\ \text{a space function } \mathsf{s}_{\mathbf{W}} \colon \mathbb{N} \to \mathbb{N} \colon \ & \mathsf{s}_{\mathbf{W}}(t) := O(t) \ . \end{aligned}$$

12. **Constructing Parameter 12.** Define the vertex subset family $\mathbf{W} = \{W_t\}_{t\in\mathbb{N}}$ by

$$W_t := \{2i\}_{1\le i\le 2^{t-1}-1} \ .$$

In other words, we set W_t to be the even-numbered vertices in $\textsc{Cyc}(1, 2^t)$ (except the 0-th one). This is so because ultimately we will be interested in random-access machines that are memory-well-behaved (see Definition 9.7), so that the first memory cell is accessed in the first time step, and input-well-behaved, so that they then read and store in memory the whole input (see Definition 9.16); in particular, in order to "observe" the input in the configurations we examine the first sufficiently-many odd-numbered configurations.

Therefore, we can simply set $i := \textsc{Find}\mathbf{W}(1^t, i)$, and thus both $\mathsf{t}_{\mathbf{W}}(t)$ and $\mathsf{s}_{\mathbf{W}}(t)$ have the above claimed values.

13. **Constructing Parameter 13.** Define the following (three) proper functions:

$$\text{a time function } \mathsf{t}_{\mathbf{F}} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{t}_{\mathbf{F}}(t) := O((1+k)w) \ ,$$
$$\text{a space function } \mathsf{s}_{\mathbf{F}} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{s}_{\mathbf{F}}(t) := O((1+k)w) \ .$$

14. **Constructing Parameter 14.** Define the extraction function family $\mathbf{F} = \{F_t \colon \{0,1\}^* \to \{0,1\}\}_{t\in\mathbb{N}}$ by

$$F_t(x, c_1, \ldots, c_{|x|}) := \text{"0 if and only if } x = \rho_1 \cdots \rho_{|x|}\text{"} \ ,$$

where $c_i = S_i = [\mathsf{pc}_i, r_0^i, \ldots, r_{k-1}^i]$, and $\rho_i = r_j^i$ if the instruction pointed to by $\mathsf{pc} - 1 \bmod 2^t$ is a read_A instruction into register j (if that instruction is not a read_A instruction, ρ_i is the empty string). Note that indeed the size of ρ is w bits. In other words, the extraction function F_t, on input x and all the reads from tape A in order, checks that the concatenation of the reads is equal to x.

Furthermore, $\textsc{Comp}\mathbf{F}(1^t, c)$ is trivial because F_t is merely a comparison between two strings.

We now give the equivalence between $\textsc{bh}$URAM and membership in $\textsc{Succinct}$GCP$(\mathsf{par}_{\mathrm{sGCP}})$ for parameters $\mathsf{par}_{\mathrm{sGCP}}$ constructed as above.

Claim 9.29. *Let M be a well-behaved two-tape random-access machine. Then, for any $(x, 1^t)$ it holds that $(x, 1^t) \in \textsc{bh}\mathrm{URAM}(M)$ if and only if $(x, 1^t) \in \textsc{Succinct}\mathrm{GCP}(\mathsf{par}_{\mathrm{sGCP}})$, where $\mathsf{par}_{\mathrm{sGCP}}$ are constructed from M following Construction 9.15.*

Proof. We prove in **(1)** one direction and in **(2)** the other direction.

(1) Let M be a well-behaved two-tape random-access machine and let $(x, 1^t) \in \textsc{bh}\mathrm{URAM}(M)$. Thus, there exists there exists another input string w, and a sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ for M such that $\vec{S}$ is code-valid, accepting, and consistent with (x, w).

Define the coloring $C \colon V_t \to C_t$ as follows: for every i, letting v_i be the i-th vertex in V_t,

$$C(v_i) := S_i \ .$$

We now argue that C is a witness for $(x, 1^t) \in \textsc{Succinct}\mathrm{GCP}(\mathsf{par}_{\mathrm{sGCP}})$.

Following Definition 8.1, we show that the following two conditions hold:

- *Satisfiability of constraints.* The coloring C satisfies the constraints induced by K_t and M_t, i.e., for every $v \in V_t$,

$$K_t\Big(M_t(v), C(v), \big(C \circ \Gamma_{t,1}\big)(v), \ldots, \big(C \circ \Gamma_{t,\alpha_{\mathbf{G}}(t)}\big)(v)\Big) = 0 \ , \tag{5}$$

  where, for $i = 1, \ldots, \alpha_{\mathbf{G}}(t)$, $\Gamma_{t,i}$ is the i-th neighbor function of $G_t = \textsc{Cyc}(1, 2^t)$. And, indeed, since $\vec{S}$ is code-valid and accepting for M, by construction of K_t and $M_t(v)$, the above equation holds.

- *Consistency with the instance.* The coloring C is consistent with the instance $(x, 1^t)$, i.e., for every index $i \in \{1, \ldots, |W_t|\}$, letting v_i be the i-th vertex in W_t, $F_t(x, C(v_1), \ldots, C(v_{|x|})) = 0$.

  Because M is well-behaved, all the input x will be read into memory at the start of the computation right after accessing the first memory cell, so instruction 2 through $2|x| + 1$ will consist of a read_A followed by a store, repeated $|x|$ times. And, indeed, we have defined the i-th vertex of W_t to be the $2i$-th vertex in $G_t = \textsc{Cyc}(1, 2^t)$ and define F_t to verify the appropriate read_A information from its configuration.

(2) Suppose that $(x, 1^t) \in \textsc{SuccinctGCP}(\mathsf{par}_{\text{sGCP}})$, and let C be a coloring witnessing this. We need to prove that there exist another input string w and a sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ for M such that $\vec{S}$ is code-valid, accepting, and consistent with (x, w).

For $i = 0, \ldots, 2^t - 1$, let $C(i) = S'_i$ and define $S_i := S'_i$. In addition, let $S_i = [\mathsf{pc}_i, r_0^i, \ldots, r_{k-1}^i]$. By Item 10 of Definition 9.28 and by the construction of $\vec{S}$, we know that $\vec{S}$ is a code-valid and accepting sequence of configurations for M.

Regarding the consistency requirement, consider the string $w = \sigma_1 \cdots \sigma_{2^t-1}$ where for any $i \in \{0, \cdots, 2^t - 1\}$ it holds that $\sigma_i = r_j$ if $\mathsf{pc}_i - 1$ points to a $\mathsf{read}_B\ j$ instruction and $\sigma'_i = \varepsilon$ otherwise. Notice that by construction it holds that C is consistent with w. As for consistency with the input x, since C is a witness to $(x, 1^t) \in \textsc{SuccinctGCP}(\mathsf{par}_{\text{sGCP}})$ by the Item (ii) of Definition 8.1 and by the way we construct $\vec{S}$ it must be the case that $\vec{S}$ is consistent with x also.

Thus we have obtained that $\vec{S}$ is a sequence of configurations that is code-valid, accepting, and consistent with (x, w). □

9.2.3 Step 3: The computational Levin reduction

We show that the parameter conversion discussed in Section 9.2.2, together with the discussion in Section 9.2.1, yields a computational Levin reduction (according to Definition 9.24) from bhRAM to $\textsc{SuccinctGCP}$ with respect to any class of random-access machines $\mathcal{P}_{\text{RAM}}$ (specifically, any random-access machine as defined in Section 7.2).

More precisely, consider the following definitions:

- Define $F_{\mathsf{p}} \colon \{0,1\}^* \to \{0,1\}^*$ to be the function that, on input a choice of random-access machine $M \in \mathcal{P}_{\text{RAM}}$ and a seed s for a collision-resistant hash, first modifies M to ensure it is well-behaved (see Definition 9.17), then modifies M according to the "untrusted-memory transformation" presented of Section 9.2.1 (and in more detail in Appendix D.1) to obtaining the new random access machine M_s and then performs the parameter conversion described in Construction 9.28:

$$M_s = \langle w, k, \mathbb{A}, \mathbb{P} \rangle \xmapsto{F_{\mathsf{p}}} \mathsf{par}_{\text{sGCP}} = \begin{pmatrix} \alpha_{\mathbf{G}}, \\ \mathsf{c}_{\mathbf{C}}, \\ \mathsf{c}_{\mathbf{T}}, \\ (\mathsf{S}_{\mathbf{M}}, \mathsf{D}_{\mathbf{M}}, \mathsf{t}_{\mathbf{M}}, \mathsf{s}_{\mathbf{M}}, \textsc{Find}\mathbf{M}), \\ (\mathsf{S}_{\mathbf{K}}, \mathsf{D}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathsf{s}_{\mathbf{K}}, \textsc{Find}\mathbf{K}), \\ (\mathsf{t}_{\mathbf{W}}, \mathsf{s}_{\mathbf{W}}, \textsc{Find}\mathbf{W}), \\ (\mathsf{t}_{\mathbf{F}}, \mathsf{s}_{\mathbf{F}}, \textsc{Comp}\mathbf{F}) \end{pmatrix},$$

  where the mapping is done by following the definitions of the various new complexity functions and algorithms (for $\textsc{SuccinctGCP}$) based on the old ones (for M_s).

- Define $F_{\mathsf{w}} \colon \{0,1\}^* \to \{0,1\}^*$ as follows: for every $M \in \mathcal{P}_{\text{RAM}}$, $t \in \mathbb{N}$, $w \in \{0,1\}^*$, and $\vec{S} = (S_0, \ldots, S_{2^t-1})$,

  $F_{\mathsf{w}}(M, 1^{2^t}, w, s) \equiv$
  1. Modify M into M_s as described above; run $M_s(x, w)$ to obtain a sequence of configurations $\vec{S}$.
  2. Color $\vec{S}$ on the graph $G_t := \textsc{Cyc}(1, 2^t)$ by setting the color of the i-th node to be S_i and let C be the resulting coloring.
  3. Output C.

We prove the following theorem:

Theorem 9.30. *The pair of functions $(F_{\mathsf{p}}, F_{\mathsf{w}})$ is a Levin reduction from* bhRAM *to* $\textsc{SuccinctGCP}$ *with respect to any class of random-access machines $\mathcal{P}_{\text{RAM}}$.*

We divide the proof of Theorem 9.30 into three claims:
- in Claim 9.31, we explain why both F_{p} and F_{w} are polynomial-time computable;
- in Claim 9.32, we show the "completeness" and "soundness" of F_{p}; and
- in Claim 9.33, we show that F_{w} produces good witnesses.

Claim 9.31. *The functions F_{p} and F_{w} are polynomial-time computable.*

Proof. The efficiency of F_{p} easily follows by inspection of how the parameters are converted in Construction 9.28 and in Appendix D.1. (Essentially, all the new functions and algorithms are "easy combinations" of previous functions and algorithms, and thus not hard to write down.) Also note that ensuring that M is well-behaved is tantamount to small changes to the program of M. The efficiency of F_{w} easily follows from the fact that it can run in time that is polynomial in 1^{2^t}, which is plenty of time for computing the coloring of $\vec{S}$ on G_t. □

Claim 9.32. *For every choice of machine $M \in \mathcal{P}_{\text{RAM}}$ and for every polynomial-size circuit family $\{C_\kappa\}_{\kappa\in\mathbb{N}}$ and every non-negative c there exists $K \in \mathbb{N}$ such that, for every $\kappa \geq K$,*

$$\Pr_{s \leftarrow \{0,1\}^\kappa}\left[\begin{array}{c} (x, 1^t) \notin \text{BHRAM}(M) \\ C \text{ is witness to } (x,1^t) \in \text{SuccinctGCP}(F_{\mathsf{p}}(M,s)) \end{array} \middle| (x,t,C) \leftarrow C_\kappa(s)\right] \leq \frac{1}{\kappa^c} \ .$$

Proof. Follows directly from Claim 9.29, Lemma D.2 and Definition 7.9 (which says that $(x, 1^t) \in \text{BHURAM}(M)$ if and only if there is another input string w, and a sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ for M that is code-valid, accepting, and consistent with (x, w)). □

Claim 9.33. *For every choice of machine $M \in \mathcal{P}_{\text{RAM}}$, seed s, and instance $(x, 1^t) \in \{0,1\}^*$, if w is a witness to "$(x, 1^t) \in \text{BHRAM}(M)$" then $F_{\mathsf{w}}(M, 1^{2^t}, w, s)$ is a witness to "$(x, 1^t) \in \text{SuccinctGCP}(F_{\mathsf{p}}(M,s))$".*

Proof. The claim follows immediately from the fact that in the proof of the "completeness" direction of the statement from Claim 9.29 where we have constructed, starting from a valid w, a sequence of configurations $\vec{S}$ and then a valid coloring C for $(x, 1^t)$. □

Remark 9.34. We note that in our definition of a random-access machine with untrusted memory we require that the machine only trusts a small and finite number of registers each of which has finite width w. Thus, Construction 9.28 above can be easily generalized to a reduction between a bounded halting problem for finite automata to a universal graph coloring problem as follows. Let $A = (\Sigma, Q, q_0, \delta, F)$ be an finite automaton. We can color a run of A on input x of length 2^t on a cyclic graph $G_t = \text{Cyc}(1, 2^t)$ by setting the colors of G_t to be the states of A and defining the circuit K_t on input (v_1, v_2) to output 0 if and only if there exists σ such that $C(v_2) \in \delta(C(v_1), \sigma)$. We pad the transitions with $\perp$ to make the length of the run to be a power of 2. Obviously this satisfies satisfiability of constraints of Definition 8.1. As for the instance consistency requirement, by [HU79, Theorem 2.2] we can assume that A reads an input letter at each step but the padded steps. Thus, by making F_t in input $x = x_1 \cdots x_n$ to verify that for any $0 \leq i < n-1$ it holds that $C(v_{i+1}) \in \delta(C(v_i), x_i)$ we obtain that the instance consistency requirement holds.

9.3 Tradeoff Between The Two Reductions

The two reductions that we presented in the previous two subsections are comparable in efficiency but offer different advantages. Concretely, the reduction in Section 9.1 has unconditional guarantees but requires large space complexity on the "prover side" to deduce a valid coloring from a valid witness, due to the routing algorithm. In contrast, the reduction in Section 9.2 only has computational soundness but the color of any given vertex can be deduced easily with little space from a given valid witness. Furthermore, this second reduction "scales" well with memory size: in general, the overhead is not $O(\log T) = O(t)$, but only $O(\log S)$ where S is the space complexity of the original random-access machine.

10 A Generic Succinct Algebraic Constraint Satisfaction Problem

We define a family of constraint satisfaction problems about polynomials in a finite field; roughly, each element in the finite field induces an "affine neighborhood", and membership of an instance in the language is determined by whether there exists a low-degree polynomial, vanishing at every affine neighborhood over a certain subset of the field, that "contains" the given instance. It is for this family of problems that [BSCGT12] construct practical PCPs.

We begin with the formal definition of the class of problems; then we discuss the intuition behind the design of the definition, and explain why we believe that it is a great class of problems for which it is worthwhile to construct practical PCPs.

Definition 10.1 (Succinct Algebraic Constraint Satisfaction)**.** Consider the following parameters:

1. a field size function $f\colon \mathbb{N} \to \mathbb{N}$, inducing a family of finite fields $\{\mathbb{F}_t\}_{t\in\mathbb{N}}$ where $\mathbb{F}_t = \mathbb{F}_2(\mathsf{x})$ and x is the root of I_t, which is the irreducible polynomial of degree $f(t)$ over $\mathbb{F}_2$ output by $\textsc{FindIrrPoly}(1^{f(t)})$;
2. three functions associated with the family $\mathbf{H}$ in Parameter 3:
   (a) a dimension function $\mathsf{m}_{\mathbf{H}}\colon \mathbb{N} \to \mathbb{N}$,
   (b) a time function $\mathsf{t}_{\mathbf{H}}\colon \mathbb{N} \to \mathbb{N}$, and
   (c) a space function $\mathsf{s}_{\mathbf{H}}\colon \mathbb{N} \to \mathbb{N}$;
3. a family $\mathbf{H} = \{H_t\}_{t\in\mathbb{N}}$ such that:
   (a) H_t is a linear subset of $\mathbb{F}_t$,
   (b) $\dim(H_t) = \mathsf{m}_{\mathbf{H}}(t)$,
   (c) each linear subset H_t is specified by a basis $\big(\alpha_1^{\mathbf{H}}(\mathsf{x}), \dots, \alpha_{\mathsf{m}_{\mathbf{H}}(t)}^{\mathbf{H}}(\mathsf{x})\big)$, and
   (d) there exists a $\mathsf{t}_{\mathbf{H}}$-time $\mathsf{s}_{\mathbf{H}}$-space algorithm $\textsc{Find}\mathbf{H}$ such that $\textsc{Find}\mathbf{H}(1^t) = \big(\alpha_1^{\mathbf{H}}(\mathsf{x}), \dots, \alpha_{\mathsf{m}_{\mathbf{H}}(t)}^{\mathbf{H}}(\mathsf{x})\big)$ for all $t \in \mathbb{N}$;
4. three functions associated with the family $\mathbf{N}$ in Parameter 5:
   (a) an affine neighborhood size function $\mathsf{c}_{\mathbf{N}}\colon \mathbb{N} \to \mathbb{N}$,
   (b) a time function $\mathsf{t}_{\mathbf{N}}\colon \mathbb{N} \to \mathbb{N}$, and
   (c) a space function $\mathsf{s}_{\mathbf{N}}\colon \mathbb{N} \to \mathbb{N}$;
5. a family $\mathbf{N} = \{N_t\}_{t\in\mathbb{N}}$ such that:
   (a) $N_t = (\mathsf{aff}_{t,i}\colon \mathbb{F}_t \to \mathbb{F}_t)_{i=1}^{\mathsf{c}_{\mathbf{N}}(t)}$ is a sequence of $\mathsf{c}_{\mathbf{N}}(t)$ affine functions over $\mathbb{F}_t$,
   (b) each affine function $\mathsf{aff}_{t,i}$ is specified (in the straightforward way) by two elements $a_{t,i}(\mathsf{x})$ and $b_{t,i}(\mathsf{x})$ in $\mathbb{F}_t$, and
   (c) there exists a $\mathsf{t}_{\mathbf{N}}$-time $\mathsf{s}_{\mathbf{N}}$-space algorithm $\textsc{Find}\mathbf{N}$ such that $\textsc{Find}\mathbf{N}(1^t, i) = (a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x}))$ for all $t \in \mathbb{N}$ and $i \in \{1, \dots, \mathsf{c}_{\mathbf{N}}(t)\}$;
6. two functions associated with the family $\mathbf{D}$ in Parameter 7:
   (a) a time function $\mathsf{t}_{\mathbf{D}}\colon \mathbb{N} \to \mathbb{N}$, and
   (b) a space function $\mathsf{s}_{\mathbf{D}}\colon \mathbb{N} \to \mathbb{N}$;
7. a family $\mathbf{D} = \big\{(d_0^t, d_1^t, \dots, d_{\mathsf{c}_{\mathbf{N}}(t)}^t)\big\}_{t\in\mathbb{N}}$ such that:
   (a) $d_i^t \in \mathbb{N}$ for $i = 0, 1, \dots, \mathsf{c}_{\mathbf{N}}(t)$,
   (b) $d_0^t + (2^{\mathsf{m}_{\mathbf{H}}(t)} - 1)\sum_{i=1}^{\mathsf{c}_{\mathbf{N}}(t)} d_i^t \le 2^{f(t)-2}$, and
   (c) there exists a $\mathsf{t}_{\mathbf{D}}$-time $\mathsf{s}_{\mathbf{D}}$-space algorithm $\textsc{Comp}\mathbf{D}$ such that $d_i^t = \textsc{Comp}\mathbf{D}(1^t, i)$ for every $t \in \mathbb{N}$ and $i \in \{0, 1, \dots, \mathsf{c}_{\mathbf{N}}(t)\}$;
8. four functions associated with the family $\mathbf{P}$ in Parameter 9:
   (a) a time function $\mathsf{t}_{\mathbf{P}}\colon \mathbb{N} \to \mathbb{N}$,
   (b) a space function $\mathsf{s}_{\mathbf{P}}\colon \mathbb{N} \to \mathbb{N}$,
   (c) a size function $\mathsf{S}_{\mathbf{P}}\colon \mathbb{N} \to \mathbb{N}$, and
   (d) a degree function $\mathsf{D}_{\mathbf{P}}\colon \mathbb{N} \to \mathbb{N}$;

9. a family $\mathbf{P} = \{P_t\}_{t\in\mathbb{N}}$ such that:
   (a) $P_t \colon \mathbb{F}_t^{1+\mathsf{c_N}(t)} \to \mathbb{F}_t$ is a constraint polynomial,
   (b) there exists a $\mathsf{t_P}$-time $\mathsf{s_P}$-space algorithm $\textsc{Find}\mathbf{P}$ such that $[P_t]^{\mathsf{A}} = \textsc{Find}\mathbf{P}(1^t)$ is a $\mathsf{S_P}$-size $\mathsf{D_P}$-degree $\mathbb{F}_t$-algebraic circuit computing P_t for every $t \in \mathbb{N}$, and
   (c) $\mathsf{D_P}(t)[i+1] \le d_i^t$ for $i = 0, 1, \ldots, \mathsf{c_N}(t)$;

10. two functions associated with the family $\mathbf{S}$ in Parameter 11:
   (a) a time function $\mathsf{t_S} \colon \mathbb{N} \to \mathbb{N}$, and
   (b) a space function $\mathsf{s_S} \colon \mathbb{N} \to \mathbb{N}$;

11. a family $\mathbf{S} = \{S_t\}_{t\in\mathbb{N}}$ such that:
   (a) S_t is a subset of H_t, and
   (b) there exists a $\mathsf{t_S}$-time $\mathsf{s_S}$-space algorithm $\textsc{Find}\mathbf{S}$ such that $\alpha_i(\mathsf{x}) := \textsc{Find}\mathbf{S}(1^t, i)$ is the i-th element in S_t (under some canonical ordering of S_t) for every $t \in \mathbb{N}$ and $i \in \{1, \ldots, |S_t|\}$.

The language $\textsc{Succinct}$ACSP, with respect to a choice $\mathsf{par}_{\text{sACSP}}$ of the above parameters, consists of instances $(x, 1^t)$, where x is a binary string and t is an integer with $|x| \le 2^t$, such that there exists an assignment polynomial $A \colon \mathbb{F}_t \to \mathbb{F}_t$ of degree less than $2^{\mathsf{m_H}(t)}$ for which the following two conditions hold:

(i) *Satisfiability of constraints.* For every element $\alpha(\mathsf{x}) \in H_t$,

$$P_t\Big(\alpha(\mathsf{x}), \big(A \circ \mathsf{aff}_{t,1}\big)(\alpha(\mathsf{x})), \ldots, \big(A \circ \mathsf{aff}_{t,\mathsf{c_N}(t)}\big)(\alpha(\mathsf{x}))\Big) = 0_{\mathbb{F}_t} \ .$$

   If so, we say that the assignment polynomial A *satisfies the constraint polynomial* P_t.

(ii) *Consistency with the instance.* For every index $i \in \{1, \ldots, |S_t|\}$, letting $\alpha_i(\mathsf{x})$ be the i-th element in S_t,

$$x = \mathrm{bit}\big(A(\alpha_1(\mathsf{x}))\big) \cdots \mathrm{bit}\big(A(\alpha_{|x|}(\mathsf{x}))\big) \ ,$$

   where $\mathrm{bit} \colon \mathbb{F}_t \to \{0, 1, \bot\}$ maps $0_{\mathbb{F}_t}$ to 0, $1_{\mathbb{F}_t}$ to 1, and anything else to $\bot$. If so, we say that the assignment polynomial A *is consistent* with the instance $(x, 1^t)$.

As with the definition for succinct graph-coloring problems (see Definition 8.1), the above definition for (univariate) succinct algebraic constraint satisfaction problems is quite a mouthful; again, this is because the language itself encodes requirements ensuring that "large objects" (in this case large subsets of a large finite field, high-degree polynomials, and so on) can be computed using very few resources by using succinct and functional representations of such objects (for example, efficiently computable bases, efficiently constructible small arithmetic circuits, and so on). Ultimately, these requirements will enable the existence of an efficient PCP verifier for membership in the language. (See [BSCGT12] for more.)

Intuitive discussion. At high level, a choice of parameters $\mathsf{par}_{\text{sACSP}}$ for $\textsc{Succinct}$ACSP identifies a collection of infinite families of objects (one object for each $t \in \mathbb{N}$). When a specific instance $(x, 1^t)$ is considered for membership in $\textsc{Succinct}$ACSP($\mathsf{par}_{\text{sACSP}}$), the t-th element from each of these families in the collection is used to determine membership of the instance; candidate witnesses are low-degree polynomials over the field $\mathbb{F}_t$. Despite the long definition, these objects interact in natural ways, so we now go over the parameters from Definition 10.1 in less formal terms, explaining some of the intuition behind the design of the definition.

- **Parameter 1.** The parameter f governs the "growth rate" of the size of the finite fields in the family $\{\mathbb{F}_t\}_{t\in\mathbb{N}}$; reductions from different languages to $\textsc{Succinct}$ACSP may yield different choices of f, and, roughly, the slower-growing the function f is the more efficient is the reduction.

- **Parameter 2 and Parameter 3.** The parameter $\mathbf{H} = \{H_t\}_{t\in\mathbb{N}}$ is a family of linear sets, where each H_t is contained in the corresponding field $\mathbb{F}_t$. A "succinct" representation of this family is provided by the algorithm $\textsc{Find}\mathbf{H}$, which, on input 1^t, generates a basis for H_t. (Moreover, $\textsc{Find}\mathbf{H}$ runs within the specified time and space complexities.) Intuitively, the subset H_t is where "interesting things will happen", and the only reason for H_t not being equal to the whole field $\mathbb{F}_t$ is that we need some "room" for some technical conditions to go through (specifically, those spelled out in condition (b) of Parameter 7). (For example, to later allow unique decoding of Reed-Solomon codes for certain distance ranges; see [BSCGT12] for more details.)

- **Parameter 4 and Parameter 5.** The parameter $\mathbf{N} = \{N_t\}_{t\in\mathbb{N}}$ is a family of collections of affine functions, where each collection N_t induces an "affine" neighborhood of size $\mathsf{c}_{\mathbf{N}}(t)$ for each element in the finite field $\mathbb{F}_t$. Later, a certain condition will impose a single "local" constraint on the values of (candidate witness) low-degree polynomials at every such affine neighborhood that can be found in H_t. Again, the (description of) affine functions that define each affine neighborhood can be computed via an algorithm FindN (which runs with the prescribed time and space complexities). The restriction to affine neighborhoods (as opposed to, say, "degree-2" neighborhoods) is technical; again, see [BSCGT12] for more details.

- **Parameter 6 and Parameter 7.** The parameter $\mathbf{D}$ is a family of vectors, where each vector in the family coordinate-wise upper bounds the vector of degrees of the variables of a constraint polynomial considered in the next two parameters. Once again, in order to make sure that these vectors can be found efficiently (as, in principle, they might not), the definition prescribes the existence of an algorithm CompD that can compute them (within the specified time and space complexities).

- **Parameter 8 and Parameter 9.** The parameter $\mathbf{P} = \{P_t\}_{t\in\mathbb{N}}$ is a family of constraint polynomials, where each polynomial determines which values of low-degree polynomials are valid for a given neighborhood (specifically, those values that make it zero). Each polynomial P_t itself may not have low-degree (and usually it will not), nor may be sparse, though it will be ensured that it can be computed via a small arithmetic circuit that can be generated efficiently, via the algorithm FindP.

At high level, the parameters described until now could, say, encode a variety of constraints that ensure "correct computation" (say, of a Turing machine, or a random-access machine), but so far did not encode anything about whether this correct computation had anything to do with the instance $(x, 1^t)$ under consideration. The remaining parameters relate the computation to the instance at hand.

- **Parameter 10 and Parameter 11.** The parameter $\mathbf{S} = \{S_t\}_{t\in\mathbb{N}}$ is a family of (not-necessarily linear)[10] subsets, each contained in the corresponding (linear) set H_t. Each subset S_t identifies on which "part" of H_t a (candidate witness) low-degree polynomial must contain information related to the instance $(x, 1^t)$; specifically, when such information is concatenated, one should obtain x. As usual, we need an algorithm FindS to ensure that accessing elements of S_t can be done appropriately efficiently.

In summary, the "satisfiability of constraints" condition of the definition should be interpreted as saying that some computation (the particulars of which are captured by the specific choices of parameters) was performed correctly, while the other condition, "consistency with the instance", should be interpreted as saying that this computation was performed on the instance at hand.

Flexibility of Definition 10.1. While for many theoretical purposes (e.g., proving inapproximability results) it suffices to construct (low-query, low-randomness) PCPs for certain NP-complete problems, when it comes to making PCPs useful in practice (by enabling computationally weak parties to verify long computations, for example), care must be exercised in choosing the NP-complete problems for which to construct PCPs.

To begin with, for there to be any hope of an efficient PCP verifier, we should only consider NP-complete problems that are "succinct and uniform" in nature, where (as already mentioned before) a long computation or large set of constraints can be compactly specified by, for example, the code of an algorithm, the description of a Turing machine, and so on. By considering such problems, we can hope that a PCP verifier can be constructed that will not only "save on" queries and randomness but will also do so on time and space complexity. (Indeed, if the input instance to the PCP verifier were as long as the computation, there is no point in probabilistically checking it with few queries and random bits.)

Moreover, the PCP verifier will likely be responsible for reducing from other (natural, and application-dependent) uniform NP-complete problems to the uniform NP-complete problem for which a PCP is constructed (because the NP-complete problems for which we are able to construct PCPs tend to be highly "structured" and thus "unnatural").

Therefore, ideally, we would like to identify a *very flexible class of succinct and uniform NP-complete problems for which we are able to construct, once and for all, a practical PCP, and at the same time are able to exhibit very "direct" reductions from a number of natural succinct and uniform problems to problems in this class.*

[10]Though, as discussed in [BSCGT12], if one is interested in constructing PCPs of Proximity, and not merely PCPs, for SuccinctACSP with efficient verifiers, then S_t should also be a *linear* subset.

We believe that (the class of problems) SuccinctACSP does exhibit these ideal properties. Certainly of a "succinct and uniform" nature, the definition is deliberately very general, leaving a lot of freedom for where the parameters might come from. The few conditions that we did impose (such as requiring that the subset H_t be linear) seem to us almost necessary, as they allow certain algebraic operations to be carried out much more efficiently than over other "non-algebraically nice" sets. Thus, SuccinctACSP provides a sufficiently general "meeting point" for which technologies can be independently developed "above" it (in terms of reductions from natural succinct and uniform problems) or "below" it (in terms of PCP technologies).

Finally, the "algebraic flavor" of the definition is a feature that will be easier to exploit in practice when attempting time and space optimizations. (Indeed, considering extension fields of $\mathbb{F}_2$ is particularly appealing for this reason.)

Remark 10.2. How is a choice of parameters $\mathsf{par}_{\text{sACSP}}$ for SuccinctACSP (concisely) specified? All the "complexity functions" from Definition 10.1 (i.e., those specifying running times, sizes of arithmetic circuits, and so on) were chosen to be proper (see Definition 6.1), and thus each has an efficient algorithm that computes it (which, for simplicity, we denote with the same name as the function); moreover, every infinite family comes with an algorithm that computes the information about the family we are interested in. Thus, a choice of parameters $\mathsf{par}_{\text{sACSP}}$ can be specified as follows:

$$\mathsf{par}_{\text{sACSP}} = \begin{pmatrix} f, \\ (\mathsf{m}_{\mathbf{H}}, \mathsf{t}_{\mathbf{H}}, \mathsf{s}_{\mathbf{H}}, \textsc{Find}\mathbf{H}), \\ (\mathsf{c}_{\mathbf{N}}, \mathsf{t}_{\mathbf{N}}, \mathsf{s}_{\mathbf{N}}, \textsc{Find}\mathbf{N}), \\ (\mathsf{t}_{\mathbf{D}}, \mathsf{s}_{\mathbf{D}}, \textsc{Comp}\mathbf{D}), \\ (\mathsf{t}_{\mathbf{P}}, \mathsf{s}_{\mathbf{P}}, \mathsf{S}_{\mathbf{P}}, \mathsf{D}_{\mathbf{P}}, \textsc{Find}\mathbf{P}), \\ (\mathsf{t}_{\mathbf{S}}, \mathsf{s}_{\mathbf{S}}, \textsc{Find}\mathbf{S}), \end{pmatrix} .$$

Remark 10.3. Our design of Definition 10.1 was inspired by related definitions that have appeared in [BSS08] and [BSGH+05]. We briefly discuss here how our definition compares to the previous ones.

The definition of Ben-Sasson and Sudan [BSS08, Definition 3.6] also considers low-degree polynomials vanishing at every affine neighborhood of a certain subset of a finite field; however, their definition (as is clear for its compactness!) does *not* require succinctness (for example, the subset H need not be linear) because their paper does not attempt to construct PCP verifiers that are efficient in time and space. (Note that, in the non-succinct case, the "consistency with the instance" requirement disappears.)

The later paper of Ben-Sasson et al. [BSGH+05], which constructs efficient PCP verifiers, does indeed give a definition [BSGH+05, Definition 6.1] for a succinct problem about polynomials (similar to the previous one of Ben-Sasson and Sudan [BSS08, Definition 3.6]), but is specific for the parameters obtained when reducing from bounded halting problems on Turing machines.

We consider a generalization of [BSGH+05, Definition 6.1] that leaves unspecified degrees of freedom that we believe important for "supporting" a variety of reductions from other models of computation.

Remark 10.4. Ben-Sasson et al. [BSGH+05] also consider a family of problems about *multivariate* polynomials [BSGH+05, Definition 6.3]. We could have also considered a generalization to their definition, analogous to our Definition 10.1 that generalizes the univariate case [BSGH+05, Definition 6.1]. (See Remark 10.3.)

We choose to leave the investigation of such a definition (along with the possibility of finding even better reductions from natural problems "above" it, and better PCPs "below" it) for future work.

11 From Succinct GCPs to Succinct ACSPs

We construct Levin reductions from certain choices of parameters for SuccinctGCP to certain choices of parameters for SuccinctACSP. Both of these classes of problems are "succinct" constraint satisfaction problems, where a candidate instance $(x, 1^t)$ must satisfy a very large set of constraints in order to be in a language. As already discussed in Section 6.1, we seek efficient Levin reductions that implicitly convert between the very large sets of constraints prescribed by the two languages.

Given the "template" Definition 6.5, we give for convenience the following specialized definition for reducing from SuccinctGCP to SuccinctACSP:

Definition 11.1. *Fix a class of parameters $\mathcal{P}_{\mathrm{sGCP}}$ for* SuccinctGCP. *We say that a pair of polynomial-time-computable functions $(F_{\mathsf{p}}, F_{\mathsf{w}})$ is a* Levin reduction from SuccinctGCP to SuccinctACSP with respect to $\mathcal{P}_{\mathrm{sGCP}}$ *if the following three conditions are satisfied for every choice of parameters $\mathsf{par}_{\mathrm{sGCP}} \in \mathcal{P}_{\mathrm{sGCP}}$:*

1. *$F_{\mathsf{p}}(\mathsf{par}_{\mathrm{sGCP}})$ is a choice of parameters for* SuccinctACSP.
2. *For every instance $(x, 1^t) \in \{0,1\}^*$,*
$$(x, 1^t) \in \text{SuccinctGCP}(\mathsf{par}_{\mathrm{sGCP}}) \textit{ if and only if } (x, 1^t) \in \text{SuccinctACSP}\big(F_{\mathsf{p}}(\mathsf{par}_{\mathrm{sGCP}})\big).$$
3. *For every instance $(x, 1^t) \in \{0,1\}^*$,*
$$\textit{if the coloring } C \textit{ is a witness to } \text{“}(x, 1^t) \in \text{SuccinctGCP}(\mathsf{par}_{\mathrm{sGCP}})\text{”,}$$
$$\textit{then } F_{\mathsf{w}}(\mathsf{par}_{\mathrm{sGCP}}, 1^{2^t}, C) \textit{ is a witness to } \text{“}(x, 1^t) \in \text{SuccinctACSP}\big(F_{\mathsf{p}}(\mathsf{par}_{\mathrm{sGCP}})\big)\text{”.}$$

The two functions F_{p} and F_{w} are respectively called as the "parameter reduction" and the "witness reduction".

This section consists of two parts:

- in Section 11.1, we construct a Levin reduction from succinct GCPs on double extended De Bruijn graphs to succinct ACSPs (covering, in particular, the choices of parameters produced by the Levin reduction from random-access machine of Section 9.1); and
- the details of the Levin reduction from succinct GCPs on cyclic graphs to succinct ACSPs will appear soon in a future version of this paper; in the meantime, it is easy to see that we can embed a GCP problem on a cyclic graph and use the reduction described in Section 11.1, though we will "pick up" an extra log factor.

At high level, each of these reductions is conceptually divided into two tasks:

- **Graph Arithmetization.** We need to convert the graph $G = (V, E)$ on which the local constraints of the succinct GCP problem are defined into a certain subspace H of a sufficiently large field $\mathbb{F}$, by defining an "affine graph" (see Definition 11.2 below) on the field. As we will only be allowing ourselves affine edges (and the efficiency of the reduction will depend on how many different affine edges we use), we need to be careful about the choice of graph embedding, to ensure that the affine edge set is small, and yet can "express" all the possible edges in the original graph.

- **Constraint Arithmetization.** We need to convert the Boolean circuits encoding the local constraints of the succinct GCP problem into arithmetic circuits that will constrain the values of a candidate-witness low-degree polynomial A over the affine neighborhood of every element in the subspace H. Doing so requires a careful arithmetization of the Boolean circuits (to ensure the resulting arithmetic circuit is small and does not have too big of a degree), as well as the construction of "neighbor selector" arithmetic circuits that are able to tell which elements in the affine neighborhood of an element are "authentic", i.e., are edges induced by the original graph. (Not all affine edges will be so, unfortunately.)

Of course, many more details are needed in order actually specify the reductions; these details mostly have to do with keeping track of how expensive it is to convert, represent, and access the very large objects involved, together with enforcing instance consistency (which was not addressed in the high level description above).

We give here the definition of an affine graph used throughout this section:

Definition 11.2. *Let $\mathbb{F}$ be a field and $\mathcal{A}$ a set of affine functions over $\mathbb{F}$. The* affine graph $\mathrm{AFF}(\mathbb{F}, \mathcal{A})$ *is a directed graph with vertex set $\mathbb{F}$ where each element $\alpha(\mathsf{x}) \in \mathbb{F}$ has a directed edge to the element $\mathsf{aff}(\alpha(\mathsf{x}))$ for each affine function $\mathsf{aff} \in \mathcal{A}$.*

Remark 11.3. We believe that the Levin reductions discussed in this section, as well as the arithmetization results developed in Section E, will provide a valuable "toolkit" for arithmetizing a variety of structured graphs (including butterfly networks and Beneš networks) for the purposes of engineering other Levin reductions from succinct constraint satisfaction problems on such graphs. Having this flexibility could allow for more direct reductions for other succinct models of computation (e.g., arithmetic formulas, finite automata), and thus ultimately lead to more efficient PCPs "specialized" for such models of computation.

11.1 Arithmetizing Double De Bruijn Graphs

We give a Levin reduction (in the sense of Definition 11.1) from succinct graph-coloring problems on *double extended De Bruijn graphs* to succinct algebraic constraints satisfaction problems. In particular, this Levin reduction will work for the parameters output by the Levin reduction from Section 9.1. See Theorem 11.16 for the formal statement of the result proved in this section.

For convenience, we provide again the definition of a double extended De Bruijn graph (which we had introduced in Definition 9.9). Recall that a double extended De Bruijn graph is simply two extended De Bruijn graphs "side by side", connected with bi-directional edges at corresponding vertices.

Definition 11.4. *Let κ and L be two positive integers. The (κ, L)* **double extended De Bruijn graph***, denoted $\mathrm{DDB}(\kappa, L)$, is a 3-regular directed graph consisting of the Cartesian graph product of the directed two-cycle and $\mathrm{DB}(\kappa, L)$. In other words, the vertex set V of $\mathrm{DDB}(\kappa, L)$ consists of vertices $v = (b, i, w)$, where $b \in \{0, 1\}$, $i \in \{0, \ldots, L-1\}$, and $w \in \{0, 1\}^\kappa$, and the edge set E is induced by the following three neighbor functions:*

- $\Gamma_1\big((b, i, w)\big) = (b, i+1 \bmod L, \mathrm{sr}(w))$*, i.e., one kind of De Bruijn edges;*
- $\Gamma_2\big((b, i, w)\big) = (b, i+1 \bmod L, \mathrm{sr}(w) \oplus e_1)$*, i.e., the other kind of De Bruijn edges; and*
- $\Gamma_3\big((b, i, w)\big) = (1 \oplus b, i, w)$*, i.e., edges between corresponding vertices of the two De Bruijn graphs.*

Remark 11.5. One could consider more general graph products of extended De Bruijn graphs, e.g., the Cartesian graph product with directed cycles larger than 2. These more complex structured graphs may very well be useful in Levin reductions from other models of computation. See Remark 11.13 for a short discussion of how our arithmetization techniques extend to these more general products.

Remark 11.6. Our arithmetization techniques and conversion of parameters also yield a Levin reduction from succinct graph coloring problems on *single* extended De Bruijn graphs to succinct algebraic constraint satisfaction problems. (A non-uniform version of this reduction was considered by [BSS08].) We happen not to use single extended De Bruijn graphs in this work, as a direct reduction from random-access machines seems to require the use of two routing networks; nonetheless, single extended De Bruijn graphs are quite expressive, and may indeed be useful in other Levin reductions.

Remark 11.7. We note that there are other "arithmetization" paths that could be used, say, starting from SuccinctGCPs on double extended De Bruijn graphs. For example, one could consider *multivariate* variants of succinct algebraic constraint satisfaction problems. Alternatively, arithmetizing separately different parts of the graph (e.g., each extended De Bruijn graph in the double extended De Bruijn graph), to obtain a univariate succinct algebraic constraint satisfaction problem with multiple polynomials as witnesses, and consistency checks between them. These other arithmetization paths would require a new class of succinct algebraic constraint satisfaction problems, thus requiring a new definition similar in spirit to Definition 10.1 (and ultimately a new construction of PCPs for it, which may or may not follow easily from the PCPs of [BSCGT12] for Definition 10.1).

Remark 11.8. The main theorem of this section, Theorem 11.16, can be thought of as a generalization of [BSGH+05, Theorem 6.2] (and our proof easily implies an explicit proof to [BSGH+05, Theorem 6.2], which was only stated without proof); the arithmetization needed in the proof is inspired by [BSS08, proof of Theorem 3.7], but requires a lot more care and involves more details.

This section is organized as follows: we first provide some algebraic lemmas establishing appropriate embeddings of double extended De Bruijn graphs into affine graphs, along with other properties (Section 11.1.1); then give the conversion of parameters (Section 11.1.2), and finally prove that the conversion of parameters yields a Levin reduction (Section 11.1.3).

11.1.1 An embedding and some lemmas for double extended De Bruijn graphs

We first prove several lemmas about (*single*) extended De Bruijn graphs, and then build on these to obtain analogous results for *double* extended De Bruijn graphs.

The single De Bruijn case. First, we show how to "arithmetize" (single) extended De Bruijn graphs by embedding them into appropriate affine graphs. More concretely, given a graph $\mathrm{DB}(\kappa, 2^\ell - 1) = (V, E)$ (where we take for convenience the number of columns to be a power of 2), we need to come up with a function $\widetilde{\Phi}_{f,\kappa,\ell}$ that maps vertices in V to field elements of $\mathbb{F}_{2^f}$ (for a sufficiently large f) and edges in E to edges of an affine graph over the elements of $\mathbb{F}_{2^f}$ induced by a small set of affine functions $\widetilde{\mathcal{A}}_{f,\kappa,\ell}$; we will call such a function a *graph embedding* (and the reason that it is not a graph isomorphism is that generally the affine graph will contain many more vertices and edges than those induced by the mapping from $\mathrm{DB}(\kappa, 2^\ell - 1)$).

Recall from Definition 6.6 that a vertex $v \in V$ is labeled as (i, w), where $i \in \{0, \dots, 2^\ell - 2\}$ and $w \in \{0, 1\}^\kappa$. To encode the label as a field element in $\mathbb{F}_{2^f}$: we will encode the string w as the high order coefficients of (the polynomial representaiton of) a field element; the low-order coefficients of (the polynomial representaiton of) the field element will be reserved for the encoding of i, which is achieved by creating an "artificial" cyclic group of size $2^\ell - 1$ (all of whose elements are polynomials of degree less than ℓ and thus can be stored as low-order coefficients) and letting i be mapped to the i-th element of this cyclic group. The encoding of the label (i, w) was designed to be "friendly" to affine functions, in the sense that all the images of vertices that are connected by an edge in E can be related by a small set of affine functions.

More precisely, we prove the following lemma:

Lemma 11.9 (Arithmetization of Extended De Bruijn Graphs)**.** *Fix three positive integers f, κ, and ℓ with $f \geq \ell + \kappa + 1$. Let I be the irreducible polynomial of degree f over $\mathbb{F}_2$ output by* FindIrrPoly(1^f)*, and let $\mathbb{F}_{2^f} := \mathbb{F}_2(\mathsf{x})$ where x is a root of I. Let $\Xi(x) \in \mathbb{F}_2[x]$ be the primitive polynomial of degree ℓ output by* FindPrimPoly(1^ℓ) *and let $\xi_0(x), \dots, \xi_{(2^\ell-2)}(x) \in \mathbb{F}_2[x]$ be the $2^\ell - 1$ distinct polynomials of degree less than ℓ implied by invoking Claim E.5 with $\Xi(x)$.*

Define the two functions $h_{f,\kappa,\ell} \colon \{0, \dots, 2^\ell - 2\} \to \mathbb{F}_{2^f}$ and $g_{f,\kappa,\ell} \colon \{0,1\}^\kappa \to \mathbb{F}_{2^f}$ by

$$h_{f,\kappa,\ell}(i) := \xi_i(\mathsf{x}) \quad \text{and} \quad g_{f,\kappa,\ell}(w) := \mathsf{x}^{\ell-1} \cdot \left(\sum_{j=1}^{\kappa} w_j \mathsf{x}^j \right) ,$$

and define the function $\widetilde{\Phi}_{f,\kappa,\ell} \colon \{0, \dots, 2^\ell - 2\} \times \{0,1\}^\kappa \to \mathbb{F}_{2^f}$ by

$$\widetilde{\Phi}_{f,\kappa,\ell}(i, w) := h_{f,\kappa,\ell}(i) + g_{f,\kappa,\ell}(w) .$$

Then, $\widetilde{\Phi}_{f,\kappa,\ell}$ is an injective isomorphism from the De Bruijn graph $\mathrm{DB}(\kappa, 2^\ell - 1) = (V, E)$ into the affine graph $\mathrm{AFF}(\mathbb{F}_{2^f}, \widetilde{\mathcal{A}}_{f,\kappa,\ell})$, where

$$\widetilde{\mathcal{A}}_{f,\kappa,\ell} = \Big\{ \mathsf{aff}_\sigma(z) = \mathsf{x} \cdot z + \sigma_1 \Xi(\mathsf{x}) + \sigma_2 \mathsf{x}^\ell + \sigma_3 \mathsf{x}^{\ell+\kappa} \in \mathbb{F}_{2^f}[z] \Big\}_{\sigma \in \{0,1\}^3} .$$

Note that $|\widetilde{\mathcal{A}}_{f,\kappa,\ell}| = 8$.

Proof. Observe that $\deg(h_{f,\kappa,\ell}(i)) = \deg(\xi_i(\mathsf{x})) < \ell$ for every $i \in \{0, \dots, 2^\ell - 2\}$ and $\deg(g_{f,\kappa,\ell}(w)) = \deg(\mathsf{x}^{\ell-1} \cdot (\sum_{j=1}^{\kappa} w_j \mathsf{x}^j)) \geq \ell$ for every not-all-zeros κ-bit string w. Hence, the function $\widetilde{\Phi}_{f,\kappa,\ell}$ is injective if both $h_{f,\kappa,\ell}$ and $g_{f,\kappa,\ell}$ are injective; injectivity of the second map is clear, and injectivity of the first map follows from Claim E.4 (which tells us that $i \neq i'$ implies that $h_{f,\kappa,\ell}(i) = \xi_i(\mathsf{x}) \neq \xi_{i'}(\mathsf{x}) = h_{f,\kappa,\ell}(i')$). Next, to prove that $\widetilde{\Phi}_{f,\kappa,\ell}$ is a graph isomorphism, we need to show that if (v, v') is an edge in $\mathrm{DB}(\kappa, 2^\ell - 1)$ then $(\widetilde{\Phi}_{f,\kappa,\ell}(v), \widetilde{\Phi}_{f,\kappa,\ell}(v'))$ is an edge in $\mathrm{AFF}(\mathbb{F}_{2^f}, \widetilde{\mathcal{A}}_{f,\kappa,\ell})$.

We first derive some relations. Recalling the definition of $h_{f,\kappa,\ell}$, from Corollary E.5 we deduce that

$$h_{f,\kappa,\ell}(i+1 \bmod (2^\ell-1)) = \xi_{(i+1 \bmod (2^\ell-1))}(\mathsf{x}) = \mathsf{x}\cdot\xi_i(\mathsf{x}) + Q(\mathsf{x})\cdot\Xi(\mathsf{x}) = \mathsf{x}\cdot h_{f,\kappa,\ell}(i) + Q(\mathsf{x})\cdot\Xi(\mathsf{x}) \ ,$$

where $Q(\mathsf{x})$ is the polynomial quotient (of degree less than 1, i.e., $Q(\mathsf{x}) = 0$ or $Q(\mathsf{x}) = 1$) when dividing $\mathsf{x}\cdot h_{f,\kappa,\ell}(i)$ by $\Xi(\mathsf{x})$. (Note that, again as above, in the computation, we have used the fact that the field $\mathbb{F}_{2^f}$ is "large enough".) Therefore,

$$h_{f,\kappa,\ell}(i+1 \bmod (2^\ell-1)) = \begin{cases} \mathsf{x}\cdot h_{f,\kappa,\ell}(i) = \mathsf{aff}_{000}\big(h_{f,\kappa,\ell}(i)\big) & \text{if } \deg\big(h_{f,\kappa,\ell}(i)\big) < \ell-1 \\ \mathsf{x}\cdot h_{f,\kappa,\ell}(i) + \Xi(\mathsf{x}) = \mathsf{aff}_{100}\big(h_{f,\kappa,\ell}(i)\big) & \text{if } \deg\big(h_{f,\kappa,\ell}(i)\big) = \ell-1 \end{cases} .$$

Similarly, recalling the definition of $g_{f,\kappa,\ell}$ and $\widetilde{\mathcal{A}}_{f,\kappa,\ell}$, we see that

$$g_{f,\kappa,\ell}\big(\mathrm{sr}(w)\big) = \begin{cases} \mathsf{x}\cdot g_{f,\kappa,\ell}(w) = \mathsf{aff}_{000}\big(g_{f,\kappa,\ell}(w)\big) & \text{if } w_\kappa = 0 \\ \mathsf{x}\cdot g_{f,\kappa,\ell}(w) + \mathsf{x}^\ell + \mathsf{x}^{\ell+\kappa} = \mathsf{aff}_{011}\big(g_{f,\kappa,\ell}(w)\big) & \text{if } w_\kappa = 1 \end{cases}$$

and

$$g_{f,\kappa,\ell}\big(\mathrm{sr}(w)\oplus e_1\big) = \begin{cases} \mathsf{x}\cdot g_{f,\kappa,\ell}(w) + \mathsf{x}^\ell = \mathsf{aff}_{010}\big(g_{f,\kappa,\ell}(w)\big) & \text{if } w_\kappa = 0 \\ \mathsf{x}\cdot g_{f,\kappa,\ell}(w) + \mathsf{x}^{\ell+\kappa} = \mathsf{aff}_{001}\big(g_{f,\kappa,\ell}(w)\big) & \text{if } w_\kappa = 1 \end{cases} .$$

(Note that, in the above two equations, we have used the fact that the field $\mathbb{F}_{2^f}$ is "large enough". Specifically, by the choice $f \geq \ell+\kappa+1$, since all the polynomials in the computation have degree less than $\ell+\kappa+1$, we do not have to worry about reducing modulo $I(\mathsf{x})$.)

Thus, parsing v as (i,w), if (v,v') is an edge in $\mathrm{DB}(\kappa, 2^\ell-1)$, then either $v' = \big((i+1 \bmod (2^\ell-1)), \mathrm{sr}(w)\big)$ or $v' = \big((i+1 \bmod (2^\ell-1)), \mathrm{sr}(w)\oplus e_1\big)$, so that either $\widetilde{\Phi}_{f,\kappa,\ell}(v') = h_{f,\kappa,\ell}\big((i+1 \bmod (2^\ell-1))\big) + g_{f,\kappa,\ell}\big(\mathrm{sr}(w)\big)$ or $\widetilde{\Phi}_{f,\kappa,\ell}(v') = h_{f,\kappa,\ell}\big((i+1 \bmod (2^\ell-1))\big) + g_{f,\kappa,\ell}\big(\mathrm{sr}(w)\oplus e_1\big)$. Either way, it is possible to write $\widetilde{\Phi}_{f,\kappa,\ell}(v') = \mathsf{aff}_\sigma(v)$ for some $\sigma \in \{0,1\}^3$ by referring to the computations above in order to deduce σ_h for the $h_{f,\kappa,\ell}$ term and σ_g for the $g_{f,\kappa,\ell}$ term, and setting $\sigma = \sigma_h \oplus \sigma_g$. □

We now discuss the computational properties of the (single) extended De Bruijn graph embedding $\widetilde{\Phi}_{f,\kappa,\ell}$ defined in Lemma 11.9. First, we note that it is efficiently computable, both via an algorithm and a certain polynomial:

Lemma 11.10. *Define*

$$\mathsf{t}_{\widetilde{\Phi}}(f,\kappa,\ell) := \mathsf{t}_{\mathrm{PRIM}}(\ell) + \ell^2\log\ell + f \ ,$$
$$\mathsf{s}_{\widetilde{\Phi}}(f,\kappa,\ell) := \mathsf{s}_{\mathrm{PRIM}}(\ell) + f \ .$$

There exists a $\mathsf{t}_{\widetilde{\Phi}}$-time $\mathsf{s}_{\widetilde{\Phi}}$-space algorithm $\textsc{Comp}\widetilde{\boldsymbol{\Phi}}$ such that $\widetilde{\Phi}_{f,\kappa,\ell}\big((i,w)\big) = \textsc{Comp}\widetilde{\boldsymbol{\Phi}}\big(1^f,1^\kappa,1^\ell,(i,w)\big)$, for every three positive integers f, κ, and ℓ as in Lemma 11.9, and for every $i \in \{0,\dots,2^\ell-2\}$ and $w \in \{0,1\}^\kappa$.

Moreover, there exists a multilinear polynomial $P^{f,\ell,\kappa}_{\widetilde{\Phi}}\colon \mathbb{F}^{\ell+\kappa}_{2^f} \to \mathbb{F}_{2^f}$ such that

$$P^{f,\ell,\kappa}_{\widetilde{\Phi}}(i,w) = \textsc{Comp}\widetilde{\boldsymbol{\Phi}}\big(1^f,1^\kappa,1^\ell,(i,w)\big)$$

for every $i \in \{0,\dots,2^\ell-2\}$ and $w \in \{0,1\}^\kappa$; moreover, this polynomial can be found and evaluated in time $O(w+2^\ell)$.

Proof. The injective graph isomorphism $\widetilde{\Phi}_{f,\kappa,\ell}\colon \{0,\dots,2^\ell-1\}\times\{0,1\}^\kappa \to \mathbb{F}_{2^f}$ can be efficiently computed by the following algorithm:

$\textsc{Comp}\widetilde{\boldsymbol{\Phi}}\big(1^f,1^\kappa,1^\ell,(i,w)\big) \equiv$

1. Compute $\Xi := \textsc{FindPrimPoly}\big(1^\ell\big)$.
2. Compute $\alpha(\mathsf{x}) := \mathsf{x}^i \bmod \Xi(\mathsf{x})$.
3. Compute $\beta(\mathsf{x}) := \sum_{j=1}^{\kappa} w_j \mathsf{x}^{j+\ell-1}$.
4. Output $\alpha(\mathsf{x}) + \beta(\mathsf{x})$.

Note that $\textsc{Comp}\widetilde{\mathbf{\Phi}}$ does not have to compute the degree-f irreducible polynomial I over $\mathbb{F}_2$ that defines the field $\mathbb{F}_{2^f}$, because both $\alpha(\mathsf{x})$ and $\beta(\mathsf{x})$, as well as $\alpha(\mathsf{x})+\beta(\mathsf{x})$, all have degree less than f. Moreover, $\textsc{Comp}\widetilde{\mathbf{\Phi}}$ has the claimed complexity parameters.

The polynomial $P^{f,\ell,\kappa}_{\widetilde{\mathbf{\Phi}}}$ is the straightforward conversion of $\textsc{Comp}\widetilde{\mathbf{\Phi}}$ into a circuit. Specifically, we can take

$$P^{f,\ell,\kappa}_{\widetilde{\mathbf{\Phi}}}(x_1,\ldots,x_{\ell+\kappa}) := \left(\sum_{i\in\{0,1\}^\ell} h_{f,\kappa,\ell}(i)\prod_{j=1}^{\ell}(x_j-(1-i_j))\right)+\mathsf{x}^{\ell-1}\cdot\left(\sum_{j=1}^{\kappa} x_{\ell+j}\mathsf{x}^j\right) \quad .$$

□

The double De Bruijn case. Next, we build on the "single extended De Bruijn case", and show how to arithmetize double extended De Bruijn graphs (according to Definition 11.4). We will not have to work hard for this arithmetization, as most of the required arithmetization work went in to understanding the single extended De Bruijn graph case. More concretely, the basic intuition for arithmetizing a double extended De Bruijn graph is quite straightforward: we simply arithmetize each of the extended De Bruijn graphs separately, and put them side by side in the same finite field, ensuring that we have additional affine edges in the resulting affine graph to connect between them; the operations of "putting side by side" intuitively consist of doubling the field size, and putting the two affine graphs in two halves of the finite field, thus obtaining an extra "bit" to toggle between the two graphs. (Technically, we actually will not have to double the field size, but it is helpful to think about it this way.)

More precisely, we prove the following lemma:

Lemma 11.11 (Arithmetization of Double Extended De Bruijn Graphs)**.** *Fix three positive integers f, κ, and ℓ with $f\geq\ell+\kappa+1$. Let I be the irreducible polynomial of degree f over $\mathbb{F}_2$ output by* $\textsc{FindIrrPoly}(1^f)$*, and let $\mathbb{F}_{2^f}:=\mathbb{F}_2(\mathsf{x})$ where x is a root of I. Invoke Lemma 11.9 with the three positive integers f, κ, and ℓ to obtain the function $\widetilde{\Phi}_{f,\kappa,\ell}$ and affine edges $\widetilde{\mathcal{A}}_{f,\kappa,\ell}$.*

Define the function $\Phi_{f,\kappa,\ell}\colon\{0,1\}\times\{0,\ldots,2^\ell-2\}\times\{0,1\}^\kappa\to\mathbb{F}_{2^f}$ by

$$\Phi_{f,\kappa,\ell}(b,i,w):=\widetilde{\Phi}_{f,\kappa,\ell}(i,w)+b\mathsf{x}^{\ell+\kappa} \quad .$$

Then, $\Phi_{f,\kappa,\ell}$ is an injective isomorphism from the double extended De Bruijn graph $\mathrm{DDB}(\kappa,2^\ell)=(V,E)$ into the affine graph $\mathrm{AFF}(\mathbb{F}_{2^f},\mathcal{A}_{f,\kappa,\ell})$, where

$$\mathcal{A}_{f,\kappa,\ell}=\{\mathsf{aff}_\times(z)=z+\mathsf{x}^{\kappa+\ell}\in\mathbb{F}_{2^f}[z]\}\cup\widetilde{\mathcal{A}}_{f,\kappa,\ell} \quad .$$

Note that $|\mathcal{A}_{f,\kappa,\ell}|=1+|\widetilde{\mathcal{A}}_{f,\kappa,\ell}|=1+8=9$.

Proof. The function $\Phi_{f,\kappa,\ell}$ is injective because $\widetilde{\Phi}_{f,\kappa,\ell}$ is injective and the degree of an element output by $\widetilde{\Phi}_{f,\kappa,\ell}$ is at most $\ell+\kappa-1$. To see why $\Phi_{f,\kappa,\ell}$ is a graph isomorphism we note that $\widetilde{\Phi}_{f,\kappa,\ell}$ is a graph isomorphism on the 0-th extended De Bruijn graph and $\mathsf{x}^{\ell+\kappa+1}+\widetilde{\Phi}_{f,\kappa,\ell}$ is a graph isomorphism on the extended 1-th De Bruijn graph; bi-directional edges between images of corresponding vertices $(0,i,w)$ and $(1,i,w)$ in the two De Bruijn graphs are indeed affine edges given by the affine function $\mathsf{aff}_\times$ in $\mathcal{A}_{f,\kappa,\ell}$ (where the "$\times$" symbol is intended to denote "crossing" between the two De Bruijn graphs). □

The double extended De Bruijn graph embedding $\Phi_{f,\kappa,\ell}$ from Lemma 11.11 is efficiently computable:

Lemma 11.12. *Define*

$$\mathsf{t}_{\mathbf{\Phi}}(f,\kappa,\ell):=\mathsf{t}_{\textsc{Prim}}(\ell)+\ell^2\log\ell+f \quad ,$$
$$\mathsf{s}_{\mathbf{\Phi}}(f,\kappa,\ell):=\mathsf{s}_{\textsc{Prim}}(\ell)+f \quad .$$

There exists a polynomial-time algorithm $\textsc{Comp}\mathbf{\Phi}$ such that $\Phi_{f,\kappa,\ell}\big((b,i,w)\big)=\textsc{Comp}\mathbf{\Phi}\big(1^f,1^\kappa,1^\ell,(b,i,w)\big)$, for every three positive integers f, κ, and ℓ as in Lemma 11.11, and for every $b\in\{0,1\}$, $i\in\{0,\ldots,2^\ell-2\}$, and $w\in\{0,1\}^\kappa$.

Moreover, there exists a multilinear polynomial $P^{f,\ell,\kappa}_{\mathbf{\Phi}}\colon\mathbb{F}^{1+\ell+\kappa}_{2^f}\to\mathbb{F}_{2^f}$ such that

$$P^{f,\ell,\kappa}_{\mathbf{\Phi}}(b,i,w)=\textsc{Comp}\mathbf{\Phi}\big(1^f,1^\kappa,1^\ell,(b,i,w)\big)$$

for every $b\in\{0,1\}$, $i\in\{0,\ldots,2^\ell-2\}$ and $w\in\{0,1\}^\kappa$.

Proof. The injective graph isomorphism $\Phi_{f,\kappa,\ell}\colon \{0,1\} \times \{0,\dots,2^\ell - 2\} \times \{0,1\}^\kappa \to \mathbb{F}_{2^f}$ can be efficiently computed by the following isomorphism:

$\text{Comp}\mathbf{\Phi}\big(1^f, 1^\kappa, 1^\ell, (b,i,w)\big) \equiv$

1. Compute $\alpha(\mathsf{x}) := \text{Comp}\widetilde{\mathbf{\Phi}}\big(1^f, 1^\kappa, 1^\ell, (i,w)\big)$. (Where $\text{Comp}\widetilde{\mathbf{\Phi}}$ comes from Lemma 11.10.)
2. If $b = 0$, output $\alpha(\mathsf{x})$; otherwise, output $\mathsf{x}^{\ell+\kappa} + \alpha(\mathsf{x})$.

The correctness and complexity guarantees of $\text{Comp}\mathbf{\Phi}$ immediately follow from those of $\text{Comp}\widetilde{\mathbf{\Phi}}$.

The polynomial $P_{\mathbf{\Phi}}^{f,\ell,\kappa}$ is the straightforward conversion of $\text{Comp}\mathbf{\Phi}$ into a circuit, as we can take:

$$P_{\mathbf{\Phi}}^{f,\ell,\kappa}(x_1,\dots,x_{1+\ell+\kappa}) := x_1\mathsf{x}^{\ell+\kappa} + P_{\widetilde{\mathbf{\Phi}}}^{f,\ell,\kappa}(x_2,\dots,x_{1+\ell+\kappa}) \ ,$$

where $P_{\widetilde{\mathbf{\Phi}}}^{f,\ell,\kappa}$ comes from Lemma 11.10. □

Remark 11.13. The arithmetization techniques in this section extend to handle a variety of products of more complex graphs (other than the directed two-cycle) with De Bruijn graphs. For example, to arithmetize the product of a (small!) clique and a De Bruijn graph, we can simply add "toggle" high-order bits in Lemma 11.11 to keep track of which De Bruijn graph we are in. As another example, to arithmetize the product of a (possibly large) cycle with a De Bruijn graph, instead of using toggle bits, we can add another "artificial" cyclic group to keep track of which De Bruijn graph we are in (the savings on the field size comes from the fact that now we only have to worry about very few structured edges). Arithmetizing such *multiple* extended De Bruijn graphs may be useful for Levin reductions from other problems.

Remark 11.14. Our Lemma 11.9 is a modified (and improved) version of [BSS08, Proposition 5.10], which also gives an embedding of a single extended De Bruijn graph into an affine graph with eight affine edges. Our modifications include removing a "hole" in the encoding (thus halving the required field size) as well as making explicit the asymptotic dependence of the embedding on the two parameters κ and ℓ (which are the logarithm of the height and the width of the graph, respectively). The explicit asymptotic dependence on these parameters allows us to make precise statements about the complexity of the embedding (see Lemma 11.10); these computational statements are needed to eventually imply the Levin reduction that we seek.

11.1.2 The conversion of parameters for double extended De Bruijn graphs

We now give a conversion of parameters for SuccinctGCP with double extended De Bruijn graphs (satisfying certain conditions) to parameters for SuccinctACSP.

At high level, the conversion will have to arithmetize the double extended De Bruijn graph (and for this part we have already proved several lemmas in Section 11.1.1) and to arithmetize the Boolean circuits representing the local coloring constraints on the graph; moreover, the parameter conversion has to keep track of how to represent and access large objects, as well as to arithmetize the requirement of input consistency.

We now proceed to the details of the parameter conversion, which are given in the following construction:

Construction 11.15. Fix four (proper) functions $\kappa, \ell, s_0, s_1 \colon \mathbb{N} \to \mathbb{N}$ and functions. Let

$$\Big((\alpha_{\mathbf{G}}, \mathbf{G}), (\mathsf{c}_{\mathbf{C}}, \mathbf{C}), (\mathsf{c}_{\mathbf{T}}, \mathbf{T}), (\mathsf{S}_{\mathbf{M}}, \mathsf{D}_{\mathbf{M}}, \mathsf{t}_{\mathbf{M}}, \mathsf{s}_{\mathbf{M}}, \mathbf{M}), (\mathsf{S}_{\mathbf{K}}, \mathsf{D}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathsf{s}_{\mathbf{K}}, \mathbf{K}), (\mathsf{t}_{\mathbf{W}}, \mathsf{s}_{\mathbf{W}}, \mathbf{W}), (\mathsf{t}_{\mathbf{F}}, \mathsf{s}_{\mathbf{F}}, \mathbf{F})\Big)$$

be a choice of parameters for SuccinctGCP, where, for every $t \in \mathbb{N}$,

- $\alpha_{\mathbf{G}}(t) = 3$;
- $G_t = \text{DDB}(\kappa(t), 2^{\ell(t)} - 1)$; and
- $F_t(x, c_1 \cdots c_{|x|})$ is the test $x \stackrel{?}{=} c'_1 \cdots c'_{|x|}$ where each c'_i is the substring of c_i from bit $s_0(t)$ to bit $s_1(t)$.

Construct a choice of parameters

$$\Big(f, (\mathsf{m}_{\mathbf{H}}, \mathsf{t}_{\mathbf{H}}, \mathsf{s}_{\mathbf{H}}, \mathbf{H}), (\mathsf{c}_{\mathbf{N}}, \mathsf{t}_{\mathbf{N}}, \mathsf{s}_{\mathbf{N}}, \mathbf{N}), (\mathsf{t}_{\mathbf{D}}, \mathsf{s}_{\mathbf{D}}, \mathbf{D}), (\mathsf{t}_{\mathbf{P}}, \mathsf{s}_{\mathbf{P}}, \mathsf{S}_{\mathbf{P}}, \mathsf{D}_{\mathbf{P}}, \mathbf{P}), (\mathsf{t}_{\mathbf{S}}, \mathsf{s}_{\mathbf{S}}, \mathbf{S}), \Big)$$

for SuccinctACSP as follows:

1. **Constructing Parameter 1.** Define the (proper) field size function $f\colon \mathbb{N} \to \mathbb{N}$ by

$$f(t) := \text{“smallest solution to Equation 10”} \ . \tag{6}$$

Recall that $\mathbb{F}_t = \mathbb{F}_2(\mathsf{x})$ and x is the root of I_t, which is the irreducible polynomial of degree $f(t)$ over $\mathbb{F}_2$ output by $\textsc{FindIrrPoly}(1^{f(t)})$.

Introducing useful families. Consider the following definitions:

- Define the family $\mathbf{X} = \left\{\Xi_t(x) \in \mathbb{F}_2[x]\right\}_{t\in\mathbb{N}}$ by $\Xi_t := \textsc{FindPrimPoly}\big(1^{\ell(t)}\big)$.
- Define the family $\mathbf{Y} = \{Y_t\}_{t\in\mathbb{N}}$ by

$$Y_t := \left\{\xi_{t,0}(x), \ldots, \xi_{t,(2^{\ell(t)}-2)}(x) \in \mathbb{F}_2[x]\right\} \ ,$$

where $\xi_{t,0}(x), \ldots, \xi_{t,(2^{\ell(t)}-2)}(x) \in \mathbb{F}_2[x]$ are the $2^{\ell(t)} - 1$ distinct polynomials of degree less than $\ell(t)$ obtained by invoking Claim E.5 with the primitive polynomial $\Xi_t \in \mathbf{X}$. Note that all of these polynomials have degree less than $\deg(I_t) = f(t)$, and therefore $\xi_{t,0}(\mathsf{x}), \ldots, \xi_{t,(2^{\ell(t)}-2)}(\mathsf{x})$ are in $\mathbb{F}_t$.

Moreover, since $\xi_{t,i}(\mathsf{x}) \equiv \mathsf{x}^i \bmod \Xi_t(\mathsf{x})$, each of the $\xi_{t,i}(\mathsf{x})$ can be efficiently computed from the input $(1^t, i)$ by the following algorithm: for $i \in \{0, \ldots, 2^{\ell(t)} - 2\}$,

$\textsc{Find}\mathbf{Y}(1^t, i) \equiv$
(a) Compute $\Xi_t := \textsc{FindPrimPoly}\big(1^{\ell(t)}\big)$.
(b) Compute $\xi_{t,i}(\mathsf{x}) := \mathsf{x}^i \bmod \Xi_t(\mathsf{x})$.
(c) Output $\xi_{t,i}(\mathsf{x})$.

- Define the dimension function $\mathsf{m}_{\hat{\mathbf{V}}}\colon \mathbb{N} \to \mathbb{N}$ by $\mathsf{m}_{\hat{\mathbf{V}}}(t) := \ell(t) + \kappa(t) + 1$.
- Define the family $\widehat{\mathbf{V}} = \big\{\widehat{V}_t\big\}_{t\in\mathbb{N}}$ by

$$\widehat{V}_t := \Phi_{f(t),\kappa(t),\ell(t)}(V_t) \ . \tag{7}$$

Recall that $\Phi_{f(t),\kappa(t),\ell(t)}$ is the injection from the double extended De Bruijn graph $\mathrm{DDB}(\kappa(t), 2^{\ell(t)}-1) = G_t = (V_t, E_t)$ into the affine graph $\mathrm{AFF}(\mathbb{F}_t, \mathcal{A}_{f(t),\kappa(t),\ell(t)})$, obtained by invoking Lemma 11.11 with the three positive integers $f(t)$, $\kappa(t)$ and $\ell(t)$. (Note that by our choice of parameters $f(t) \geq \ell(t) + \kappa(t) + 1$, as required by the invocation of the lemma.)

Observe that

$$\begin{aligned}\widehat{V}_t &= \left\{ \xi_{t,i}(\mathsf{x}) + \mathsf{x}^{\ell(t)-1} \cdot \left(\sum_{j=1}^{\kappa(t)} w_j \mathsf{x}^j\right) + b\mathsf{x}^{\ell(t)+\kappa(t)} \right\}_{\substack{i=0,\ldots,2^{\ell(t)}-1\,,\\ w\in\{0,1\}^{\kappa(t)}\,,\\ b\in\{0,1\}}} \\ &= \mathrm{span}(\mathsf{x}^j)_{j\in\{0,\ldots,\ell(t)+\kappa(t)\}} \ ,\end{aligned}$$

so that $\widehat{V}_t$ is a $\mathsf{m}_{\hat{\mathbf{V}}}(t)$-dimensional linear subset of $\mathbb{F}_t$ specified by a basis

$$\mathcal{B}_{\widehat{V}_t} := \left(\alpha_1^{\hat{\mathbf{V}}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\hat{\mathbf{V}}}(t)}^{\hat{\mathbf{V}}}(\mathsf{x})\right) = \left(1_{\mathbb{F}_t}, \mathsf{x}, \ldots, \mathsf{x}^{\mathsf{m}_{\hat{\mathbf{V}}}(t)-1}\right) \ .$$

Moreover, the basis $\mathcal{B}_{\widehat{V}_t}$ for $\widehat{V}_t$ can be efficiently computed from the input 1^t by the following algorithm:

$\textsc{Find}\widehat{\mathbf{V}}(1^t) \equiv$
(a) Compute the irreducible polynomial $I_t := \textsc{FindIrrPoly}\big(1^{f(t)}\big)$; it has root x.
(b) Compute the dimension function $\mathsf{m}_{\hat{\mathbf{V}}}(t) := \ell(t) + \kappa(t) + 1$.
(c) Compute the elements $1_{\mathbb{F}_t}, \mathsf{x}, \ldots, \mathsf{x}^{\mathsf{m}_{\hat{\mathbf{V}}}(t)-1}$ of $\mathbb{F}_2(\mathsf{x})$.
(d) Output the basis $\big(1_{\mathbb{F}_t}, \mathsf{x}, \ldots, \mathsf{x}^{\mathsf{m}_{\hat{\mathbf{V}}}(t)-1}\big)$.

- Define the dimension function $\mathsf{m}_{\mathbf{\Theta}}\colon \mathbb{N} \to \mathbb{N}$ by

$$\mathsf{m}_{\mathbf{\Theta}}(t) := \lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil + \lceil \log \mathsf{c}_{\mathbf{C}}(t) \rceil = \lceil \log(\ell(t) + \kappa(t) + 1) \rceil + \lceil \log \mathsf{c}_{\mathbf{C}}(t) \rceil \ . \tag{8}$$

- Define the family $\boldsymbol{\Theta} = \{\theta_{t,1}(\mathsf{x}), \ldots, \theta_{t,\mathsf{m}_{\boldsymbol{\Theta}}(t)}(\mathsf{x}) \in \mathbb{F}_t\}_{t\in\mathbb{N}}$ by $\theta_{t,i}(\mathsf{x}) := \mathsf{x}^{\mathsf{m}_{\hat{\mathbf{V}}}(t)-1+i}$. Each $\theta_t(\mathsf{x})$ can be efficiently computed from the input 1^t and index i by the following algorithm:

  $\textsc{Find}\boldsymbol{\Theta}(1^t, i) \equiv$
  (a) Compute the irreducible polynomial $I_t := \textsc{FindIrrPoly}(1^{f(t)})$; it has root x.
  (b) Compute the dimension function $\mathsf{m}_{\hat{\mathbf{V}}}(t) := \ell(t) + \kappa(t) + 1$.
  (c) Compute the element $\mathsf{x}^{\mathsf{m}_{\hat{\mathbf{V}}}(t)-1+i}$ of $\mathbb{F}_2(\mathsf{x})$.
  (d) Output $\mathsf{x}^{\mathsf{m}_{\hat{\mathbf{V}}}(t)-1+i}$.

  For convenience, for $k \in \{0, \ldots, 2^{\mathsf{m}_{\boldsymbol{\Theta}}(t)} - 1\}$, we define $\theta^{(k)}(\mathsf{x})$ to be the element $\sum_{\ell=1}^{\mathsf{m}_{\boldsymbol{\Theta}}(t)} \sigma_\ell \theta_{t,\ell}(\mathsf{x})$, where $\sigma_1 \cdots \sigma_{\mathsf{m}_{\boldsymbol{\Theta}}(t)}$ is the binary expansion of $k-1$. Note that $\theta^{(0)}(\mathsf{x}) = 0_{\mathbb{F}_t}$.

- Define the family $\widehat{\mathbf{V}'} = \{\widehat{V'}_t\}_{t\in\mathbb{N}}$ by

$$\widehat{V'}_t := \bigcup_{b_1,\ldots,b_{\mathsf{m}_{\boldsymbol{\Theta}}(t)} \in \{0,1\}} \left(\widehat{V}_t + b_1\theta_{t,1}(\mathsf{x}) + \cdots + b_{\mathsf{m}_{\boldsymbol{\Theta}}(t)}\theta_{t,\mathsf{m}_{\boldsymbol{\Theta}}(t)}(\mathsf{x})\right) \ . \tag{9}$$

  Observe that

$$\widehat{V'}_t = \mathrm{span}\Big(\alpha_1^{\hat{\mathbf{V}}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\hat{\mathbf{V}}}(t)}^{\hat{\mathbf{V}}}(\mathsf{x}), \theta_{t,1}(\mathsf{x}), \ldots, \theta_{t,\mathsf{m}_{\boldsymbol{\Theta}}(t)}(\mathsf{x})\Big) \ ,$$

  so that $\widehat{V'}_t$ is a $\mathsf{m}_{\widehat{\mathbf{V}'}}(t)$-dimensional linear subset of $\mathbb{F}_t$ specified by a basis

$$\mathcal{B}_{\widehat{V'}_t} := \Big(\alpha_1^{\widehat{\mathbf{V}'}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}'}}(t)}^{\widehat{\mathbf{V}'}}(\mathsf{x})\Big) = \Big(\alpha_1^{\hat{\mathbf{V}}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\hat{\mathbf{V}}}(t)}^{\hat{\mathbf{V}}}(\mathsf{x}), \theta_{t,1}(\mathsf{x}), \ldots, \theta_{t,\mathsf{m}_{\boldsymbol{\Theta}}(t)}(\mathsf{x}), \zeta_t(\mathsf{x})\Big) \ .$$

  Moreover, the basis $\mathcal{B}_{\widehat{V'}_t}$ for $\widehat{V'}_t$ can be efficiently computed from the input 1^t by the following algorithm:

  $\textsc{Find}\widehat{\mathbf{V}'}(1^t) \equiv$
  (a) Compute the irreducible polynomial $I_t := \textsc{FindIrrPoly}(1^{f(t)})$.
  (b) Compute the basis $\Big(\alpha_1^{\hat{\mathbf{V}}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\hat{\mathbf{V}}}(t)}^{\hat{\mathbf{V}}}(\mathsf{x})\Big) := \textsc{Find}\widehat{\mathbf{V}}(1^t)$.
  (c) For $i = 1, \ldots, \mathsf{m}_{\boldsymbol{\Theta}}(t)$, compute $\theta_{t,i}(\mathsf{x}) := \textsc{Find}\boldsymbol{\Theta}(1^t, i)$.
  (d) Output the basis $\Big(\alpha_1^{\hat{\mathbf{V}}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\hat{\mathbf{V}}}(t)}^{\hat{\mathbf{V}}}(\mathsf{x}), \theta_{t,1}(\mathsf{x}), \ldots, \theta_{t,\mathsf{m}_{\boldsymbol{\Theta}}(t)}(\mathsf{x})\Big)$.

- Define the family $\mathbf{Z} = \{\zeta_t(\mathsf{x}) \in \mathbb{F}_t\}_{t\in\mathbb{N}}$ by $\zeta_t(\mathsf{x}) := \mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}'}}(t)}$. Each $\zeta_t(\mathsf{x})$ can be efficiently computed from the input 1^t by the following algorithm:

  $\textsc{Find}\mathbf{Z}(1^t) \equiv$
  (a) Compute the irreducible polynomial $I_t := \textsc{FindIrrPoly}(1^{f(t)})$; it has root x.
  (b) Compute the dimension function $\mathsf{m}_{\widehat{\mathbf{V}'}}(t) := \ell(t) + \kappa(t) + 1 + \mathsf{m}_{\boldsymbol{\Theta}}(t)$.
  (c) Compute the element $\mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}'}}(t)}$ of $\mathbb{F}_2(\mathsf{x})$.
  (d) Output $\mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}'}}(t)}$.

2. **Constructing Parameter 2.** Define the following three (proper) functions:

$$\text{a dimension function } \mathsf{m}_{\mathbf{H}} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{m}_{\mathbf{H}}(t) := \mathsf{m}_{\hat{\mathbf{V}}}(t) + \mathsf{m}_{\boldsymbol{\Theta}}(t) + 1 = \ell(t) + \kappa(t) + \mathsf{m}_{\boldsymbol{\Theta}}(t) + 2 \ ,$$
$$\text{a time function } \mathsf{t}_{\mathbf{H}} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{t}_{\mathbf{H}}(t) := \mathsf{t}_{\mathrm{IRR}}(f(t)) + O(\ell(t) + \kappa(t) + \mathsf{m}_{\boldsymbol{\Theta}}(t)) \ ,$$
$$\text{a space function } \mathsf{s}_{\mathbf{H}} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{s}_{\mathbf{H}}(t) := \max\{\mathsf{s}_{\mathrm{IRR}}(f(t)), O(\ell(t) + \kappa(t) + \mathsf{m}_{\boldsymbol{\Theta}}(t))\} \ .$$

3. **Constructing Parameter 3.** Define the family $\mathbf{H} = \{H_t\}_{t\in\mathbb{N}}$ by

$$H_t := \widehat{V'}_t \cap \left(\widehat{V'}_t + \zeta_t(\mathsf{x})\right) \ .$$

   Observe that

$$H_t = \mathrm{span}\Big(\alpha_1^{\widehat{\mathbf{V}'}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\hat{\mathbf{V}}}(t)}^{\widehat{\mathbf{V}'}}(\mathsf{x}), \zeta_t(\mathsf{x})\Big) \ ,$$

   so that H_t is an $\mathsf{m}_{\mathbf{H}}(t)$-dimensional linear subset of $\mathbb{F}_t$ specified by a basis

$$\mathcal{B}_{H_t} := \Big(\alpha_1^{\mathbf{H}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\mathbf{H}}(t)}^{\mathbf{H}}(\mathsf{x})\Big) = \Big(\alpha_1^{\widehat{\mathbf{V}'}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}'}}(t)}^{\widehat{\mathbf{V}'}}(\mathsf{x}), \zeta_t(\mathsf{x})\Big) \ .$$

   Moreover, the basis $\mathcal{B}_{H_t}$ for H_t can be efficiently computed from the input 1^t by the following algorithm:

$\textsc{Find}\mathbf{H}(1^t) \equiv$

(a) Compute the irreducible polynomial $I_t := \textsc{FindIrrPoly}\big(1^{f(t)}\big)$.

(b) Compute the basis $\left(\alpha_1^{\widehat{\mathbf{V}'}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}'}}(t)}^{\widehat{\mathbf{V}'}}(\mathsf{x})\right) := \textsc{Find}\widehat{\mathbf{V}'}(1^t)$.

(c) Compute the element $\zeta_t(\mathsf{x}) := \textsc{Find}\mathbf{Z}(1^t)$.

(d) Output the basis $\left(\alpha_1^{\widehat{\mathbf{V}'}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}'}}(t)}^{\widehat{\mathbf{V}'}}(\mathsf{x}), \zeta_t(\mathsf{x})\right)$.

Note that $\textsc{Find}\mathbf{H}$ is a $\mathsf{t}_{\mathbf{H}}$-time $\textsc{Find}\mathbf{H}$-space algorithm, where:

$$\mathsf{t}_{\mathbf{H}}(t) := \mathsf{t}_{\mathrm{IRR}}(f(t)) + O(\ell(t) + \kappa(t) + \mathsf{m}_{\mathbf{\Theta}}(t)) \enspace ,$$
$$\mathsf{s}_{\mathbf{H}}(t) := \max\{\mathsf{s}_{\mathrm{IRR}}(f(t)), O(\ell(t) + \kappa(t) + \mathsf{m}_{\mathbf{\Theta}}(t))\} \enspace ,$$

matching the time and space complexities defined previously.

4. **Constructing Parameter 4.** Define the following three (proper) functions:

$$\text{an affine neighborhood size function } \mathsf{c}_{\mathbf{N}} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{c}_{\mathbf{N}}(t) := 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}(1 + |\mathcal{A}_{f(t),\kappa(t),\ell(t)}|) + 1 = 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}10 + 1 \enspace ,$$
$$\text{a time function } \mathsf{t}_{\mathbf{N}} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{t}_{\mathbf{N}}(t) := \mathsf{t}_{\mathrm{IRR}}(f(t)) + \mathsf{t}_{\mathrm{PRIM}}(\ell(t)) + 2\ell(t) + 2\kappa(t) + 1 \enspace ,$$
$$\text{a space function } \mathsf{s}_{\mathbf{N}} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{s}_{\mathbf{N}}(t) := \max\{\mathsf{s}_{\mathrm{IRR}}(f(t)), \mathsf{s}_{\mathrm{PRIM}}(\ell(t)), \ell(t) + \kappa(t) + 1\} \enspace .$$

Recall that $\mathcal{A}_{f(t),\kappa(t),\ell(t)}$ is the set of affine functions obtained by invoking Lemma 11.11 with the three positive integers $f(t)$, $\kappa(t)$, and $\ell(t)$.

5. **Constructing Parameter 5.** Define the family $\mathbf{N} = \{N_t\}_{t\in\mathbb{N}}$ by $N_t := \{\mathsf{aff}_{t,1}, \ldots, \mathsf{aff}_{t,\mathsf{c}_{\mathbf{N}}(t)} \colon \mathbb{F}_t \to \mathbb{F}_t\}$, where each $\mathsf{aff}_{t,i}$ is an affine function and is defined as follows:

- $\mathsf{aff}_{t,k}(z) := z + \theta^{(k)}(\mathsf{x})$, for $k = 1, \ldots, 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}$;
- $\mathsf{aff}_{t,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}+j+9(k-1)}(z) := \mathsf{aff}_{\mathrm{bin}(j-1)}(z)+\theta^{(k)}(\mathsf{x})$, for $j = 1, \ldots, 8$ and $k = 1, \ldots, 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}$, where $\mathsf{aff}_{000}, \ldots, \mathsf{aff}_{111}$ are 8 among the 9 affine functions in the set $\mathcal{A}_{f(t),\kappa(t),\ell(t)}$ obtained by invoking Lemma 11.11 with the three positive integers $f(t)$, $\kappa(t)$, and $\ell(t)$;
- $\mathsf{aff}_{t,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}+9+9(k-1)}(z) := \mathsf{aff}_{\times}(z) + \theta^{(k)}(\mathsf{x})$, for $k = 1, \ldots, 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}$, where $\mathsf{aff}_{\times}$ is the 9-th affine function in the set $\mathcal{A}_{f(t),\kappa(t),\ell(t)}$; and
- $\mathsf{aff}_{t,10\cdot 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}+1}(z) := z + \zeta_t(\mathsf{x})$.

Note that $\mathsf{c}_{\mathbf{N}}(t) = 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}(1 + |\mathcal{A}_{f(t),\kappa(t),\ell(t)}|) + 1$, so there are no more affine functions to define in N_t.

Moreover, the representation of each of the $\mathsf{c}_{\mathbf{N}}(t)$ affine functions in N_t can be efficiently computed from the input 1^t by the following algorithm: for $i \in \{1, \ldots, \mathsf{c}_{\mathbf{N}}(t)\}$,

$\textsc{Find}\mathbf{N}(1^t, i) \equiv$

(a) Compute the irreducible polynomial $I_t := \textsc{FindIrrPoly}\big(1^{f(t)}\big)$; it has root x.

(b) For $i = 1, \ldots, \mathsf{m}_{\mathbf{\Theta}}(t)$, compute the element $\theta_{t,i}(\mathsf{x}) := \textsc{Find}\mathbf{\Theta}(1^t, i)$.

(c) Compute the element $\zeta_t(\mathsf{x}) := \textsc{Find}\mathbf{Z}(1^t)$.

(d) Compute the primitive polynomial $\Xi_t := \textsc{FindPrimPoly}\big(1^{\ell(t)}\big)$.

(e) If $i \in \{1, \ldots, 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}\}$, then letting $\sigma_1 \cdots \sigma_{\mathsf{m}_{\mathbf{\Theta}}(t)} := \mathrm{bin}(i-1)$ output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (1_{\mathbb{F}_t}, \sum_{\ell=1}^{\mathsf{m}_{\mathbf{\Theta}}(t)} \sigma_\ell \theta_{t,\ell}(\mathsf{x}))$.

(f) If $i = 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}+j+9(k-1)$ for some $j \in \{1, \ldots, 9\}$ and $k \in \{1, \ldots, 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}\}$, then letting $\sigma_1 \cdots \sigma_{\mathsf{m}_{\mathbf{\Theta}}(t)} :=$ $\mathrm{bin}(k-1)$ and $\theta^{(k)}(\mathsf{x}) := \sum_{\ell=1}^{\mathsf{m}_{\mathbf{\Theta}}(t)} \sigma_\ell \theta_{t,\ell}(\mathsf{x})$,

   i. If $j = 1$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \theta^{(k)}(\mathsf{x}))$.

   ii. If $j = 2$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \mathsf{x}^{\ell(t)+\kappa(t)} + \theta^{(k)}(\mathsf{x}))$.

   iii. If $j = 3$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \mathsf{x}^{\ell(t)} + \theta^{(k)}(\mathsf{x}))$.

   iv. If $j = 4$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \mathsf{x}^{\ell(t)} + \mathsf{x}^{\ell(t)+\kappa(t)} + \theta^{(k)}(\mathsf{x}))$.

   v. If $j = 5$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \Xi_t(\mathsf{x}) + \theta^{(k)}(\mathsf{x}))$.

   vi. If $j = 6$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \Xi_t(\mathsf{x}) + \mathsf{x}^{\ell(t)+\kappa(t)} + \theta^{(k)}(\mathsf{x}))$.

   vii. If $j = 7$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \Xi_t(\mathsf{x}) + \mathsf{x}^{\ell(t)} + \theta^{(k)}(\mathsf{x}))$.

   viii. If $j = 8$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \Xi_t(\mathsf{x}) + \mathsf{x}^{\ell(t)} + \mathsf{x}^{\ell(t)+\kappa(t)} + \theta^{(k)}(\mathsf{x}))$.

   ix. If $j = 9$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (1_{\mathbb{F}_t}, \mathsf{x}^{\ell(t)+\kappa(t)} + \theta^{(k)}(\mathsf{x}))$.

(g) If $i = 10 \cdot 2^{\mathsf{m}_\Theta(t)} + 1$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (1_{\mathbb{F}_t}, \zeta_t(\mathsf{x}))$.

Note that FIND**N** is a $\mathsf{t}_\mathbf{N}$-time $\mathsf{s}_\mathbf{N}$-space algorithm, where:

$$\mathsf{t}_\mathbf{N}(t) := \mathsf{t}_{\text{IRR}}(f(t)) + \mathsf{t}_{\text{PRIM}}(\ell(t)) + O(\ell(t) + \kappa(t) + \mathsf{m}_\Theta(t)) \enspace,$$
$$\mathsf{s}_\mathbf{N}(t) := \max\{\mathsf{s}_{\text{IRR}}(f(t)), \mathsf{s}_{\text{PRIM}}(\ell(t)), O(\ell(t) + \kappa(t) + \mathsf{m}_\Theta(t))\} \enspace,$$

matching the time and space complexities defined previously.

6. **Constructing Parameter 6.** Define the following two (proper) functions:

$$\text{a time function } \mathsf{t}_\mathbf{D} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{t}_\mathbf{D}(t) := \mathsf{t}_\mathbf{K}(t) + \mathsf{t}_\mathbf{M}(t) + \mathsf{m}_\mathbf{H}(t) \enspace,$$
$$\text{a space function } \mathsf{s}_\mathbf{D} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{s}_\mathbf{D}(t) := \mathsf{t}_\mathbf{K}(t) + \mathsf{t}_\mathbf{M}(t) + \mathsf{m}_\mathbf{H}(t) \enspace.$$

7. **Constructing Parameter 7.** Define the family $\mathbf{D} = \big\{(d_0^t, d_1^t, \ldots, d_{\mathsf{c}_\mathbf{N}(t)}^t)\big\}_{t\in\mathbb{N}}$ by

$$d_0^t := 1 + 2^{\mathsf{m}_\mathbf{H}(t)}$$

$$d_i^t := \begin{cases} 1 + \max\limits_{j\in\{1,\ldots,\mathsf{c}_\mathbf{T}(t)\}} \big(\mathsf{D}_\mathbf{M}(t)[i\to j] \cdot \mathsf{D}_\mathbf{K}(t)[j]\big) & \text{if } i \in \{1,\ldots,\mathsf{m}_{\hat{\mathbf{V}}}(t)\} - \{\ell(t), \ell(t)+\kappa(t)\} \\ 1 + \max\limits_{j\in\{1,\ldots,\mathsf{c}_\mathbf{T}(t)\}} \big(\mathsf{D}_\mathbf{M}(t)[i\to j] \cdot \mathsf{D}_\mathbf{K}(t)[j]\big) & \\ \qquad + \sum_{j=1}^{\mathsf{c}_\mathbf{C}(t)} \sum_{r=1}^{2} \mathsf{D}_\mathbf{K}(t)[2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} + r\mathsf{c}_\mathbf{C}(t) + j] & \text{if } i = \ell(t) \\ 1 + \max\limits_{j\in\{1,\ldots,\mathsf{c}_\mathbf{T}(t)\}} \big(\mathsf{D}_\mathbf{M}(t)[i\to j] \cdot \mathsf{D}_\mathbf{K}(t)[j]\big) & \\ \qquad + \sum_{j=1}^{\mathsf{c}_\mathbf{C}(t)} \sum_{r=1}^{2} 2\mathsf{D}_\mathbf{K}(t)[2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} + r\mathsf{c}_\mathbf{C}(t) + j] & \text{if } i = \ell(t) + \kappa(t) \\ 0 & \text{if } i \in \{\mathsf{m}_{\hat{\mathbf{V}}}(t)+1, \ldots, 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil}\} \end{cases}$$

$$d_i^t := \begin{cases} \mathsf{D}_\mathbf{K}(t)[\mathsf{m}_{\hat{\mathbf{V}}}(t) + i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil}] & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} \in \{1,\ldots,\mathsf{c}_\mathbf{C}(t)\} \\ 0 & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} \in \{\mathsf{c}_\mathbf{C}(t)+1, \ldots, 2^{\lceil \log \mathsf{c}_\mathbf{C}(t)\rceil}\} \end{cases}$$

$$d_i^t := \begin{cases} \sum_{j=1}^{2} \mathsf{D}_\mathbf{K}(t)[\mathsf{m}_{\hat{\mathbf{V}}}(t) + j\mathsf{c}_\mathbf{C}(t) + i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} - 2^{\mathsf{m}_\Theta(t)}] & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} - 2^{\mathsf{m}_\Theta(t)} \in \{1,\ldots,\mathsf{c}_\mathbf{C}(t)\} \\ \sum_{j=1}^{2} \mathsf{D}_\mathbf{K}(t)[\mathsf{m}_{\hat{\mathbf{V}}}(t) + j\mathsf{c}_\mathbf{C}(t) + i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} - 2\cdot 2^{\mathsf{m}_\Theta(t)}] & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} - 2\cdot 2^{\mathsf{m}_\Theta(t)} \in \{1,\ldots,\mathsf{c}_\mathbf{C}(t)\} \\ \vdots & \vdots \\ \sum_{j=1}^{2} \mathsf{D}_\mathbf{K}(t)[\mathsf{m}_{\hat{\mathbf{V}}}(t) + j\mathsf{c}_\mathbf{C}(t) + i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} - 8\cdot 2^{\mathsf{m}_\Theta(t)}] & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} - 8\cdot 2^{\mathsf{m}_\Theta(t)} \in \{1,\ldots,\mathsf{c}_\mathbf{C}(t)\} \\ 0 & \text{for other } i \in \{2^{\mathsf{m}_\Theta(t)}+1, \ldots, 9\cdot 2^{\mathsf{m}_\Theta(t)}\} \end{cases}$$

$$d_i^t := \begin{cases} \mathsf{D}_\mathbf{K}(t)[\mathsf{m}_{\hat{\mathbf{V}}}(t) + 3\mathsf{c}_\mathbf{C}(t) + i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} - 9\cdot 2^{\mathsf{m}_\Theta(t)}] & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil} - 9\cdot 2^{\mathsf{m}_\Theta(t)} \in \{1,\ldots,\mathsf{c}_\mathbf{C}(t)\} \\ 0 & \text{for other } i \in \{9\cdot 2^{\mathsf{m}_\Theta(t)}+1, \ldots, 10\cdot 2^{\mathsf{m}_\Theta(t)}\} \end{cases}$$

$$d_{\mathsf{c}_\mathbf{N}(t)}^t := 2$$

Recall that $\mathsf{c}_\mathbf{N}(t) = 10 \cdot 2^{\mathsf{m}_\Theta(t)} + 1$ and $\mathsf{m}_\Theta(t) = \lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t)\rceil + \lceil \log \mathsf{c}_\mathbf{C}(t)\rceil$.

As dictated by Parameter 7, it must hold that

$$d_0^t + (2^{\mathsf{m}_\mathbf{H}(t)} - 1)\textstyle\sum_{i=1}^{\mathsf{c}_\mathbf{N}(t)} d_i^t \le 2^{f(t)-2} \text{ with the above definitions} \enspace; \tag{10}$$

indeed, this is how we set $f(t)$ in Equation 6.

Moreover, the i-th coordinate of the t-th vector can be efficiently computed from the input $(1^t, i)$ by an algorithm COMP**D**, on input $(1^t, i)$, by following the above definitions. Furthermore, it is easy to see that COMP**D** has the claimed time and space complexities.

8. **Constructing Parameter 8.** Define the following four (proper) functions:

$$\text{a size function } \mathsf{t}_\mathbf{P} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{t}_\mathbf{P}(t) := \mathsf{S}_\mathbf{K}(t) + \mathsf{S}_\mathbf{M}(t) + \mathsf{c}_\mathbf{N}(t) + \mathsf{m}_\mathbf{H}(t) \enspace,$$
$$\text{a degree function } \mathsf{s}_\mathbf{P} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{s}_\mathbf{P}(t)[i] := d_{i-1}^t \text{ for } i = 1, \ldots, 1 + \mathsf{c}_\mathbf{N}(t) \enspace,$$
$$\text{a time function } \mathsf{S}_\mathbf{P} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{S}_\mathbf{P}(t) := \mathsf{t}_\mathbf{K}(t) + \mathsf{t}_\mathbf{M}(t) + \mathsf{c}_\mathbf{N}(t) + \mathsf{m}_\mathbf{H}(t) \enspace,$$
$$\text{a space function } \mathsf{D}_\mathbf{P} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{D}_\mathbf{P}(t) := \mathsf{s}_\mathbf{K}(t) + \mathsf{s}_\mathbf{M}(t) + \mathsf{c}_\mathbf{N}(t) + \mathsf{m}_\mathbf{H}(t) \enspace.$$

9. **Constructing Parameter 9.** We proceed one step at a time:

- Define the family of Boolean functions

$$\mathbf{U} = \left\{ U_t \colon V_t \times C_t^{1+\alpha_{\mathbf{G}}(t)} \to \{0,1\} \right\}_{t \in \mathbb{N}}$$

by

$$U_t\big(v, c_0, c_1, \ldots, c_{\alpha_{\mathbf{G}}(t)}\big) := K_t\big(M_t(v), c_0, c_1, \ldots, c_{\alpha_{\mathbf{G}}(t)}\big) \enspace , \tag{11}$$

where:
  - $M_t \colon V_t \to T_t$ is the t-th Boolean function in the family $\mathbf{M}$, which gives the type of each vertex (see Parameter 8 in Definition 8.1); and
  - $K_t \colon T_t \times C_t^{1+\alpha_{\mathbf{G}}(t)} \to \{0,1\}$ is the t-th Boolean function in the family $\mathbf{K}$, which enforces constraints over the type of a vertex and the colors of the vertex and its neighbors (see Parameter 10 in Definition 8.1).

  Recall that $\alpha_{\mathbf{G}}(t) = 3$ in this reduction.

  Satisfiability. If a coloring $C \colon V_t \to C_t$ satisfies the coloring constraints induced by K_t and M_t (see Equation 1 in Definition 8.1), then, from the definition of U_t (see Equation 11), for every vertex $v \in V_t$,

$$U_t\Big(v, C(v), (C \circ \Gamma_{t,1})(v), \ldots, \big(C \circ \Gamma_{t,\alpha_{\mathbf{G}}(t)}\big)(v)\Big) = 0 \enspace . \tag{12}$$

  If so, we say that the coloring C satisfies the coloring constraints induced by U_t.
  The reverse direction also holds.

  Complexity. Recall that $\mathbf{M}$ is an $(\mathsf{S}_{\mathbf{M}}, \mathsf{D}_{\mathbf{M}}, \mathsf{t}_{\mathbf{M}}, \mathsf{s}_{\mathbf{M}})$-uniform family of Boolean functions and $\mathbf{K}$ is an $(\mathsf{S}_{\mathbf{K}}, \mathsf{D}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathsf{s}_{\mathbf{K}})$-uniform family of Boolean functions. Therefore, the family $\mathbf{U}$ is an $(\mathsf{S}_{\mathbf{U}}, \mathsf{D}_{\mathbf{U}}, \mathsf{t}_{\mathbf{U}}, \mathsf{s}_{\mathbf{U}})$-uniform family of Boolean functions with:

$$\begin{aligned}
\text{size function } \mathsf{S}_{\mathbf{U}}(t) &:= \mathsf{S}_{\mathbf{K}}(t) + \mathsf{S}_{\mathbf{M}}(t) \enspace , \\
\text{degree function } \mathsf{D}_{\mathbf{U}}(t)[i] &:= \begin{cases} \max\limits_{j \in \{1,\ldots,\mathsf{c}_{\mathbf{T}}(t)\}} \big(\mathsf{D}_{\mathbf{M}}(t)[i \to j] \cdot \mathsf{D}_{\mathbf{K}}(t)[j]\big) & \text{if } i \in \{1,\ldots,\log|V_t|\} \\ \mathsf{D}_{\mathbf{K}}(t)[i] & \text{if } i - \log|V_t| \in \{1,\ldots,(1+\alpha_{\mathbf{G}}(t))\mathsf{c}_{\mathbf{C}}(t)\} \end{cases} , \\
\text{time function } \mathsf{t}_{\mathbf{U}}(t) &:= \mathsf{t}_{\mathbf{K}}(t) + \mathsf{t}_{\mathbf{M}}(t) \enspace , \\
\text{space function } \mathsf{s}_{\mathbf{U}}(t) &:= \max\{\mathsf{s}_{\mathbf{K}}(t), \mathsf{s}_{\mathbf{M}}(t), \mathsf{S}_{\mathbf{K}}(t) + \mathsf{S}_{\mathbf{M}}(t)\} \enspace .
\end{aligned}$$

  Indeed, we can define a $\mathsf{t}_{\mathbf{U}}$-time $\mathsf{s}_{\mathbf{U}}$-space algorithm $\textsc{Find}\mathbf{U}$ as follows: for every $t \in \mathbb{N}$,

  $\textsc{Find}\mathbf{U}(1^t) \equiv$
  (a) Compute $[M_t]^{\mathsf{B}} := \textsc{Find}\mathbf{M}(1^t)$.
  (b) Compute $[K_t]^{\mathsf{B}} := \textsc{Find}\mathbf{K}(1^t)$.
  (c) Using $[M_t]^{\mathsf{B}}$ and $[K_t]^{\mathsf{B}}$, follow Equation 11 to obtain $[U_t]^{\mathsf{B}}$.
  (d) Output $[U_t]^{\mathsf{B}}$.

  Clearly, $\textsc{Find}\mathbf{U}(1^t)$ correctly generates a Boolean circuit $[U_t]^{\mathsf{B}}$ computing U_t for all $t \in \mathbb{N}$, and has the claimed complexity parameters.

- Define the family of Boolean functions

$$\mathbf{A} = \left\{ A_t \colon \mathbb{F}_t^{\mathsf{m}_{\hat{\mathbf{V}}}(t) + (1+\alpha_{\mathbf{G}}(t))\mathsf{c}_{\mathbf{C}}(t)} \to \mathbb{F}_t \right\}_{t \in \mathbb{N}}$$

by

$$A_t(x_1, \ldots, x_{\mathsf{m}_{\hat{\mathbf{V}}}(t) + (1+\alpha_{\mathbf{G}}(t))\mathsf{c}_{\mathbf{C}}(t)}) := \text{“the polynomial that directly arithmetizes } U_t\text{”} \enspace , \tag{13}$$

where:
  - $U_t \colon V_t \times C_t^{1+\alpha_{\mathbf{G}}(t)} \to \{0,1\}$ is the t-th Boolean function in the family $\mathbf{U}$, which enforces constraints over the colors of a vertex and its neighbors (see Equation 11).

Recall that $\alpha_{\mathbf{G}}(t) = 3$ in this reduction; moreover, elements in V_t can be described by $\mathsf{m}_{\hat{\mathbf{V}}}(t)$ bits.

Satisfiability. If a coloring $C\colon V_t \to C_t$ satisfies the constraints induced by U_t (see Equation 12), then, from the definition of A_t (see Equation 13), for every vertex $v \in V_t$,

$$\begin{aligned} A_t\Big(& v_1, \ldots, v_{\mathsf{m}_{\hat{\mathbf{V}}}(t)}, C'(v,1), \ldots, C'(v, \mathsf{c}_{\mathbf{C}}(t)), \\ & C'(\Gamma_{t,1}(v),1), \ldots, C'(\Gamma_{t,1}(v), \mathsf{c}_{\mathbf{C}}(t)), \\ & \ldots, \\ & C'(\Gamma_{t,\alpha_{\mathbf{G}}(t)}(v),1), \ldots, C'(\Gamma_{t,\alpha_{\mathbf{G}}(t)}(v), \mathsf{c}_{\mathbf{C}}(t))\Big) = 0_{\mathbb{F}_t} \ , \end{aligned} \tag{14}$$

where $v_1, \ldots, v_{\mathsf{m}_{\hat{\mathbf{V}}}(t)}$ is the $\{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$-binary representation of v and $C'\colon V_t \times \{1, \ldots, \mathsf{c}_{\mathbf{C}}(t)\} \to \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$ is defined by

$$C'(v,i) := 0_{\mathbb{F}_t} \text{ if the } i\text{-th bit of } C(v) \text{ is } 0 \text{ and } C'(v,i) := 1_{\mathbb{F}_t} \text{ otherwise.} \tag{15}$$

(In other words, C' is the "bit expansion" of C.)

If so, we say that the coloring C' satisfies the coloring constraints induced by A_t.

The reverse direction also holds by appropriately defining C out of C'.

Complexity. Recall that $\mathbf{U}$ is an $(\mathsf{S}_{\mathbf{U}}, \mathsf{D}_{\mathbf{U}}, \mathsf{t}_{\mathbf{U}}, \mathsf{s}_{\mathbf{U}})$-uniform family of Boolean functions. Therefore, the family $\mathbf{A}$ is an $(\mathsf{S}_{\mathbf{A}}, \mathsf{D}_{\mathbf{A}}, \mathsf{t}_{\mathbf{A}}, \mathsf{s}_{\mathbf{A}})$-uniform family of arithmetic circuits with:

$$\begin{aligned} \text{size function } & \mathsf{S}_{\mathbf{A}}(t) := \mathsf{S}_{\mathbf{U}}(t) \ , \\ \text{degree function } & \mathsf{D}_{\mathbf{A}}(t)[i] := \mathsf{D}_{\mathbf{U}}(t)[i] \text{ for } i = 1, \ldots, \mathsf{m}_{\hat{\mathbf{V}}}(t) + (1 + \alpha_{\mathbf{G}}(t))\mathsf{c}_{\mathbf{C}}(t) \ , \\ \text{time function } & \mathsf{t}_{\mathbf{A}}(t) := \mathsf{t}_{\mathbf{U}}(t) \ , \\ \text{space function } & \mathsf{s}_{\mathbf{A}}(t) := \mathsf{s}_{\mathbf{U}}(t) \ . \end{aligned}$$

Indeed, we can define the $\mathsf{t}_{\mathbf{A}}$-time $\mathsf{s}_{\mathbf{A}}$-space algorithm FIND**A** as follows: for every $t \in \mathbb{N}$,

FIND$\mathbf{A}(1^t) \equiv$

(a) Compute $[U_t]^{\mathsf{B}} :=$ FIND$\mathbf{U}(1^t)$.
(b) Let $[A_t]^{\mathsf{A}}$ be the direct arithmetization of $[U_t]^{\mathsf{B}}$.
(c) Output $[A_t]^{\mathsf{A}}$.

Clearly, FIND$\mathbf{A}(1^t)$ correctly generates an arithmetic circuit $[A_t]^{\mathsf{A}}$ computing A_t for all $t \in \mathbb{N}$, and has the claimed complexity parameters.

- Define the family of polynomials

$$\mathbf{R} = \left\{R_t\colon \mathbb{F}_t^{10 \cdot 2^{\mathsf{m}_{\Theta}(t)}} \to \mathbb{F}_t\right\}_{t \in \mathbb{N}} \ ,$$

by

$$\begin{aligned} & R_t(y_1^{(0)}, \ldots, y_{2^{\mathsf{m}_{\Theta}(t)}}^{(0)}, y_1^{(1)}, \ldots, y_{2^{\mathsf{m}_{\Theta}(t)}}^{(1)}, \ldots, y_1^{(9)}, \ldots, y_{2^{\mathsf{m}_{\Theta}(t)}}^{(9)}) \\ & \quad := A_t(y_1^{(0)}, \ldots, y_{\mathsf{m}_{\hat{\mathbf{V}}}(t)}^{(0)}, \\ & \qquad y_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+1}^{(0)}, \ldots, y_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+\mathsf{c}_{\mathbf{C}}(t)}^{(0)}, \\ & \qquad \mathrm{MUX}_{\mathbb{F}_{2^f},3}(y_{\ell(t)}^{(0)}, y_{\ell(t)+\kappa(t)}^{(0)}, y_{\ell(t)+\kappa(t)}^{(0)}, y_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+1}^{(1)}, \ldots, y_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+1}^{(8)}), \\ & \qquad \ldots, \\ & \qquad \mathrm{MUX}_{\mathbb{F}_{2^f},3}(y_{\ell(t)}^{(0)}, y_{\ell(t)+\kappa(t)}^{(0)}, y_{\ell(t)+\kappa(t)}^{(0)}, y_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+\mathsf{c}_{\mathbf{C}}(t)}^{(1)}, \ldots, y_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+\mathsf{c}_{\mathbf{C}}(t)}^{(8)}), \\ & \qquad \mathrm{MUX}_{\mathbb{F}_{2^f},3}(y_{\ell(t)}^{(0)}, 1_{\mathbb{F}_t} + y_{\ell(t)+\kappa(t)}^{(0)}, y_{\ell(t)+\kappa(t)}^{(0)}, y_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+1}^{(1)}, \ldots, y_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+1}^{(8)}), \\ & \qquad \ldots, \\ & \qquad \mathrm{MUX}_{\mathbb{F}_{2^f},3}(y_{\ell(t)}^{(0)}, 1_{\mathbb{F}_t} + y_{\ell(t)+\kappa(t)}^{(0)}, y_{\ell(t)+\kappa(t)}^{(0)}, y_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+\mathsf{c}_{\mathbf{C}}(t)}^{(1)}, \ldots, y_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+\mathsf{c}_{\mathbf{C}}(t)}^{(8)}), \end{aligned} \tag{16}$$

$$y^{(9)}_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+1}, \ldots, y^{(9)}_{2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}+\mathsf{c}_{\mathbf{C}}(t)}\Big) \enspace ,$$

where:

- $A_t \colon \mathbb{F}_t^{\mathsf{m}_{\hat{\mathbf{V}}}(t)+(1+\alpha_{\mathbf{G}}(t))\mathsf{c}_{\mathbf{C}}(t)} \to \mathbb{F}_t$ is the t-th polynomial in the family $\mathbf{A}$, which enforces constraints over the type of a vertex and the colors of the vertex and its neighbors (see Equation 13), and
- $\mathrm{MUX}_{\mathbb{F}_{2^f},3}$ is the 3-bit multiplexer polynomial over the field $\mathbb{F}_{2^f}$ from Definition E.22.

Recall that $\mathsf{m}_{\Theta}(t) = \lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil + \lceil \log \mathsf{c}_{\mathbf{C}}(t) \rceil$.

Satisfiability. Suppose that a coloring $C' \colon V_t \times \{1, \ldots, \mathsf{c}_{\mathbf{C}}(t)\} \to \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$ satisfies the constraints induced by A_t (see Equation 14). Define the function $C'' \colon V_t \times \{1, \ldots, 2^{\mathsf{m}_{\Theta}(t)}\} \to \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$ as follows: for every $v \in V_t$ and $i \in \{1, \ldots, 2^{\mathsf{m}_{\Theta}(t)}\}$,

$$C''(v,i) := \begin{cases} v_i & \text{if } i \in \{1, \ldots, \mathsf{m}_{\hat{\mathbf{V}}}(t)\} \\ C'(v, i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}) & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} \in \{1, \ldots, \mathsf{c}_{\mathbf{C}}(t)\} \\ 0_{\mathbb{F}_t} & \text{otherwise} \end{cases} \enspace . \tag{17}$$

Define the function $\Phi_{\widehat{V'}_t} \colon V_t \times \{1, \ldots, 2^{\mathsf{m}_{\Theta}(t)}\} \to \widehat{V'}_t$ as follows: for every $v \in V_t$ and $k \in \{1, \ldots, 2^{\mathsf{m}_{\Theta}(t)}\}$,

$$\Phi_{\widehat{V'}_t}(v,k) := \Phi_{f(t),\kappa(t),\ell(t)}(v) + \theta^{(k)}(\mathsf{x}) \enspace .$$

Also define the function $\Phi^{-1}_{\widehat{V'}_t} \colon \widehat{V'}_t \to V_t \times \{1, \ldots, 2^{\mathsf{m}_{\Theta}(t)}\}$ to be the inverse of $\Phi_{\widehat{V'}_t}$; namely, for every $\omega(\mathsf{x}) \in \widehat{V'}_t$, we can write $\omega(\mathsf{x}) = \alpha(\mathsf{x}) + \theta^{(k)}(\mathsf{x})$ for some $\alpha(\mathsf{x}) \in V_t$ and $k \in \{1, \ldots, 2^{\mathsf{m}_{\Theta}(t)}\}$ and then we define

$$\Phi^{-1}_{\widehat{V'}_t}(\omega(\mathsf{x})) := \big(\Phi^{-1}_{f(t),\kappa(t),\ell(t)}(\alpha(\mathsf{x})), k\big) \enspace .$$

We note that, for any $\alpha(\mathsf{x}) \in \hat{V}_t$, $\alpha(\mathsf{x}) + \theta^{(1)}(\mathsf{x}), \ldots, \alpha(\mathsf{x}) + \theta^{(2^{\mathsf{m}_{\Theta}(t)})}(\mathsf{x})$ all share the same "low-order bits". In particular, it is easy to see that, for $j = 1, 2$ and $k \in \{1, \ldots, 2^{\mathsf{m}_{\Theta}(t)}\}$,

$$\mathrm{MUX}_{\mathbb{F}_{2^f},3}(b_{j,1}, b_{j,2}, b_{j,3}, x_1, \ldots, x_8) = x_i$$

where $i \in \{1, \ldots, 8\}$ is the (unique) index such that

$$\Big(\Phi_{\widehat{V'}_t} \circ \Gamma_j \circ \Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x}) + \theta^{(k)}(\mathsf{x})\big) = \mathsf{aff}_{\mathrm{bin}(i-1)}\big(\alpha(\mathsf{x}) + \theta^{(k)}(\mathsf{x})\big)$$

and where:

- $b_{j,1}$ is the $\ell(t)$-th bit of $\Phi^{-1}_{\widehat{V'}_t}(\alpha(\mathsf{x}) + \theta^{(k)}(\mathsf{x}))$;
- $b_{j,2}$ is the $(\ell(t) + \kappa(t)$-th bit of $\Phi^{-1}_{\widehat{V'}_t}(\alpha(\mathsf{x}) + \theta^{(k)}(\mathsf{x}))$, and XORed with 1 if $j = 2$; and
- $b_{j,3}$ is the $(\ell(t) + \kappa(t)$-th bit of $\Phi^{-1}_{\widehat{V'}_t}(\alpha(\mathsf{x}) + \theta^{(k)}(\mathsf{x}))$.

Therefore, we see that we are using the MUX polynomials in the definition of R_t to "select out" the correct neighbor.[11]

Define the function $\widetilde{A} \colon \widehat{V'}_t \to \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$ by

$$\widetilde{A}\big(\alpha(\mathsf{x})\big) := \Big(C'' \circ \Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})\big) \enspace ; \tag{18}$$

and then define the univariate polynomial $A \colon \mathbb{F}_t \to \mathbb{F}_t$ to be the low-degree extension (see Theorem E.14) of $\widetilde{A}$ in $\mathbb{F}_t$, i.e.,

$$A := \mathrm{LDE}_{\mathbb{F}_t, 1, \widehat{V'}_t}(\widetilde{A}) \enspace . \tag{19}$$

[11] Indeed, the double extended De Bruijn graph embedding $\Phi_{f(t),\kappa(t),\ell(t)}$ from Lemma 11.11 was proved for an affine graph with *nine* edges per field element, while the regularity of $\mathrm{DDB}(\kappa(t), 2^{\ell(t)} - 1)$ is only *three*. We thus need to compute the indices of the two "real" affine neighbors (among the eight for which we have uncertainty) of an element $\omega(\mathsf{x})$ in the image of $\Phi_{f(t),\kappa(t),\ell(t)}$ (or, more precisely, $\Phi_{\widehat{V'}_t}$), i.e., those De Bruijn edges that are in fact induced by each De Bruijn graph in $\mathrm{DDB}(\kappa(t), 2^{\ell(t)} - 1)$. (Note that there is no uncertainty for the "cross" edges between the two graphs.)

Then, for every $\alpha(\mathsf{x}) \in \widehat{V}_t$,

$$
\begin{aligned}
&R_t\Big(\big(A\big(\mathsf{aff}_{t,k}(\alpha(\mathsf{x}))\big)\big)_{k=1,\dots,10\cdot 2^{\mathsf{m}_{\mathbf{\Theta}}(t)}}\Big) \qquad (20)\\
&= A_t\Big(\big(A\big(\mathsf{aff}_{t,k}(\alpha(\mathsf{x}))\big)\big)_{k=1,\dots,\mathsf{m}_{\widehat{\mathbf{V}}}(t)},\\
&\quad \big(A\big(\mathsf{aff}_{t,k}(\alpha(\mathsf{x}))\big)\big)_{k-2^{\lceil \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)},\\
&\quad\quad \Big(\mathrm{MUX}_{\mathbb{F}_{2^f},3}\Big(A\big(\mathsf{aff}_{t,\ell(t)}(\alpha(\mathsf{x}))\big), A\big(\mathsf{aff}_{t,\ell(t)+\kappa(t)}(\alpha(\mathsf{x}))\big), A\big(\mathsf{aff}_{t,\ell(t)+\kappa(t)}(\alpha(\mathsf{x}))\big),\\
&\quad\quad\quad A\big(\mathsf{aff}_{t,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}+1+9(k-1)}(\alpha(\mathsf{x}))\big),\dots, A\big(\mathsf{aff}_{t,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}+8+9(k-1)}(\alpha(\mathsf{x}))\big)\Big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}},\\
&\quad\quad \Big(\mathrm{MUX}_{\mathbb{F}_{2^f},3}\Big(A\big(\mathsf{aff}_{t,\ell(t)}(\alpha(\mathsf{x}))\big), 1_{\mathbb{F}_t} + A\big(\mathsf{aff}_{t,\ell(t)+\kappa(t)}(\alpha(\mathsf{x}))\big), A\big(\mathsf{aff}_{t,\ell(t)+\kappa(t)}(\alpha(\mathsf{x}))\big),\\
&\quad\quad\quad A\big(\mathsf{aff}_{t,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}+1+9(k-1)}(\alpha(\mathsf{x}))\big),\dots, A\big(\mathsf{aff}_{t,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}+8+9(k-1)}(\alpha(\mathsf{x}))\big)\Big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}},\\
&\quad\quad\quad\quad \big(A\big(\mathsf{aff}_{t,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}+9+9(k-1)}(\alpha(\mathsf{x}))\big)\big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}}\Big)\\
&= A_t\Big(\big(A\big(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x})\big)\big)_{k=1,\dots,\mathsf{m}_{\widehat{\mathbf{V}}}(t)},\\
&\quad \big(A\big(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x})\big)\big)_{k-2^{\lceil \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)},\\
&\quad\quad \Big(\mathrm{MUX}_{\mathbb{F}_{2^f},3}\Big(A\big(\alpha(\mathsf{x})+\theta^{(\ell(t))}(\mathsf{x})\big), A\big(\alpha(\mathsf{x})+\theta^{(\ell(t)+\kappa(t))}(\mathsf{x})\big), A\big(\alpha(\mathsf{x})+\theta^{(\ell(t)+\kappa(t))}(\mathsf{x})\big),\\
&\quad\quad\quad A\big(\mathsf{aff}_{000}(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x}))\big),\dots, A\big(\mathsf{aff}_{111}(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x}))\big)\Big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}},\\
&\quad\quad \Big(\mathrm{MUX}_{\mathbb{F}_{2^f},3}\Big(A\big(\alpha(\mathsf{x})+\theta^{(\ell(t))}(\mathsf{x})\big), 1_{\mathbb{F}_t} + A\big(\alpha(\mathsf{x})+\theta^{(\ell(t)+\kappa(t))}(\mathsf{x})\big), A\big(\alpha(\mathsf{x})+\theta^{(\ell(t)+\kappa(t))}(\mathsf{x})\big),\\
&\quad\quad\quad A\big(\mathsf{aff}_{000}(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x}))\big),\dots, A\big(\mathsf{aff}_{111}(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x}))\big)\Big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}},\\
&\quad\quad\quad\quad \big(A\big(\mathsf{aff}_{\times}(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x}))\big)\big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,2^{\mathsf{m}_{\mathbf{\Theta}}(t)}}\Big)\\
&= A_t\Big(\big(A\big(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x})\big)\big)_{k=1,\dots,\mathsf{m}_{\widehat{\mathbf{V}}}(t)},\\
&\quad \big(A\big(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x})\big)\big)_{k-2^{\lceil \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)},\\
&\quad\quad \Big(\Big(A\circ\Phi_{\widehat{V'}_t}\circ\Gamma_{t,1}\circ\Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})+\theta^{(k)}\big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)},\\
&\quad\quad \Big(\Big(A\circ\Phi_{\widehat{V'}_t}\circ\Gamma_{t,2}\circ\Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})+\theta^{(k)}\big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)},\\
&\quad\quad \Big(\Big(A\circ\Phi_{\widehat{V'}_t}\circ\Gamma_{t,3}\circ\Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})+\theta^{(k)}\big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)}\Big)\\
&= A_t\Big(\big(\widetilde{A}\big(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x})\big)\big)_{k=1,\dots,\mathsf{m}_{\widehat{\mathbf{V}}}(t)},\\
&\quad \big(\widetilde{A}\big(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x})\big)\big)_{k-2^{\lceil \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)},\\
&\quad\quad \Big(\Big(\widetilde{A}\circ\Phi_{\widehat{V'}_t}\circ\Gamma_{t,1}\circ\Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})+\theta^{(k)}\big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)},\\
&\quad\quad \Big(\Big(\widetilde{A}\circ\Phi_{\widehat{V'}_t}\circ\Gamma_{t,2}\circ\Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})+\theta^{(k)}\big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)},\\
&\quad\quad \Big(\Big(\widetilde{A}\circ\Phi_{\widehat{V'}_t}\circ\Gamma_{t,3}\circ\Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})+\theta^{(k)}\big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)}\Big)\\
&= A_t\Big(\Big(\Big(C''\circ\Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x})\big)\Big)_{k=1,\dots,\mathsf{m}_{\widehat{\mathbf{V}}}(t)},\\
&\quad \Big(\Big(C''\circ\Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})+\theta^{(k)}(\mathsf{x})\big)\Big)_{k-2^{\lceil \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)},\\
&\quad\quad \Big(\Big(C''\circ\Gamma_{t,1}\circ\Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})+\theta^{(k)}\big)\Big)_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=1,\dots,\mathsf{c}_{\mathbf{C}}(t)},
\end{aligned}
$$

$$\Big(\Big(C'' \circ \Gamma_{t,2} \circ \Phi^{-1}_{\widehat{V'}_t}\Big)\left(\alpha(\mathsf{x}) + \theta^{(k)}\right)\Big)_{k-2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}=1,\ldots,\mathsf{c}_{\mathbf{C}}(t)},$$

$$\Big(\Big(C'' \circ \Gamma_{t,3} \circ \Phi^{-1}_{\widehat{V'}_t}\Big)\left(\alpha(\mathsf{x}) + \theta^{(k)}\right)\Big)_{k-2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}=1,\ldots,\mathsf{c}_{\mathbf{C}}(t)}\Big)$$

$$= A_t\Big(\left(C''(v,i)\right)_{k=1,\ldots,\mathsf{m}_{\hat{\mathbf{V}}}(t)},$$
$$\left(C''(v,i)\right)_{k-2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}=1,\ldots,\mathsf{c}_{\mathbf{C}}(t)},$$
$$\left(C''(\Gamma_{t,1}(v),i)\right)_{k-2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}=1,\ldots,\mathsf{c}_{\mathbf{C}}(t)},$$
$$\left(C''(\Gamma_{t,2}(v),i)\right)_{k-2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}=1,\ldots,\mathsf{c}_{\mathbf{C}}(t)},$$
$$\left(C''(\Gamma_{t,3}(v),i)\right)_{k-2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}=1,\ldots,\mathsf{c}_{\mathbf{C}}(t)}\Big) = 0_{\mathbb{F}_t} \quad ,$$

where in the second to last step we have used definition of $\widetilde{A}$ (see Equation 18) and in the last step we let $v \in V_t$ be such that $\Phi_{f(t),\kappa(t),\ell(t)}(v) = \alpha(\mathsf{x})$. In other words, the univariate polynomial $A\colon \mathbb{F}_t \to \mathbb{F}_t$ is such that, for every $\alpha(\mathsf{x}) \in \widehat{V}_t$,

$$R_t\Big(\Big(A\big(\mathsf{aff}_{t,k}(\alpha(\mathsf{x}))\big)\Big)_{k=1,\ldots,10\cdot 2^{\mathsf{m}_{\Theta}(t)}}\Big) = 0_{\mathbb{F}_t} \quad . \tag{21}$$

Moreover, $A(\widehat{V'}_t) \subseteq \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$ and for every $\alpha(\mathsf{x}) \in \widehat{V}_t$ we have that $\Phi^{-1}_{\widehat{V}_t}(\alpha(\mathsf{x})) = \Big(A\big(\mathsf{aff}_{t,k}(\alpha(\mathsf{x}))\big)\Big)_{k=1,\ldots,\mathsf{m}_{\hat{\mathbf{V}}}(t)}$. If both of the previous conditions hold, we say that the assignment polynomial A satisfies the constraint polynomial R_t.

The reverse direction also holds by appropriately defining C' out of A.

Complexity. Recall that $\mathbf{A}$ is an $(\mathsf{S}_{\mathbf{A}}, \mathsf{D}_{\mathbf{A}}, \mathsf{t}_{\mathbf{A}}, \mathsf{s}_{\mathbf{A}})$-uniform family of Boolean functions. Therefore, the family $\mathbf{R}$ is an $(\mathsf{S}_{\mathbf{R}}, \mathsf{D}_{\mathbf{R}}, \mathsf{t}_{\mathbf{R}}, \mathsf{s}_{\mathbf{R}})$-uniform family of arithmetic circuits with:

$$\text{size function } \mathsf{S}_{\mathbf{R}}(t) := \mathsf{S}_{\mathbf{A}}(t) + 10 \cdot 2^{\mathsf{m}_{\Theta}(t)} \quad ,$$

degree function $\mathsf{D}_{\mathbf{R}}(t)[i]$

$$:= \begin{cases} \mathsf{D}_{\mathbf{A}}(t)[i] & \text{if } i \in \{1,\ldots,\mathsf{m}_{\hat{\mathbf{V}}}(t)\} - \{\ell(t), \ell(t)+\kappa(t)\} \\ \mathsf{D}_{\mathbf{A}}(t)[i] + \sum_{j=1}^{\mathsf{c}_{\mathbf{C}}(t)} \mathsf{D}_{\mathbf{A}}(t)[2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} + \mathsf{c}_{\mathbf{C}}(t) + j] + \mathsf{D}_{\mathbf{A}}(t)[2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} + 2\mathsf{c}_{\mathbf{C}}(t) + j] & \text{if } i = \ell(t) \\ \mathsf{D}_{\mathbf{A}}(t)[i] + \sum_{j=1}^{\mathsf{c}_{\mathbf{C}}(t)} 2\mathsf{D}_{\mathbf{A}}(t)[2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} + \mathsf{c}_{\mathbf{C}}(t) + j] + 2\mathsf{D}_{\mathbf{A}}(t)[2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} + 2\mathsf{c}_{\mathbf{C}}(t) + j] & \text{if } i = \ell(t)+\kappa(t) \\ \mathsf{D}_{\mathbf{A}}(t)[\mathsf{m}_{\hat{\mathbf{V}}}(t) + i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil}] & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} \in \{1,\ldots,\mathsf{c}_{\mathbf{C}}(t)\} \\ \sum_{j=1}^{2} \mathsf{D}_{\mathbf{A}}(t)[\mathsf{m}_{\hat{\mathbf{V}}}(t) + j\mathsf{c}_{\mathbf{C}}(t) + i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} - 2^{\mathsf{m}_{\Theta}(t)}] & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} - 2^{\mathsf{m}_{\Theta}(t)} \in \{1,\ldots,\mathsf{c}_{\mathbf{C}}(t)\} \\ \sum_{j=1}^{2} \mathsf{D}_{\mathbf{A}}(t)[\mathsf{m}_{\hat{\mathbf{V}}}(t) + j\mathsf{c}_{\mathbf{C}}(t) + i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} - 2 \cdot 2^{\mathsf{m}_{\Theta}(t)}] & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} - 2 \cdot 2^{\mathsf{m}_{\Theta}(t)} \in \{1,\ldots,\mathsf{c}_{\mathbf{C}}(t)\} \\ \vdots & \vdots \\ \sum_{j=1}^{2} \mathsf{D}_{\mathbf{A}}(t)[\mathsf{m}_{\hat{\mathbf{V}}}(t) + j\mathsf{c}_{\mathbf{C}}(t) + i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} - 8 \cdot 2^{\mathsf{m}_{\Theta}(t)}] & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} - 8 \cdot 2^{\mathsf{m}_{\Theta}(t)} \in \{1,\ldots,\mathsf{c}_{\mathbf{C}}(t)\} \\ \mathsf{D}_{\mathbf{A}}(t)[\mathsf{m}_{\hat{\mathbf{V}}}(t) + 3\mathsf{c}_{\mathbf{C}}(t) + i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} - 9 \cdot 2^{\mathsf{m}_{\Theta}(t)}] & \text{if } i - 2^{\lceil \log \mathsf{m}_{\hat{\mathbf{V}}}(t) \rceil} - 9 \cdot 2^{\mathsf{m}_{\Theta}(t)} \in \{1,\ldots,\mathsf{c}_{\mathbf{C}}(t)\} \\ 0 & \text{otherwise} \end{cases},$$

$$\text{time function } \mathsf{t}_{\mathbf{R}}(t) := \mathsf{t}_{\mathbf{A}}(t) + 10 \cdot 2^{\mathsf{m}_{\Theta}(t)} \quad ,$$
$$\text{space function } \mathsf{s}_{\mathbf{R}}(t) := \mathsf{s}_{\mathbf{A}}(t) + 10 \cdot 2^{\mathsf{m}_{\Theta}(t)} \quad .$$

Therefore, we can define an algorithm $\textsc{Find}\mathbf{R}$ as follows: for every $t \in \mathbb{N}$,

$\textsc{Find}\mathbf{R}(1^t) \equiv$
(a) Compute $[A_t]^{\mathsf{A}} := \textsc{Find}\mathbf{A}(1^t)$.
(b) Using $[A_t]^{\mathsf{A}}$, follow Equation 16 to compute $[R]^{\mathsf{A}}$.
(c) Output $[R_t]^{\mathsf{A}}$.

Clearly, $\textsc{Find}\mathbf{R}(1^t)$ correctly generates an arithmetic circuit $[R_t]^{\mathsf{A}}$ computing R_t for all $t \in \mathbb{N}$.

- Define the family of polynomials

$$\mathbf{Q} = \Big\{Q_t \colon \mathbb{F}_t^{1+10\cdot 2^{\mathsf{m}_{\Theta}(t)}} \to \mathbb{F}_t\Big\}_{t \in \mathbb{N}}$$

by

$$Q_t(x_0, x_1, \ldots, x_{10 \cdot 2^{\mathsf{m}_\Theta(t)}}) := J\Big(R_t(x_1, \ldots, x_{10 \cdot 2^{\mathsf{m}_\Theta(t)}}), x_0 - P_{\mathbf{\Phi}}^{f(t),\ell(t),\kappa(t)}(x_1, \ldots, x_{\mathsf{m}_{\hat{\mathbf{V}}}(t)})\Big) \ , \tag{22}$$

where:

- $J(x_1, x_2) := x_1 x_2 + x_1 + x_2$ and note that $J(\alpha_1(\mathsf{x}), \alpha_2(\mathsf{x})) = 0_{\mathbb{F}_t}$ if and only if $\alpha_1(\mathsf{x}) = \alpha_2(\mathsf{x}) = 0_{\mathbb{F}_t}$;
- $P_{\mathbf{\Phi}}^{f(t),\ell(t),\kappa(t)}$ is the polynomial computing $\Phi_{f(t),\kappa(t),\ell(t)}$, obtained from Lemma 11.12; and
- $R_t \colon \mathbb{F}_t^{10 \cdot 2^{\mathsf{m}_\Theta(t)}} \to \mathbb{F}_t$ is the t-th polynomial in the family $\mathbf{R}$ (see Equation 16).

Satisfiability. Suppose that an assignment polynomial $A \colon \mathbb{F}_t \to \mathbb{F}_t$ satisfies the constraints polynomial R_t (see Equation 21). Then, for every $\alpha(\mathsf{x}) \in \widehat{V}_t$,

$$R_t\Big(\Big(A\big(\mathsf{aff}_{t,\mathsf{c}_\mathbf{N}(t)}(\alpha(\mathsf{x}))\big)\Big)_{k=1,\ldots,10 \cdot 2^{\mathsf{m}_\Theta(t)}}\Big) = 0_{\mathbb{F}_t} \ ;$$

moreover, because for every $\alpha(\mathsf{x}) \in \widehat{V}_t$ we have that $\Phi_{\widehat{V}_t}^{-1}(\alpha(\mathsf{x})) = \big(A\big(\mathsf{aff}_{t,k}(\alpha(\mathsf{x}))\big)\big)_{k=1,\ldots,\mathsf{m}_{\hat{\mathbf{V}}}(t)}$, we also have that

$$\alpha(\mathsf{x}) - P_{\mathbf{\Phi}}^{f(t),\ell(t),\kappa(t)}(A\big(\mathsf{aff}_{t,1}(\alpha(\mathsf{x}))\big), \ldots, A\big(\mathsf{aff}_{t,\mathsf{m}_{\hat{\mathbf{V}}}(t)}(\alpha(\mathsf{x}))\big)) = 0_{\mathbb{F}_t} \ .$$

Thus, $A(\widehat{V'}_t) \subseteq \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$ and, moreover, for every $\alpha(\mathsf{x}) \in \widehat{V}_t$,

$$Q_t\Big(\alpha(\mathsf{x}), A\big(\mathsf{aff}_{t,1}(\alpha(\mathsf{x}))\big), \ldots, A\big(\mathsf{aff}_{t,\mathsf{c}_\mathbf{N}(t)}(\alpha(\mathsf{x}))\big)\Big) = 0_{\mathbb{F}_t} \ . \tag{23}$$

If so, we say that the assignment polynomial A satisfies the constraint polynomial P_t.
The reverse direction also holds.

Complexity. Recall that the family $\mathbf{R}$ is an $(\mathsf{S}_\mathbf{R}, \mathsf{D}_\mathbf{R}, \mathsf{t}_\mathbf{R}, \mathsf{s}_\mathbf{R})$-uniform family of polynomials. Furthermore, recall from Lemma 11.12 that the (multilinear) polynomial $P_{\mathbf{\Phi}}^{f(t),\ell(t),\kappa(t)}$ can be constructed and evaluated in time $O(\kappa(t) + 2^{\ell(t)})$. Therefore, $\mathbf{Q}$ is a $(\mathsf{S}_\mathbf{Q}, \mathsf{D}_\mathbf{Q}, \mathsf{t}_\mathbf{Q}, \mathsf{s}_\mathbf{Q})$-uniform family of polynomials with:

$$\text{size function } \mathsf{S}_\mathbf{Q}(t) := \mathsf{S}_\mathbf{R}(t) + O(\kappa(t) + 2^{\ell(t)}) \ ,$$

$$\text{degree function } \mathsf{D}_\mathbf{Q}(t)[i] := \begin{cases} 1 & \text{if } i = 1 \\ \mathsf{D}_\mathbf{R}[i-1] + 1 & \text{if } i - 1 \in \{1, \ldots, \mathsf{m}_{\hat{\mathbf{V}}}(t)\} \\ \mathsf{D}_\mathbf{R}[i-1] & \text{otherwise} \end{cases} \ ,$$

$$\text{time function } \mathsf{t}_\mathbf{Q}(t) := \mathsf{t}_\mathbf{R}(t) + O(\kappa(t) + 2^{\ell(t)}) \ ,$$

$$\text{space function } \mathsf{s}_\mathbf{Q}(t) := \mathsf{s}_\mathbf{R}(t) + O(\kappa(t) + 2^{\ell(t)}) \ .$$

Indeed, we can define the $\mathsf{t}_\mathbf{Q}$-time $\mathsf{s}_\mathbf{Q}$-space algorithm Find$\mathbf{Q}$ as follows: for every $t \in \mathbb{N}$,

Find$\mathbf{Q}(1^t) \equiv$

(a) Compute $[R_t]^\mathsf{A} = \text{Find}\mathbf{R}(1^t)$.
(b) Compute $[P_{\mathbf{\Phi}}^{f(t),\ell(t),\kappa(t)}]^\mathsf{A}$.
(c) Compute $[Q_t]^\mathsf{A}$, using $[R_t]^\mathsf{A}$ and $[P_{\mathbf{\Phi}}^{f(t),\ell(t),\kappa(t)}]^\mathsf{A}$, following Equation 22.
(d) Output $[Q_t]^\mathsf{A}$.

Clearly, Find$\mathbf{Q}(1^t)$ correctly generates an $\mathbb{F}_t$-arithmetic circuit $[Q_t]^\mathsf{A}$ computing Q_t for all $t \in \mathbb{N}$, and has the claimed complexity parameters.

- Define the family of polynomials

$$\mathbf{P} = \Big\{P_t \colon \mathbb{F}_t^{1+\mathsf{c}_\mathbf{N}(t)} \to \mathbb{F}_t\Big\}_{t \in \mathbb{N}}$$

by

$$P_t(x_0, x_1, \ldots, x_{\mathsf{c}_\mathbf{N}(t)}) := Y_{\mathbb{F}_t, H_t, \widehat{V}_t}(x_0) \cdot Q_t(x_0, x_1, \ldots, x_{\mathsf{c}_\mathbf{N}(t)-1}) + Y_{\mathbb{F}_t, H_t, \widehat{V'}_t}(x_0 + \zeta_t(\mathsf{x})) \cdot Z_{\mathbb{F}_t, \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}}(x_{\mathsf{c}_\mathbf{N}(t)}) \ , \tag{24}$$

where:

– $Y_{\mathbb{F}_t,H_t,\widehat{V}_t}$ is the low-degree extension in $\mathbb{F}_t$ of the function that is equal to 1 on $\widehat{V}_t$ and is equal to 0 on $H_t - \widehat{V}_t$ (see Theorem E.23); similarly for $Y_{\mathbb{F}_t,H_t,\widehat{V'}_t}$;
– $Z_{\mathbb{F}_t,\{0_{\mathbb{F}_t},1_{\mathbb{F}_t}\}}$ is the vanishing polynomial on the (linear) subset $\{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$ in $\mathbb{F}_t$ (see Definition E.9); and
– $Q_t \colon \mathbb{F}_t^{1+10\cdot 2^{\mathsf{m}_{\Theta}(t)}} \to \mathbb{F}_t$ is the t-th polynomial in the family $\mathbf{Q}$ (see Equation 22).

Recall that $\mathsf{c}_{\mathbf{N}}(t) = 10 \cdot 2^{\mathsf{m}_{\Theta}(t)} + 1$.

Satisfiability. Suppose that an assignment polynomial $A \colon \mathbb{F}_t \to \mathbb{F}_t$ satisfies the constraint polynomial Q_t (see Equation 23). Then, for every $\alpha(\mathsf{x}) \in H_t$:

– If $\alpha(\mathsf{x}) \in \widehat{V}_t$, we have that $Y_{\mathbb{F}_t,H_t,\widehat{V}_t}\big(\alpha(\mathsf{x})\big) = 1_{\mathbb{F}_t}$ and $Y_{\mathbb{F}_t,H_t,\widehat{V'}_t}\big(\alpha(\mathsf{x}) + \zeta_t(\mathsf{x})\big) = 0_{\mathbb{F}_t}$, so that

$$\begin{aligned} & P_t\Big(\alpha(\mathsf{x}), \Big(A\big(\mathsf{aff}_{t,\mathsf{c}_{\mathbf{N}}(t)}(\alpha(\mathsf{x}))\big)\Big)_{k=1,\dots,\mathsf{c}_{\mathbf{N}}(t)}\Big) \\ &= Q_t\Big(\alpha(\mathsf{x}), \big(A\big(\mathsf{aff}_{t,k}(\alpha(\mathsf{x}))\big)\big)_{k=1,\dots,10\cdot 2^{\mathsf{m}_{\Theta}(t)}}\Big) \\ &= 0_{\mathbb{F}_t} \quad . \end{aligned} \tag{25}$$

– If $\alpha(\mathsf{x}) \in \widehat{V'}_t - \widehat{V}_t$, we have that $Y_{\mathbb{F}_t,H_t,\widehat{V}_t}\big(\alpha(\mathsf{x})\big) = 0_{\mathbb{F}_t}$ and $Y_{\mathbb{F}_t,H_t,\widehat{V'}_t}\big(\alpha(\mathsf{x}) + \zeta_t(\mathsf{x})\big) = 0_{\mathbb{F}_t}$, so that

$$P_t\Big(\alpha(\mathsf{x}), \Big(A\big(\mathsf{aff}_{t,\mathsf{c}_{\mathbf{N}}(t)}(\alpha(\mathsf{x}))\big)\Big)_{k=1,\dots,\mathsf{c}_{\mathbf{N}}(t)}\Big) = 0_{\mathbb{F}_t} \quad .$$

– If $\alpha(\mathsf{x}) \in \widehat{V'}_t + \zeta_t(\mathsf{x})$, we have that $Y_{\mathbb{F}_t,H_t,\widehat{V}_t}\big(\alpha(\mathsf{x})\big) = 0_{\mathbb{F}_t}$ and $Y_{\mathbb{F}_t,H_t,\widehat{V'}_t}\big(\alpha(\mathsf{x}) + \zeta_t(\mathsf{x})\big) = 1_{\mathbb{F}_t}$, so that

$$\begin{aligned} & P_t\Big(\alpha(\mathsf{x}), \Big(A\big(\mathsf{aff}_{t,\mathsf{c}_{\mathbf{N}}(t)}(\alpha(\mathsf{x}))\big)\Big)_{k=1,\dots,\mathsf{c}_{\mathbf{N}}(t)}\Big) \\ &= Z_{\mathbb{F}_t,\{0_{\mathbb{F}_t},1_{\mathbb{F}_t}\}}\big(A\big(\mathsf{aff}_{t,10\cdot 2^{\mathsf{m}_{\Theta}(t)}+1}(\alpha(\mathsf{x}))\big)\big) \\ &= 0_{\mathbb{F}_t} \quad , \end{aligned}$$

where the last equality follows from the fact that, since $\alpha(\mathsf{x}) \in \widehat{V'}_t + \zeta_t(\mathsf{x})$, $\mathsf{aff}_{t,10\cdot 2^{\mathsf{m}_{\Theta}(t)}+1}(\alpha(\mathsf{x})) = \alpha(\mathsf{x}) + \zeta_t(\mathsf{x}) \in \widehat{V'}_t$, so that $A(\alpha(\mathsf{x}) + \zeta_t(\mathsf{x})) \in \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$, and thus $Z_{\mathbb{F}_t,\{0_{\mathbb{F}_t},1_{\mathbb{F}_t}\}}(A(\alpha(\mathsf{x}) + \zeta_t(\mathsf{x}))) = 0_{\mathbb{F}_t}$.

Thus, for every $\alpha(\mathsf{x}) \in H_t$,

$$P_t\Big(\alpha(\mathsf{x}), \Big(A\big(\mathsf{aff}_{t,\mathsf{c}_{\mathbf{N}}(t)}(\alpha(\mathsf{x}))\big)\Big)_{k=1,\dots,\mathsf{c}_{\mathbf{N}}(t)}\Big) = 0_{\mathbb{F}_t} \quad . \tag{26}$$

If so, we say that the assignment polynomial A satisfies the constraint polynomial Q_t.
The reverse direction also holds.

Complexity. Recall that:

– $\mathbf{Q}$ is a $(\mathsf{S}_{\mathbf{Q}}, \mathsf{D}_{\mathbf{Q}}, \mathsf{t}_{\mathbf{Q}}, \mathsf{s}_{\mathbf{Q}})$-uniform family of polynomials;
– $\{Y_{\mathbb{F}_t,H_t,\widehat{V}_t}\}_{t\in\mathbb{N}}$ is a uniform family of polynomials where a $O(\mathsf{m}_{\mathbf{H}}(t))$-size $2^{\mathsf{m}_{\mathbf{H}}(t)}$-degree $\mathbb{F}_t$-arithmetic circuit $[Y_{\mathbb{F}_t,H_t,\widehat{V}_t}]^{\mathsf{A}}$ can be computed in time $O(\mathsf{m}_{\mathbf{H}}(t)^2)$ and space $O(\mathsf{m}_{\mathbf{H}}(t))$; similarly for the family $\{Y_{\mathbb{F}_t,H_t,\widehat{V'}_t}\}_{t\in\mathbb{N}}$; and
– $\{Z_{\mathbb{F}_t,\{0_{\mathbb{F}_t},1_{\mathbb{F}_t}\}}\}_{t\in\mathbb{N}}$ is a very simple uniform family of polynomials as $Z_{\mathbb{F}_t,\{0_{\mathbb{F}_t},1_{\mathbb{F}_t}\}}(x) = x^2 + x$.

Therefore, $\mathbf{P}$ is a $(\mathsf{S}_{\mathbf{P}}, \mathsf{D}_{\mathbf{P}}, \mathsf{t}_{\mathbf{P}}, \mathsf{s}_{\mathbf{P}})$-uniform family of polynomials with:

$$\begin{aligned} \text{size function } \mathsf{S}_{\mathbf{P}}(t) &:= \mathsf{S}_{\mathbf{Q}}(t) + O(\mathsf{m}_{\mathbf{H}}(t)) \ , \\ \text{degree function } \mathsf{D}_{\mathbf{P}}(t)[i] &:= \begin{cases} \max\{2^{\mathsf{m}_{\mathbf{H}}(t)} + \mathsf{D}_{\mathbf{Q}}(t)[i] \,,\, 2^{\mathsf{m}_{\mathbf{H}}(t)}\} & \text{if } i = 1 \\ \mathsf{D}_{\mathbf{Q}}(t)[i] & \text{if } i-1 \in \{1,\dots,\mathsf{c}_{\mathbf{N}}(t)-1\} \\ 2 & \text{if } i = \mathsf{c}_{\mathbf{N}}(t)+1 \end{cases} \ , \\ \text{time function } \mathsf{t}_{\mathbf{P}}(t) &:= \mathsf{t}_{\mathbf{Q}}(t) + O(\mathsf{m}_{\mathbf{H}}(t)^2) \ , \\ \text{space function } \mathsf{s}_{\mathbf{P}}(t) &:= \max\{\mathsf{s}_{\mathbf{Q}}(t), \mathsf{m}_{\mathbf{H}}(t), \mathsf{S}_{\mathbf{Q}}(t) + \mathsf{m}_{\mathbf{H}}(t)\} \ . \end{aligned}$$

Indeed, we can define the $\mathsf{t}_{\mathbf{P}}$-time $\mathsf{s}_{\mathbf{P}}$-space algorithm FIND$\mathbf{P}$ as follows: for every $t \in \mathbb{N}$,

$\textsc{Find}\mathbf{P}(1^t) \equiv$

(a) Compute $[Y_{\mathbb{F}_t,H_t,\widehat{V}_t}]^{\mathsf{A}} := \textsc{FindAlternator}(\mathbb{F}_t, \mathcal{B}_{H_t}, \mathsf{m}_{\widehat{\mathbf{V}}}(t))$.
(b) Compute $[Y_{\mathbb{F}_t,H_t,\widehat{V'}_t}]^{\mathsf{A}} := \textsc{FindAlternator}(\mathbb{F}_t, \mathcal{B}_{H_t}, \mathsf{m}_{\widehat{\mathbf{V}'}}(t))$.
(c) Compute $[Q_t]^{\mathsf{A}} = \textsc{Find}\mathbf{Q}(1^t)$.
(d) Compute $[Z_{\mathbb{F}_t,\widehat{C}_t}]^{\mathsf{A}} = \textsc{FindSubspPoly}(\mathbb{F}_t, \mathcal{B}_{\widehat{C}_t})$.
(e) Compute $[P_t]^{\mathsf{A}}$, using $[Y_{\mathbb{F}_t,H_t,\widehat{V}_t}]^{\mathsf{A}}$, $[Y_{\mathbb{F}_t,H_t,\widehat{V'}_t}]^{\mathsf{A}}$, $[Q_t]^{\mathsf{A}}$, and $[Z_{\mathbb{F}_t,\widehat{C}_t}]^{\mathsf{A}}$, following Equation 24.
(f) Output $[P_t]^{\mathsf{A}}$.

Clearly, $\textsc{Find}\mathbf{P}(1^t)$ correctly generates an $\mathbb{F}_t$-arithmetic circuit $[P_t]^{\mathsf{A}}$ computing P_t for all $t \in \mathbb{N}$, and has the claimed complexity parameters.

10. **Constructing Parameter 10.** Define the following two (proper) functions:

$$\text{a time function } \mathsf{t}_{\mathbf{S}} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{t}_{\mathbf{S}}(t) := \mathsf{t}_{\mathbf{W}}(t) + \mathsf{t}_{\mathbf{\Phi}}(t) = \mathsf{t}_{\mathbf{W}}(t) + \mathsf{t}_{\textsc{Prim}}(\ell(t)) + \ell(t)^2 \log \ell(t) + f(t) \enspace ,$$
$$\text{a space function } \mathsf{s}_{\mathbf{S}} \colon \mathbb{N} \to \mathbb{N}\colon\ \mathsf{s}_{\mathbf{S}}(t) := \max\{\mathsf{s}_{\mathbf{W}}(t), \mathsf{s}_{\mathbf{\Phi}}(t)\} = \max\{\mathsf{s}_{\mathbf{W}}(t), \mathsf{s}_{\textsc{Prim}}(\ell(t)) + f(t)\} \enspace .$$

Recall that $\mathsf{t}_{\mathbf{W}}$ and $\mathsf{s}_{\mathbf{W}}$ are functions from Parameter 11 in Definition 8.1; also, $\mathsf{t}_{\mathbf{\Phi}}$ and $\mathsf{s}_{\mathbf{\Phi}}$ are functions from Lemma 11.12 invoked with the three positive integers $f(t)$, $\kappa(t)$, and $\ell(t)$.

11. **Constructing Parameter 11.** Define the family $\mathbf{S} = \{S_t\}_{t\in\mathbb{N}}$ by

$$S_t := \bigcup_{k-2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}=s_0(t),\dots,s_1(t)} \left(\Phi_{f(t),\kappa(t),\ell(t)}(W_t) + \theta^{(k)}\right) \enspace . \tag{27}$$

Recall that $W_t \in \mathbf{W}$, and $\mathbf{W}$ is the family from Parameter 12 in Definition 8.1. Also, W_t is a subset of the vertex set V_t of $G_t = \mathrm{DDB}(\kappa(t), 2^{\ell(t)})$, so that we can indeed apply the injection $\Phi_{f(t),\kappa(t),\ell(t)} \colon V_t \to \mathbb{F}_t$ to vertices in W_t, and, moreover, there exists a $\mathsf{t}_{\mathbf{W}}$-time $\mathsf{s}_{\mathbf{W}}$-space algorithm $\textsc{Find}\mathbf{W}$ that, for every positive integer t and every $i \in \{1,\dots,|W_t|\}$, on input $(1^t, i)$, computes the i-th vertex in W_t (under some canonical ordering of W_t).

Hence, $|S_t| = \mathsf{c}_{\mathbf{C}}(t)|W_t|$ and, as required, $S_t \subseteq H_t$; we can define the algorithm $\textsc{Find}\mathbf{S}$ as follows: for every positive integer t,

$\textsc{Find}\mathbf{S}(1^t, i) \equiv$

(a) Parse i as $i_1\mathsf{c}_{\mathbf{C}}(t) + i_2$ with $i_2 \in \{s_0(t),\dots,s_1(t)\}$.
(b) Compute $v_{i_1} := \textsc{Find}\mathbf{W}(1^t, i_1)$.
(c) Compute $\alpha_{i_1}(\mathsf{x}) := \textsc{Comp}\mathbf{\Phi}(1^{f(t)}, 1^{\kappa(t)}, 1^{\ell(t)}, v_{i_1})$.
(d) For $\ell = 1,\dots,\mathsf{m}_{\mathbf{\Theta}}(t)$, compute the element $\theta_{t,\ell}(\mathsf{x}) := \textsc{Find}\mathbf{\Theta}(1^t, \ell)$.
(e) Letting $\sigma_1 \cdots \sigma_{\mathsf{m}_{\mathbf{\Theta}}(t)} := \mathrm{bin}(i_2)$, compute $\theta^{(2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}+i_2+1)} = \sum_{\ell=1}^{\mathsf{m}_{\mathbf{\Theta}}(t)} \sigma_\ell \theta_{t,\ell}(\mathsf{x})$.
(f) Output $\alpha_{i_1}(\mathsf{x}) + \theta^{(^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t)\rceil}+i_2+1)}$.

Thus, the algorithm $\textsc{Find}\mathbf{S}$ is a $\mathsf{t}_{\mathbf{S}}$-time $\mathsf{s}_{\mathbf{S}}$-space algorithm, where:

$$\mathsf{t}_{\mathbf{S}}(t) := \mathsf{t}_{\mathbf{W}}(t) + \mathsf{t}_{\mathbf{\Phi}}(t) \enspace ,$$
$$\mathsf{s}_{\mathbf{S}}(t) := \max\{\mathsf{s}_{\mathbf{W}}(t), \mathsf{s}_{\mathbf{\Phi}}(t)\} \enspace ,$$

matching the time and space complexities defined previously.

The construction of the parameters for $\textsc{Succinct}$ACSP is now complete.

11.1.3 The Levin reduction for double extended De Bruijn graphs

We show that the parameter conversion discussed in Section 11.1.2 yields a Levin reduction (according to Definition 11.1) from $\textsc{Succinct}$GCP to $\textsc{Succinct}$ACSP with respect to the class of parameters considered in Construction 11.15.

More precisely, consider the following definitions:

- Define $\mathcal{P}_{\text{DDB}}$ to be the class of parameters for SuccinctGCP considered by Construction 11.15, i.e., those choices of parameters for which, for some proper functions $\kappa, \ell \colon \mathbb{N} \to \mathbb{N}$, for all $t \in \mathbb{N}$, $\alpha_{\mathbf{G}}(t) = 3$ and $G_t = \text{DDB}(\kappa(t), 2^{\ell(t)} - 1)$.
- Define $F_{\mathsf{p}} \colon \{0,1\}^* \to \{0,1\}^*$ to be the function that, on input a parameter choice $\mathsf{par}_{\text{sGCP}} \in \mathcal{P}_{\text{DDB}}$, performs the parameter conversion described in Construction 11.15. More precisely, F_{p} works as follows:

$$\mathsf{par}_{\text{sGCP}} = \begin{pmatrix} \alpha_{\mathbf{G}}, \\ \mathsf{c}_{\mathbf{C}}, \\ \mathsf{c}_{\mathbf{T}}, \\ (\mathsf{S}_{\mathbf{M}}, \mathsf{D}_{\mathbf{M}}, \mathsf{t}_{\mathbf{M}}, \mathsf{s}_{\mathbf{M}}, \text{Find}\mathbf{M}), \\ (\mathsf{S}_{\mathbf{K}}, \mathsf{D}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathsf{s}_{\mathbf{K}}, \text{Find}\mathbf{K}), \\ (\mathsf{t}_{\mathbf{W}}, \mathsf{s}_{\mathbf{W}}, \text{Find}\mathbf{W}), \\ (\mathsf{t}_{\mathbf{F}}, \mathsf{s}_{\mathbf{F}}, \text{Comp}\mathbf{F}) \end{pmatrix} \overset{F_{\mathsf{p}}}{\longmapsto} \mathsf{par}_{\text{sACSP}} = \begin{pmatrix} f, \\ (\mathsf{m}_{\mathbf{H}}, \mathsf{t}_{\mathbf{H}}, \mathsf{s}_{\mathbf{H}}, \text{Find}\mathbf{H}), \\ (\mathsf{c}_{\mathbf{N}}, \mathsf{t}_{\mathbf{N}}, \mathsf{s}_{\mathbf{N}}, \text{Find}\mathbf{N}), \\ (\mathsf{t}_{\mathbf{D}}, \mathsf{s}_{\mathbf{D}}, \text{Comp}\mathbf{D}), \\ (\mathsf{t}_{\mathbf{P}}, \mathsf{s}_{\mathbf{P}}, \mathsf{S}_{\mathbf{P}}, \mathsf{D}_{\mathbf{P}}, \text{Find}\mathbf{P}), \\ (\mathsf{t}_{\mathbf{S}}, \mathsf{s}_{\mathbf{S}}, \text{Find}\mathbf{S}), \end{pmatrix},$$

  where the mapping is done by following the definitions of the various new complexity functions and algorithms (for SuccinctACSP) based on the old ones (for SuccinctGCP).
- Define $F_{\mathsf{w}} \colon \{0,1\}^* \to \{0,1\}^*$ as follows: for every $\mathsf{par}_{\text{sGCP}} \in \mathcal{P}_{\text{DDB}}$, $t \in \mathbb{N}$ and $C \colon V_t \to C_t$,

  $F_{\mathsf{w}}(\mathsf{par}_{\text{sGCP}}, 1^{2^t}, C) \equiv$
  1. Define C'' based on C' (Equation 17), in turn based on C (Equation 15).
  2. Define the function $\widetilde{A} \colon \widehat{V'}_t \to \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$ by $\widetilde{A}\big(\alpha(\mathsf{x})\big) := \Big(C'' \circ \Phi^{-1}_{\widehat{V'}_t}\Big)\big(\alpha(\mathsf{x})\big)$ for all $\alpha(\mathsf{x}) \in \widehat{V'}_t$. (See Equation 18.)
  3. Define the polynomial $A \colon \mathbb{F}_t \to \mathbb{F}_t$ as the low-degree extension of $\widetilde{A}$ in $\mathbb{F}_t$, i.e., $A := \text{LDE}_{\mathbb{F}_t, 1, \widehat{V}_t}(\widetilde{A})$. (See Equation 19.)
  4. Output A.

We prove the following theorem:

Theorem 11.16. *The pair of functions $(F_{\mathsf{p}}, F_{\mathsf{w}})$ is a Levin reduction from* SuccinctGCP *to* SuccinctACSP *with respect to $\mathcal{P}_{\text{DDB}}$.*

We divide the proof of Theorem 11.16 into three claims:
- in Claim 11.17, we explain why both F_{p} and F_{w} are polynomial-time computable;
- in Claim 11.18, we show the "completeness" and "soundness" of F_{p}; and
- in Claim 11.19, we show that F_{w} produces good witnesses.

Claim 11.17. *The functions F_{p} and F_{w} are polynomial-time computable.*

Proof. The efficiency of F_{p} easily follows by inspection of how the parameters are converted in Construction 11.15. (Essentially, all the new functions and algorithms are "easy combinations" of previous functions and algorithms, and thus not hard to write down.) The efficiency of F_{w} easily follows from the fact that it can run in time that is polynomial in 1^{2^t}, which is plenty of time for computing the low-degree extension of $\widetilde{A}$ based on the input coloring C. □

Claim 11.18. *For every choice of parameters $\mathsf{par}_{\text{sGCP}} \in \mathcal{P}_{\text{DDB}}$ and for every instance $(x, 1^t) \in \{0,1\}^*$, $(x, 1^t) \in$* SuccinctGCP$(\mathsf{par}_{\text{sGCP}})$ *if and only if $(x, 1^t) \in$* SuccinctACSP$(F_{\mathsf{p}}(\mathsf{par}_{\text{sGCP}}))$.

Proof. Fix a choice of parameters $\mathsf{par}_{\text{sGCP}} \in \mathcal{P}_{\text{DDB}}$ for SuccinctGCP and an instance $(x, 1^t) \in \{0,1\}^*$.

Completeness. First, we prove the "completeness" direction of the statement: we need to prove that if $(x, 1^t) \in$ SuccinctGCP$(\mathsf{par}_{\text{sGCP}})$ then $(x, 1^t) \in$ SuccinctACSP$(F_{\mathsf{p}}(\mathsf{par}_{\text{sGCP}}))$. So suppose that $(x, 1^t) \in$ SuccinctGCP, and let $C \colon V_t \to C_t$ be a coloring that witnesses this fact. Define $A := F_{\mathsf{w}}(\mathsf{par}_{\text{sGCP}}, 1^{2^t}, C)$. We now argue that A is a witness for $(x, 1^t) \in$ SuccinctACSP$(F_{\mathsf{p}}(\mathsf{par}_{\text{sGCP}}))$.

Following the definition of SuccinctACSP (Definition 10.1), there are two requirements to satisfy:

- *Satisfiability of constraints.* We need to show that the assignment polynomial A satisfies the constraint polynomial P_t, i.e.,

$$\forall\, \alpha(\mathsf{x}) \in H_t\,,\ P_t\Big(\alpha(\mathsf{x}), A\big(\mathsf{aff}_{t,1}(\alpha(\mathsf{x}))\big), \ldots, A\big(\mathsf{aff}_{t,\mathsf{c_N}(t)}(\alpha(\mathsf{x}))\big)\Big) = 0_{\mathbb{F}_t} \ .$$

  And, indeed, since $C\colon V_t \to C_t$ satisfies the coloring constraints induced by K_t and M_t,
  - C satisfies the coloring constraints induced by U_t (see Equation 12), so that
  - C satisfies the coloring constraints induced by A_t (see Equation 14), so that
  - C satisfies the coloring constraints induced by R_t (see Equation 21), so that
  - A satisfies the constraints polynomial Q_t (see Equation 23), so that
  - A satisfies the constraint polynomial P_t (see Equation 26),

  as desired.

- *Consistency with the instance.* We need to show that the assignment polynomial A is consistent with the instance $(x, 1^t)$, i.e.,

$$x = \mathrm{bit}\big(A(\alpha_1(\mathsf{x}))\big) \cdots \mathrm{bit}\big(A(\alpha_{|x|}(\mathsf{x}))\big) \ ,$$

  where $\alpha_i(\mathsf{x})$ is the i-th element in S_t. And, indeed, since we know that $F_t\big(x, \big(C(v_i)\big)_{i=1}^{|x|}\big) = 0$, where v_i is the i-th element in W_t, and we also know by assumption that $F_t(x, c_1 \cdots c_{|x|})$ is the test $x \stackrel{?}{=} c'_1 \cdots c'_{|x|}$ where each c'_i is the substring of c_i from bit $s_0(t)$ to bit $s_1(t)$, by our definition of A the above equation holds: essentially, we have "distributed" the bit of each color of a vertex v_i among various field elements, and all of these field elements that are relevant for the test are accounted for in the definition of S_t (see Equation 27).

This concludes the proof of the "completeness" direction of the statement.

Soundness. Next, we prove the "soundness" direction of the statement: we need to prove that if $(x, 1^t) \in$ SuccinctACSP$(F_{\mathsf{p}}(\mathsf{par}_{\mathrm{sGCP}}))$ then $(x, 1^t) \in$ SuccinctGCP$(\mathsf{par}_{\mathrm{sGCP}})$. So suppose instead that we have $(x, 1^t) \in$ SuccinctACSP$(F_{\mathsf{p}}(\mathsf{par}_{\mathrm{sGCP}}))$, and let $A\colon \mathbb{F}_t \to \mathbb{F}_t$ be a polynomial that witnesses this fact. Consider $A|_{\widehat{V}_t}$, the restriction of A to $\widehat{V}_t$. We argue that $A|_{\widehat{V}_t}$ takes on values in $\{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$. Indeed, assume by way of contradiction that there exists $\alpha(\mathsf{x}) \in \widehat{V}_t$ such that $A(\alpha(\mathsf{x})) \notin \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$. Let $\alpha'(\mathsf{x}) := \mathsf{aff}_{t,\mathsf{c_N}(t)}(\alpha(\mathsf{x})) = \alpha(\mathsf{x}) + \zeta_t(\mathsf{x})$, and note that $\alpha'(\mathsf{x}) \in \widehat{V}_t + \zeta_t(\mathsf{x})$. By setting $\big(x_0, x_1, \ldots, x_{\mathsf{c_N}(t)}\big) := \big(\alpha(\mathsf{x}), A(\mathsf{aff}_{t,1}(\alpha(\mathsf{x}))), \ldots, A(\mathsf{aff}_{t,\mathsf{c_N}(t)}(\alpha(\mathsf{x})))\big)$, this means that the second summand in Equation 24 does not vanish; since the first summand does vanish, we reach a contradiction to the fact that A is a witness to $(x, 1^t) \in$ SuccinctACSP (because it does not satisfy the constraint polynomial P_t). We conclude that $A|_{\widehat{V}_t}$ takes on values in $\{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$, as claimed. We can now define the function $C\colon V_t \to C_t$ by

$$\forall\, v \in V_t\,,\ C(v) := \Big(A\big(\Phi_{f(t),\kappa(t),\ell(t)}(v) + \theta^{(k)}\big)\Big)_{k - 2^{\lceil \log \mathsf{m}_{\widehat{\mathbf{V}}}(t) \rceil} = 1, \ldots, \mathsf{c_C}(t)} \ . \tag{28}$$

It is easy to see that C is a witness for $(x, 1^t) \in$ SuccinctGCP$(\mathsf{par}_{\mathrm{sGCP}})$ by inspection of the construction. This concludes the proof of the "soundness" direction of the statement. As both directions have now been shown, the proof is now complete. □

Claim 11.19. *For every choice of parameters $\mathsf{par}_{\mathrm{sGCP}} \in \mathcal{P}_{\mathrm{DDB}}$ and for every instance $(x, 1^t) \in \{0,1\}^*$, if C is a witness to "$(x, 1^t) \in$ SuccinctGCP$(\mathsf{par}_{\mathrm{sGCP}})$" then $F_{\mathsf{w}}(1^t, C)$ is a witness to "$(x, 1^t) \in$ SuccinctACSP$(F_{\mathsf{p}}(\mathsf{par}_{\mathrm{sGCP}}))$".*

Proof. The claim follows immediately from the fact that in the proof of the "completeness" direction of the statement from Claim 11.18, we have constructed, starting from a valid coloring C and according to F_{p}, a valid assignment polynomial A for $(x, 1^t)$: by first considering the function $\widetilde{A}$, and then takings its low-degree extension over $\widehat{V}_t$ in $\mathbb{F}_t$. □

A Other Related Work

Choices of abstract machines for verifier-efficient PCPs. Babai et al. [BFLS91] originally suggested a choice of *pointer machine* as the abstract machine for which to construct their transparent proof system; specifically, they suggested using Kolmogorov–Uspenskiĭ machines [Kol53][KU58] (in a more general form following the storage modification machines of Schönhage [Sch80]).[12] Most pointer machines (including Kolmogorov-Uspenskiĭ machines) and random-access machines are equivalent up to polylogarithmic factors; nonetheless, we choose to work with random-access machines because they are more natural in light of today's computer architectures.

Ben-Sasson et al. [BSGH+05] use *Turing machines*. While non-determinism makes Turing machines and random-access machines equivalent up to polylogarithmic factors,[13] random-access machines are not believed to be simulatable within polylogarithmic factors on (even multi-tape) Turing machines. Either way, it is clear that Turing machines are too "clumsy" to program.

Routing as a method to "structure" computation. Previous work studying PCPs (especially when studying short PCPs) used *routing techniques* to generate "highly-structured" constraint satisfaction problems from the constraint satisfaction problems induced by computations on abstract machines (e.g., ensuring that the computation of a Turing machine or a pointer machine was carried out correctly).

Such techniques seem to necessary when one wishes the cost of the reduction to be only quasilinear (and usually there are trivial reductions with quadratic cost).

Ofman [Ofm65] first used *generalized connection networks* for simulation purposes. Babai et al. [BFLS91] observed that such techniques could be used to construct a reduction from the problem of verifying correct computation on Kolmogorov–Uspenskiĭ machines to a coloring problem on sorting networks (a problem which could be then easily arithmetized and for which they show how to construct transparent proofs).

Polishchuk and Spielman [PS94], as part of their work obtaining almost quasilinear-size PCPs, show a quasilinear reduction from circuit satisfiability to a coloring problem over De Bruijn graphs, which are well-known to be able to route any permutation. De Bruijn graphs were again used in [BSGH+04] and in [BSGH+05]. (In fact, [BSGH+05] also relied on [PF79], which can also be viewed as a routing technique.)

Robson [Rob91] constructed a fast reduction from random-access machines to Boolean formulas, but, since his work was not in the context of PCPs, he did not face the additional constraints imposed by the need to eventually carefully arithmetize the constraint satisfaction problem; furthermore, his reduction hides large constants.

We also use routing techniques to generate highly-structured constraint satisfaction problems, in our case, starting from constraint satisfaction problems induced by computations on random-access machines. (In fact, in one of our reductions, we show how to forego routing altogether at the expense of making computational assumptions.) As already discussed, we shall also make explicit and optimize the necessary routing algorithms, and suggest how routing techniques may be useful in the larger context of generating Boolean circuits for computations of interest.

Finally, we note that oblivious random-access machines [GO96] seem to not help in our setting, as the "structuring" offered by masking access patterns is of the probabilistic kind, rather than deterministic. (Unlike the analogous result for Turing machines by [PF79], which was indeed used in [BSGH+05].)

[12] See Ben-Amram [BA95] for a discussion of several different models of pointer machines; at high level, such machines have storage that is in the form of a labeled graph whose topology changes as the computation progresses. See Gurevich [Gur88] for a more extended discussion about Kolmogorov–Uspenskiĭ machines.

[13] In fact, non-determinism makes simulation between very different abstract machines very efficient: Gurevich and Shelah prove a robustness theorem [GS89, Theorem 1] for the class of non-deterministic quasilinear time. See also follow-up work by [NRS94].

B Routing on De Bruijn Graphs

We describe an algorithm for routing a given permutation over "three and a half" De Bruijn graphs connected in tandem; this will imply, in particular, a proof of Claim 6.7. While the routing properties of De Bruijn graphs are folklore, we have not been able to find explicit algorithms in the literature for routing, so, given that we are interested in an explicit implementation, we devote this section to deduce an explicit routing algorithm.

B.1 Butterfly Networks and Isomorphic Graphs of Interest

We begin by introducing a fundamental family of graphs studied in parallel systems: butterfly networks.

Definition B.1. *Let κ be a positive integer. The κ-dimensional* **butterfly network***, denoted* BN_κ*, is a directed graph consisting of $\kappa+1$ "columns" each containing 2^κ vertices identified with κ-bit strings. A vertex v in layer $i \in \{0, \ldots, \kappa-1\}$ with identifier $w \in \{0,1\}^\kappa$ has two neighbors in layer $i+1$ with identifier w and $w \oplus e_{i+1}$. (See Figure 3a for an example.)*

A *reversed* butterfly network is, as the name suggests, obtained by reversing the direction of the edges in a butterfly network:

Definition B.2. *Let κ be a positive integer. The κ-dimensional* **reversed butterfly network***, denoted* $\mathrm{BN}^{\mathrm{r}}_\kappa$*, is a directed graph consisting of $\kappa+1$ "columns" each containing 2^κ vertices identified with κ-bit strings. A vertex v in layer $i \in \{0, \ldots, \kappa-1\}$ with identifier $w \in \{0,1\}^\kappa$ has two neighbors in layer $i+1$ with identifier w and $w \oplus e_{\kappa-i}$. (See Figure 3b for an example.)*

The reversed butterfly network $\mathrm{BN}^{\mathrm{r}}_\kappa$ is in fact isomorphic to the butterfly network BN_κ via a graph isomorphism permuting the rows by reversing the bits of a row's identifier.

Claim B.3. *Let κ be a positive integer. The map $\phi_\kappa \colon \{0, \ldots, \kappa\} \times \{0,1\}^\kappa \to \{0, \ldots, \kappa\} \times \{0,1\}^\kappa$ defined by $\phi_\kappa(i, w) = (i, w^{\mathrm{r}})$ is a graph isomorphism from $\mathrm{BN}^{\mathrm{r}}_\kappa$ to BN_κ.*

Note that indeed the graph isomorphism ϕ_κ is "row-rigid" in the sense that it only "shuffles" the rows of the reversed butterfly network, by mapping a row with identifier w to the row w^{r}.

Proof. Let $a = ((i, w), (i+1, w'))$ be an edge in $\mathrm{BN}^{\mathrm{r}}_\kappa$. Then:

- If $w' = w$, then $\phi_\kappa(a) = ((i, w^{\mathrm{r}}), (i+1, w^{\mathrm{r}}))$ is an edge in BN_κ.
- If $w' = w \oplus e_{\kappa-i}$, then $\phi_\kappa(a) = ((i, w^{\mathrm{r}}), (i+1, (w \oplus e_{\kappa-i})^{\mathrm{r}})) = ((i, w^{\mathrm{r}}), (i+1, w^{\mathrm{r}} \oplus e_{i+1}))$ is an edge in BN_κ.

Conversely, let $b = ((i, w), (i+1, w'))$ be an edge in BN_κ. Then:

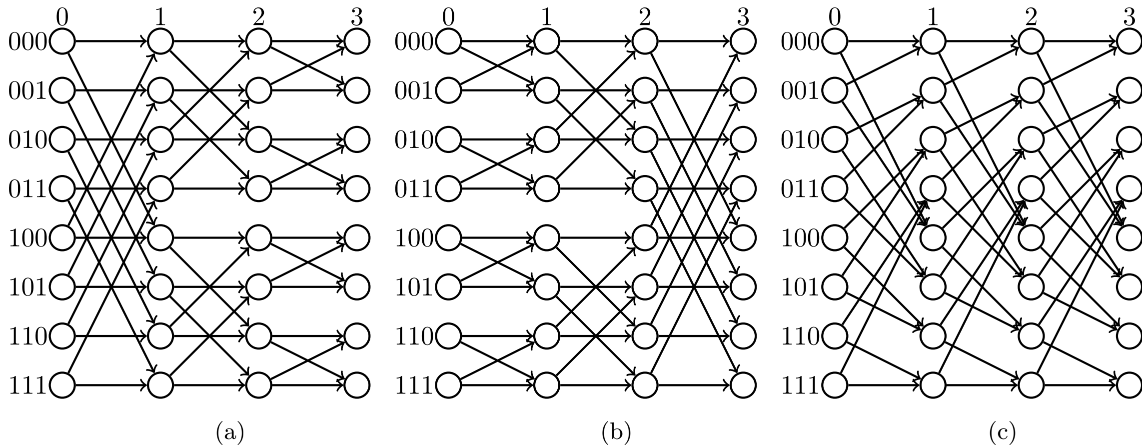


Figure 3: Three-dimensional (a) butterfly network, (b) reversed butterfly network, and (c) De Bruijn graph.

- If $w' = w$, then $\phi_\kappa^{-1}(b) = ((i, w^{\mathrm{r}}), (i+1, w^{\mathrm{r}}))$ is an edge in $\mathrm{BN}_\kappa^{\mathrm{r}}$.
- If $w' = w \oplus e_{i+1}$, then $\phi_\kappa^{-1}(b) = ((i, w^{\mathrm{r}}), (i+1, (w \oplus e_{i+1})^{\mathrm{r}})) = ((i, w^{\mathrm{r}}), (i+1, w^{\mathrm{r}} \oplus e_{\kappa-i}))$ is an edge in $\mathrm{BN}_\kappa^{\mathrm{r}}$.

As there are the same number of edges in both $\mathrm{BN}_\kappa^{\mathrm{r}}$ and BN_κ, the proof of the claim is complete. □

We shall also be interested in De Bruijn graphs.

Definition B.4. *Let κ be a positive integer. The κ-dimensional* **De Bruijn graph***, denoted DB_κ, is a directed graph consisting of $\kappa+1$ "columns" each containing 2^κ vertices identified with κ-bit strings. A vertex v in layer $i \in \{0, \ldots, \kappa-1\}$ with identifier $w \in \{0,1\}^\kappa$ has two neighbors in layer $i+1$ with identifier $\mathrm{sr}(w)$ and $\mathrm{sr}(w) \oplus e_1$, where sr denotes the cyclic "shift right" bit operation. (See Figure 3c for an example.)*

A De Bruijn graph DB_κ is also isomorphic to the butterfly network BN_κ via a graph isomorphism that cyclically shifts the bits of the identifier of a vertex depending on the index of the column in which the vertex lies.

Claim B.5. *Let κ be a positive integer. The map $\psi_\kappa \colon \{0, \ldots, \kappa\} \times \{0,1\}^\kappa \to \{0, \ldots, \kappa\} \times \{0,1\}^\kappa$ defined by $\psi_\kappa(i, w) = (i, \mathrm{sr}^{i-1}(w^{\mathrm{r}}))$ is a graph isomorphism from BN_κ to DB_κ.*

Note that, unlike the graph isomorphism from a reversed butterfly network to a butterfly network, the graph isomorphism from a butterfly network to a De Bruijn graph does not "mess up" the order of vertices in the first and last columns; this fact is important later in this section because it implies that the graph isomorphism easily extends to when networks are connected in tandem.

Proof. Let $a = ((i, w), (i+1, w'))$ be an edge in BN_κ. Then:

- If $w' = w$, then $\psi_\kappa(a) = ((i, \mathrm{sr}^{i-1}(w)^{\mathrm{r}}), (i+1, \mathrm{sr}^{i}(w^{\mathrm{r}})))$ is an edge in DB_κ.
- If $w' = w \oplus e_{i+1}$, then $\psi_\kappa(a) = ((i, \mathrm{sr}^{i-1}(w^{\mathrm{r}})), (i+1, \mathrm{sr}^{i}((w \oplus e_{i+1})^{\mathrm{r}}))) = ((i, \mathrm{sr}^{i-1}(w^{\mathrm{r}})), (i+1, \mathrm{sr}^{i}(w^{\mathrm{r}}) \oplus e_1))$ is an edge in DB_κ.

Conversely, let $b = ((i, w), (i+1, w'))$ be an edge in DB_κ. Then:

- If $w' = \mathrm{sr}(w)$, then $\phi_\kappa^{-1}(b) = ((i, \mathrm{sl}^{i-1}(w)^{\mathrm{r}}, (i+1, \mathrm{sl}^{i}(\mathrm{sr}(w))^{\mathrm{r}})) = ((i, \mathrm{sl}^{i-1}(w)^{\mathrm{r}}, (i+1, \mathrm{sl}^{i-1}(w)^{\mathrm{r}}))$ is an edge in BN_κ.
- If $w' = \mathrm{sr}(w) \oplus e_1$, then $\phi_\kappa^{-1}(b) = ((i, \mathrm{sl}^{i-1}(w)^{\mathrm{r}}, (i+1, \mathrm{sl}^{i}(\mathrm{sr}(w) \oplus e_1)^{\mathrm{r}})) = ((i, \mathrm{sl}^{i-1}(w)^{\mathrm{r}}, (i+1, \mathrm{sl}^{i-1}(w)^{\mathrm{r}} \oplus e_{i+1}))$ is an edge in BN_κ.

As there are the same number of edges in both BN_κ and DB_κ, the proof of the claim is complete. □

B.2 Beneš Networks and Their Rearrangeability

A Beneš network is a routing network that is able to route *any* permutation; this property is known as *rearrangeability.*

A Beneš network is the "concatenation" of a butterfly network and a reversed butterfly network:

Definition B.6. *Let κ be a positive integer. The κ-dimensional* Beneš network*, denoted $\textsc{Beneš}_\kappa$, is a κ-dimensional butterfly network and a κ-dimensional reversed butterfly network connected in tandem. More precisely, $\textsc{Beneš}_\kappa$ is a directed graph with $2\kappa+1$ "columns" numbered $0, \ldots, 2\kappa$, each containing 2^κ vertices identified with κ-bit strings; a vertex $v = (i, w)$ in layer $i \in \{0, 1, \ldots, 2\kappa-1\}$ with identifier $w \in \{0,1\}^\kappa$ has two neighbors in layer $i+1$ with identifiers w and $w \oplus a_i$ respectively, where $a_i \in \{0,1\}^\kappa$ is equal to e_{i+1} if $i \in \{0, \ldots, \kappa-1\}$ and is equal to $e_{2\kappa-i}$ if $i \in \{\kappa, \ldots, 2\kappa-1\}$.*

More concretely, an κ-dimensional Beneš network can be used to route either $2^{\kappa+1}$ packets using edge disjoint paths (where each vertex in the network receives and sends two packets) [Lei92, Theorem 3.10] or to route 2^κ packets using vertex disjoint paths (where each vertex in the network receives and sends exactly one packet) [Lei92, Theorem 3.11].

As we are interested in the latter form of routing, we recall explicitly the theorem and its constructive proof:

Theorem B.7 ([Lei92, Theorem 3.11])**.** *Let κ be a positive integer and $\pi\colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ a permutation. There exists a set S_π of 2^κ vertex-disjoint paths such that each vertex $(0,w)$ in $\textsc{Beneš}_\kappa$ is connected to $(2\kappa+1, \pi(w))$. Moreover, S_π can be found in $O(\kappa \cdot 2^\kappa)$ time and space.*

Proof. The existence of the "routing" S_π will follow from the correctness of the algorithm that we present, which will always return a valid solution.

First let us describe the idea at high level. The idea is to use the recursive structure of a Beneš network; indeed, note that, by removing the "leftmost" and "rightmost" layers of a Beneš network (i.e., layer 0 and layer 2κ), we obtain two $(\kappa-1)$-dimensional Beneš networks — a "top" one and a "bottom" one. We can recursively solve any routing problem on the smaller Beneš networks, and thus we are left to reduce the routing on the original network to use the routing on the smaller sub-networks. (And, of course, the base case with $\kappa = 1$ is trivial to solve.)

A fast algorithm immediately follows from the above intuition. Indeed, consider the algorithm $\textsc{BenešRoute}$ that, on input a positive integer κ and $\pi\colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ (specified as a table), is defined as follows:

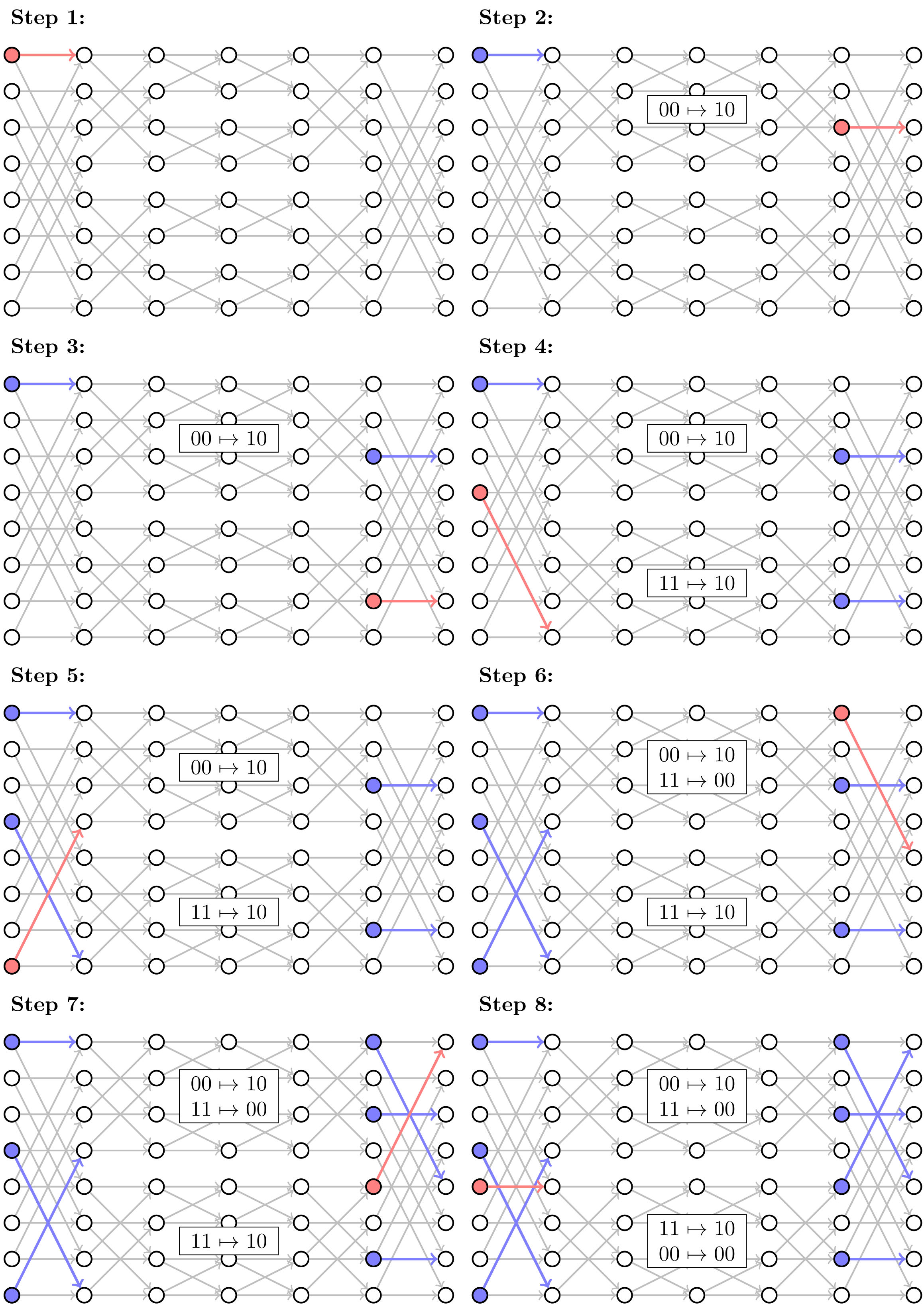
$\textsc{BenešRoute}(\kappa, \pi) \equiv$

1. If $\kappa = 1$, then:
   (a) If $\pi(0) = 0$ and $\pi(1) = 1$ then route both $(0,0)$ and $(0,1)$ using identity edges.
   (b) Otherwise, if $\pi(0) = 1$ and $\pi(1) = 0$ then route both $(0,0)$ and $(0,1)$ using cross edges.
2. If $\kappa > 1$, then repeat the following until all vertices in the column 0 are routed:
   (a) Choose vertex $u = (0,w)$ that is not routed ant let $(2\kappa, w')$ where $w' = \pi(w)$.
   (b) Route u using the upper sub-network:
      i. "Forward" route u to $(1,w)$ if $w_1 = 0$ and to $(1, w \oplus e_1)$ otherwise.
      ii. "Backward" route $(2\kappa, w')$ to $(2\kappa-1, w')$ if $w'_1 = 0$ and to $(2\kappa-1, w' \oplus e_1)$ otherwise.
   (c) Set $\pi'(\widehat{w})$ to be $\widehat{w}'$ where $\widehat{w}$ and $\widehat{w}'$ are the least significant $\kappa-1$ bits of w and w' respectively.
   (d) Route $(2\kappa, w' \oplus e_1)$ using the lower sub-network: Let $(0, w'')$ be the source of $(2\kappa, w')$ where $w'' = \pi^{-1}(w')$.
      i. Route $(2\kappa, w')$ to $(2\kappa-1, w')$ if $w'_1 = 1$ and to $(2\kappa, w' \oplus e_1)$ otherwise.
      ii. Route $(0, w'')$ to $(1, w'')$ if $w''_1 = 1$ and to $(1, w'' \oplus e_1)$ otherwise.
   (e) Set $\pi''^{-1}(\widehat{w}')$ to be $\widehat{w}''$ where $\widehat{w}'$ and $\widehat{w}''$ are the least significant $\kappa-1$ bits of w' and w'' respectively.
   (f) Set $u = (0, w'' \oplus e_1)$.
   (g) If u is routed then goto Step 2 Otherwise goto Step 2b.
3. Run $\textsc{BenešRoute}(\kappa-1, \pi')$ recursively on the upper sub-network.
4. Run $\textsc{BenešRoute}(\kappa-1, \pi'')$ recursively on the lower sub-network.

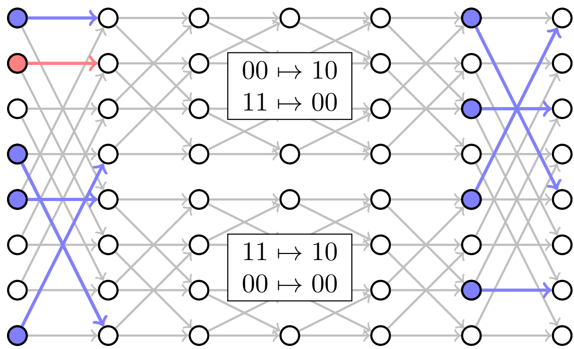
The correctness of $\textsc{BenešRoute}$ easily follows from an induction argument. Furthermore, since the algorithm visits every vertex in $\textsc{Beneš}_\kappa$ a constant number of times and $\textsc{Beneš}_\kappa$ has $(\kappa+1)\cdot 2^\kappa$ vertices, we deduce that the space and time complexities are $O(\kappa \cdot 2^\kappa)$. □

Example B.1. We give a pictorial example for the computation of $\textsc{BenešRoute}(3, \pi)$ when π is given by the following permutation:

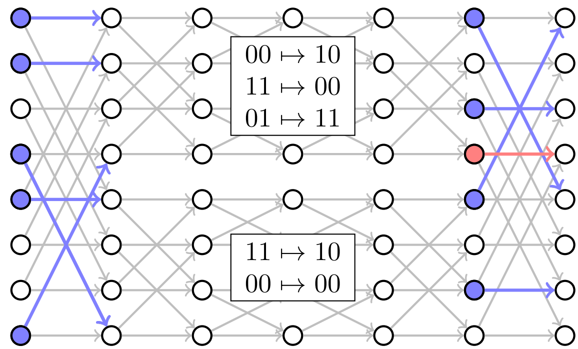
$$000 \mapsto 010\,,\ \ 001 \mapsto 011\,,\ \ 010 \mapsto 101\,,\ \ 011 \mapsto 110\,,\ \ 100 \mapsto 000\,,\ \ 101 \mapsto 001\,,\ \ 110 \mapsto 111\,,\ \ 111 \mapsto 100\ \ .$$


Step 1:
Step 2:
00 ↦ 10
Step 3:
00 ↦ 10
Step 4:
00 ↦ 10
11 ↦ 10
Step 5:
00 ↦ 10
11 ↦ 10
Step 6:
00 ↦ 10
11 ↦ 00
11 ↦ 10
Step 7:
00 ↦ 10
11 ↦ 00
11 ↦ 10
Step 8:
00 ↦ 10
11 ↦ 00
11 ↦ 10
00 ↦ 00

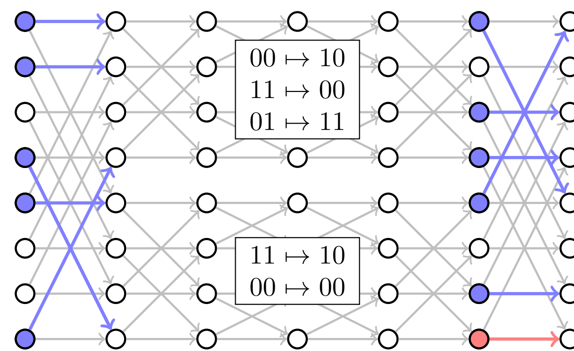
Step 9:



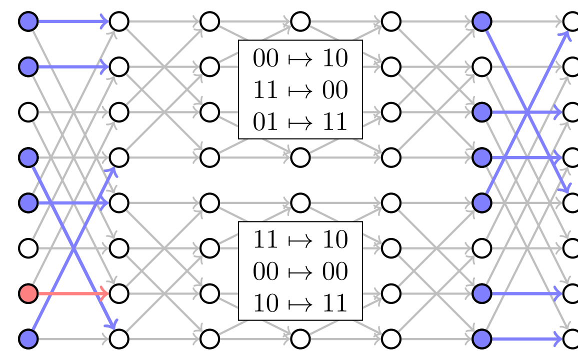
Step 10:



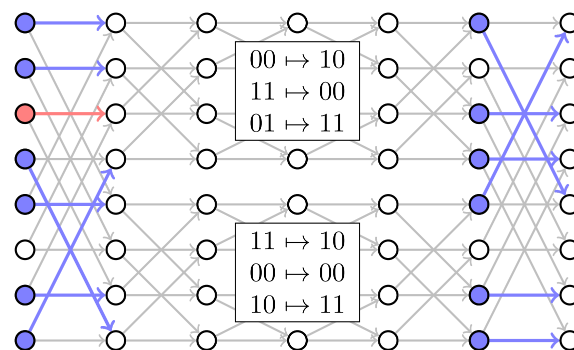
Step 11:



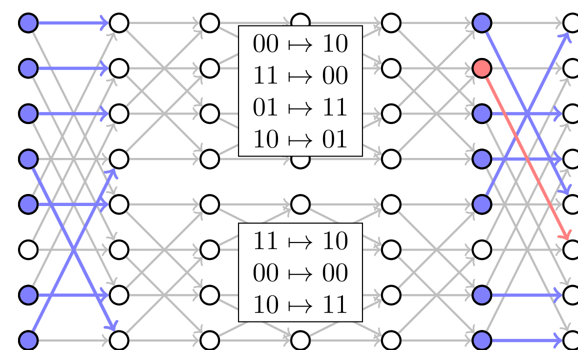
Step 12:



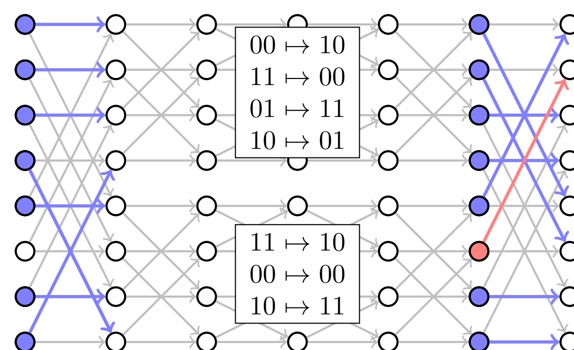
Step 13:



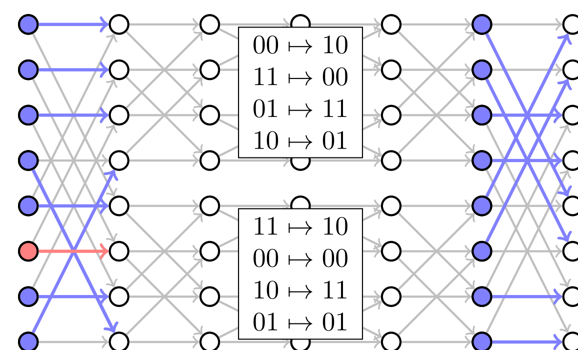
Step 14:



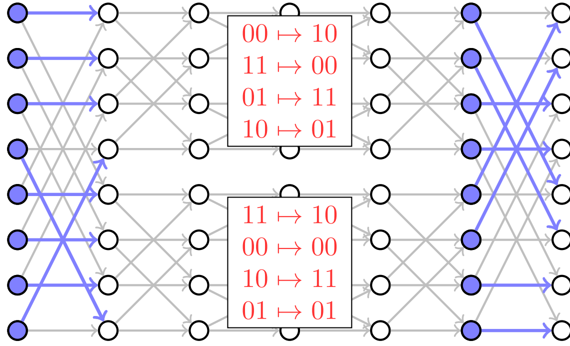
Step 15:



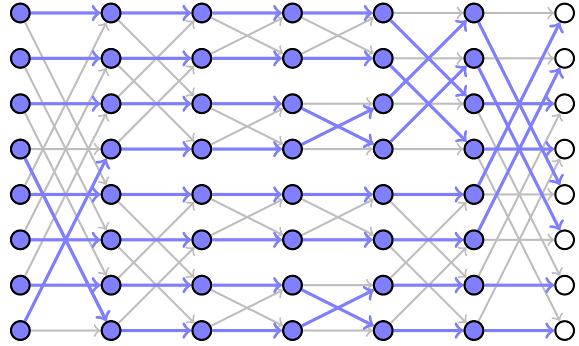
Step 16:



Step 17:



Step 18:



In the last step, the algorithm recursively routes the upper permutation through the upper sub-network and the lower permutation through the lower sub-network, and finally obtains the desired 8 vertex disjoint paths.

B.3 Routing Bit-Reversal Permutations

The κ-dimensional *bit-reversal* permutation $\mathrm{br}_\kappa \colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ is the permutation that, for every $w \in \{0,1\}^\kappa$, is defined by $\mathrm{br}_\kappa(w) := w^{\mathrm{r}}$.

We observe that two butterfly networks connected in tandem are capable of "routing" the bit-reversing permutation.

Claim B.8. *Let κ be a positive integer. The permutation br_κ can be routed on two κ-dimensional butterfly networks connected in tandem. Moreover, the routing can be found in $O(\kappa \cdot 2^\kappa)$ time and space.*

Proof. Consider the operations of "fold left" and "fold right", denoted fl_κ and fr_κ respectively, on a κ-bit string w that are defined as follows: letting $w = \sigma_0||\sigma_1$ if κ is even and $w = \sigma_0||b||\sigma_1$ if κ is odd, with $|\sigma_0| = |\sigma_1|$ and $b \in \{0,1\}$,

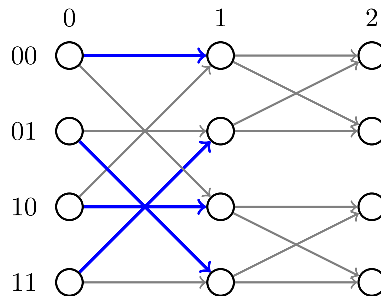
$$\begin{array}{lllll} & \textbf{fold left} & & \textbf{fold right} & \\ \text{even case} & \sigma_0||\sigma_1 & \longrightarrow (\sigma_0 \oplus \sigma_1^{\mathrm{r}})||\sigma_1 & \sigma_0||\sigma_1 & \longrightarrow \sigma_0||(\sigma_1 \oplus \sigma_0^{\mathrm{r}}) \\ \text{odd case} & \sigma_0||b||\sigma_1 & \longrightarrow (\sigma_0 \oplus \sigma_1^{\mathrm{r}})||b||\sigma_1 & \sigma_0||b||\sigma_1 & \longrightarrow \sigma_0||b||(\sigma_1 \oplus \sigma_0^{\mathrm{r}}) \end{array} .$$

Now observe that $\mathrm{br}_\kappa(w) = \big(\mathsf{fl}_\kappa \circ \mathsf{fr}_\kappa \circ \mathsf{fl}_\kappa\big)(w)$.

We now show the following two facts:

1. One can route fl_κ using layers 0 through $\lceil \kappa/2 \rceil$ of a κ-dimensional butterfly network. The proof is by induction over the dimension κ:

   - For $\kappa = 1$, we have that fl_1 is the identity permutation and thus can be trivially routed using layers 0 and 1 of a 1-dimensional butterfly network by simply using the two straight edges.
   - For $\kappa = 2$, the routing is the following:



   - For $\kappa > 2$, define the function $f_\kappa \colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ as follows

$$f_\kappa(w_1 \cdots w_\kappa) = w_1 \oplus w_\kappa || w_2 \cdots w_\kappa \ .$$

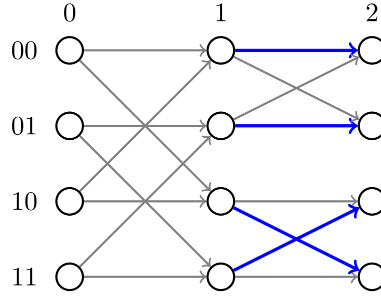
   For any $w \in \{0,1\}^\kappa$, $\mathsf{fl}_\kappa(w)$ can be computed using f_κ and $\mathsf{fl}_{\kappa-2}$ in the following way:

(a) Compute $w_1 \oplus w_\kappa || w_2 \cdots w_\kappa \leftarrow f_\kappa(w)$.
(b) Compute $w' = \mathsf{fl}_{\kappa-2}(w_2 \cdots w_{\kappa-1})$.
(c) Output $w_1 \oplus w_\kappa || w' || w_\kappa$.

Recall that each vertex (i, w) is connected to $(i+1, w)$ and $(i+1, w \oplus e_{i+1})$ in BN_κ. Thus, we can first route any $(0, w)$ to $(1, f_\kappa(w))$ using layers 0 and 1 of the butterfly network, by choosing the appropriate edge depending on w_κ. Then, using the induction hypothesis, we can route each vertex $(1, w_1 \cdots w_\kappa)$ to $(\lceil \kappa/2 \rceil, w_1 || \mathsf{fl}_{\kappa-2}(w_2 \cdots w_{\kappa-1}) || w_\kappa)$ using the two $(\kappa-1)$-dimensional subnetworks corresponding to the two different "fixed" values of w_1. This could be done since by the induction hypothesis for any $w_2 \cdots w_{\kappa-1}$ we can route $(1, w_2 \cdots w_{\kappa-1})$ to $(\lceil (\kappa-2)/2 \rceil, \mathsf{fl}_{\kappa-2}(w_2 \cdots w_{\kappa-1}))$ using a $(\kappa-2)$-dimensional network and therefore any $(1, w_2 \cdots w_\kappa)$ can be routed to $(\lceil (\kappa-1)/2 \rceil,$ $\mathsf{fl}_{\kappa-2}(w_2 \cdots w_{\kappa-1}) || w_\kappa)$ using a $(\kappa-1)$-dimensional butterfly network.

2. One can route fr_κ using layers $\lceil \kappa/2 \rceil$ through κ of a κ-dimensional butterfly network.

- For $\kappa = 1$, we have that fr_1 is the identity permutation and thus can be trivially routed using layers 0 and 1 of a 1-dimensional butterfly network by simply using the two straight edges.
- For $\kappa = 2$, the routing is the following:



- For $\kappa > 2$, define the function $g_\kappa \colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ as follows

$$g_\kappa(w_1 \cdots w_\kappa) = w_1 \cdots w_{\kappa-1} || w_\kappa \oplus w_1 \quad .$$

For any $w \in \{0,1\}^\kappa$, $\mathsf{fr}_\kappa(w)$ can be computed using g_κ and $\mathsf{fr}_{\kappa-2}$ in the following way:

(a) Compute $w'_1 \cdots w'_{\kappa-1} = \mathsf{fr}_{\kappa-2}(w_2 \cdots w_{\kappa-1})$.
(b) Compute $w_1 || w'_1 \cdots w'_{\kappa-1} || w_\kappa \oplus w_1 \leftarrow f_\kappa(w_1 || w'_1 \cdots w'_{\kappa-1} || w_\kappa)$.
(c) Output $w_1 || w'_1 \cdots w'_{\kappa-1} || w_\kappa \oplus w_1$.

By using the induction hypothesis, we know that we can route each vertex $(\lceil \kappa/2 \rceil, w_1 \cdots w_\kappa)$ to $(\kappa-1, w_1 || \mathsf{fr}_{\kappa-2}(w_2 \cdots w_{\kappa-1}) || w_\kappa)$ using the $2^{\lfloor \kappa/2 \rfloor}$ $\lceil \kappa/2 \rceil$-dimensional subnetworks corresponding to the four different "fixed" values for the tuple $(w_1, \cdots, w_{\lfloor \kappa/2 \rfloor})$. Recall that each vertex (i, w) is connected to $(i+1, w)$ and $(i+1, w \oplus e_{i+1})$ in BN_κ. Thus, we can next route any $(\kappa-1, w)$ to $(\kappa, g_\kappa(w))$ using layers $\kappa-1$ and κ of the butterfly network, by choosing the appropriate edge depending on w_κ.

Finally, we note that we are essentially done: to route $\mathrm{br}_\kappa = \mathsf{fl}_\kappa \circ \mathsf{fr}_\kappa \circ \mathsf{fl}_\kappa$ on two κ-dimensional butterfly networks connected in tandem, we first route fl_κ using the first half of the first network, then route fr_κ using the second half of the first network, then route fl_κ using the first half of the second network, and then finally route the identity permutation using the remaining second half of the second network. □

B.4 Simulating Beneš Networks with Butterfly Networks

Recall that a Beneš network is the concatenation of a butterfly network and a reversed butterfly network. For technical reasons, the reversed butterfly network is very inconvenient from an arithmetization standpoint (due to the need to keep the out-degree of the graph embedding in Section 11 as low as possible). Thus, we seek to do without it.

Specifically, we now show how, as far as routing is concerned, the reversed butterfly network can be "simulated" via five butterfly networks. In particular, because Beneš networks are re-arrangeable (see Theorem B.7), we deduce that so are six butterfly networks connected in tandem, a graph which we denote $\mathrm{BN}^{\ddagger 6}_\kappa$.

The high level idea is to simply use the the graph isomorphism from a reversed butterfly network to a butterfly network given by Claim B.3. However, the isomorphism "messes up" the connections with the preceding butterfly network. Fortunately, the isomorphism only shuffles rows of the reversed butterfly network according to a bit reversal permutation, and therefore we can "undo" the damage by preceding and following the butterfly network obtained by the isomorphism by bit-reversal permutations (each of which can be realized with two butterfly networks, as we saw in Claim B.8).

We therefore obtain the following claim:

Claim B.9. *Let κ be a positive integer and $\pi\colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ a permutation. There exists a set S'_π of 2^κ vertex-disjoint paths such that each vertex $(0, w)$ in $\mathrm{BN}_\kappa^{\ddagger 6}$ is connected to $(6\kappa + 1, \pi(w))$. Moreover, S'_π can be found in $O(\kappa \cdot 2^\kappa)$ time and space.*

Proof. Invoking Theorem B.7, we know that we can find (efficiently) a set of paths S_π routing π on a κ-dimensional Beneš network. We show how to map the 2^κ vertex-disjoint paths S_π over BENEŠ_κ to 2^κ vertex-disjoint paths S'_π over $\mathrm{BN}_\kappa^{\ddagger 6}$, by appropriately replacing the "second half" of each path p in S_π (that is the part of the path that would have traveled through the reversed butterfly network, which needs to be simulated).

Specifically, for each path p in S_π write $p = p_1 p_2$ by "splitting" the path in half; note that p_2 uses the reversed butterfly network. Let w and w' be the starting vertex and ending vertex in the path p_2. Our strategy is to replace p_2 with a new (longer) path p'_2, over five butterfly networks, that is equivalent to p_2 as far as routing is concerned. Concretely, let $\tilde{p}_2$ be the path obtained by taking the image of p_2 under ϕ_κ (the graph isomorphism from $\mathrm{BN}_\kappa^{\mathrm{r}}$ to BN_κ); then set $p'_2 := q_a \tilde{p}_2 q_b$ where q_a and q_b are respectively the paths (each over two butterfly networks) used to route w and $(w')^{\mathrm{r}}$ according to the bit-reversal permutation obtained via Claim B.8.

It is then easy to verify that the path $p' = p_1 p'_2$ has the same "end points" as p but, instead of using a Beneš network, uses six butterfly networks connected in tandem. Note that the collection S'_π of paths p' obtained as above, each from a path p in S_π, is indeed vertex-disjoint (as each of the three "segments" q_a, $\tilde{p}_2$, and q_b of p'_2 were picked without replacement from vertex-disjoint sets of paths) and routes π.

Finally, the efficiency guarantees of Theorem B.7 and Claim B.8, as well as the efficiency of computing ϕ_κ, easily imply an algorithm with the claimed efficiency. □

In fact, we can improve Claim B.9 to use only *four* butterfly networks connected in tandem, by simply omitting the bit reversal permutation carried out by the last two butterfly networks, by simply routing a different permutation π' obtained from π which *already* reverses the bits of the output of π.

Claim B.10. *Let κ be a positive integer and $\pi\colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ a permutation. There exists a set S'_π of 2^κ vertex-disjoint paths such that each vertex $(0, w)$ in $\mathrm{BN}_\kappa^{\ddagger 4}$ is connected to $(4\kappa + 1, \pi(w))$. Moreover, S'_π can be found in $O(\kappa \cdot 2^\kappa)$ time and space.*

Proof. We can modify the proof of Claim B.9 by routing the permutation $\pi' := \mathrm{br}_\kappa \circ \pi$ on the first four butterfly networks and neglecting to use the last two butterfly networks to "undo" the bit reversal over rows induced by the graph isomorphism ϕ_κ. □

B.5 De Bruijn Graphs and Their Rearrangeability

We finally deduce the fact that four De Bruijn graphs connected in tandem form a rearrangeable network.

Claim B.11. *Let κ be a positive integer and $\pi\colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ a permutation. There exists a set S''_π of 2^κ vertex-disjoint paths such that each vertex $(0, w)$ in $\mathrm{DB}_\kappa^{\ddagger 4}$ is connected to $(4\kappa + 1, \pi(w))$. Moreover, S''_π can be found in $O(\kappa \cdot 2^\kappa)$ time and space.*

Proof. From Claim B.5 (and the comment after it), we deduce that there is a graph isomorphism from $\mathrm{BN}_\kappa^{\ddagger 4}$ to $\mathrm{DB}_\kappa^{\ddagger 4}$ that preserves the order of the first and last column. Therefore, in order to find the desired set of paths S''_π we can simply take the image under the graph isomorphism of each path in the set S'_π guaranteed by Claim B.10. The efficiency of the algorithm follows from the efficiency guarantees of Claim B.5 and the efficiency of computing the graph isomorphism. □

How to route with only $3\kappa + \lceil \kappa/2 \rceil$ ("*three and a half*") De Bruijn graphs connected in tandem?

To begin with, we note that we can already "save a column": in light of the explicit algorithm in the proof of Theorem B.7, we could have defined Beneš networks (Definition B.6) to only have 2κ columns (by avoiding

the repeated "minimal cross" at the juncture of the butterfly network and the reversed butterfly network), and the Beneš network routing algorithm could have still been made to work.

Then, to save a half of a De Bruijn, we do as follows:

1. Find first a set of paths routing the given permutation on the newly defined Beneš network.
2. Use the first $\kappa + 1$ De Bruijn columns (i.e., column 0 through κ) to hold the partial paths on the first $\kappa + 1$ Beneš columns.
3. Then use De Bruijn columns κ through $2\kappa + \lceil \kappa/2 \rceil$ to route a bit-reversing permutation, following Claim B.8.
4. Then use De Bruijn columns $2\kappa + \lceil \kappa/2 \rceil$ through $3\kappa + \lceil \kappa/2 \rceil - 1$ to hold the partial paths on the Beneš columns $\kappa + 1$ through $2\kappa - 1$.

Overall we have used $3\kappa + \lceil \kappa/2 \rceil$ De Bruijn columns.

C Circuits

Let M be a two-tape random-access machine $M = \langle w, k, \mathbb{A}, \mathbb{P} \rangle$ (see Definition 7.1). Let us recall and define some new notation:

$$w = \text{ register width}$$
$$k = \text{ number of registers}$$
$$W = (1 + k)w = \text{ size of configuration}$$
$$n = \text{ number of instructions}$$
$$d = \text{ width of opcode.}$$

We write down the circuits we encounter in this paper:

- in Section C.1, we give the circuit for the the transition function δ_M of M,
- in Section C.2, we give the constraint circuit obtained for the reduction from bhRAM(M) to SuccinctGCP on De Bruijn graphs from Section 9.1, and
- in Section C.3, we give the constraint circuit obtained for the reduction from bhRAM(M) to SuccinctGCP on cyclic graphs from Section 9.2.

Notation. We use AND, OR, XOR, and NOT gates, as well as standard components such as MUX, DEMUX, and CMP (comparator). For CMP taking two input wires x and y, we label the output by "=", ">", or "<", to mean that the output is taken to be 1 respectively if $x = y$, $x > y$, and $x < y$.

C.1 Transition Function

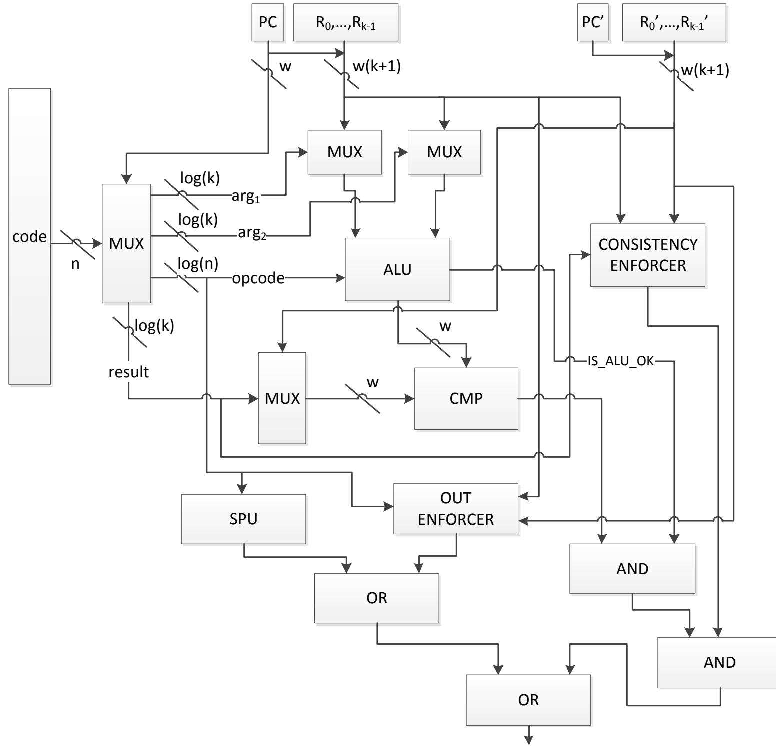


Figure 4: High-level view of a circuit for $[\delta_M]^{\mathsf{B}}$, the circuit for the **transition function** of M. (See Definition 7.7.) There are three "modules": the *consistency enforcer* is given in Figure 5, the *out enforcer* is given in Figure 6, and the *special-instruction processing unit* (SPU) is given in Figure 7.

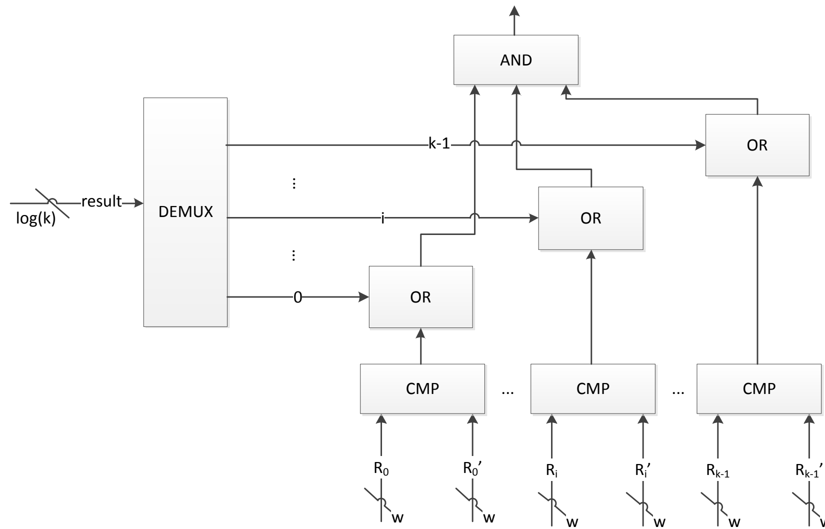


Figure 5: High-level view of the **consistency enforcer**. This circuit outputs 1 if and only if all the registers but the destination register are the same; it has depth $\log(w) + 2 + \log(k)$.

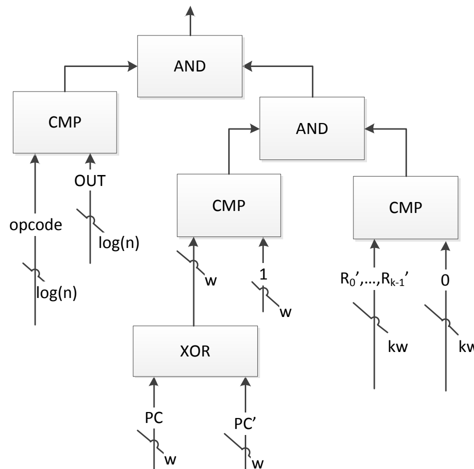


Figure 6: High-level view of the **out-enforcer**. This circuit outputs 1 if and only if $\mathsf{pc} = 2^w$ and $\mathsf{pc} = 0$ and all the primed registers are 0 and the opcode is out; it has depth $2 + \max\{\log(w), \log(w \cdot k), \log(n)\}$.

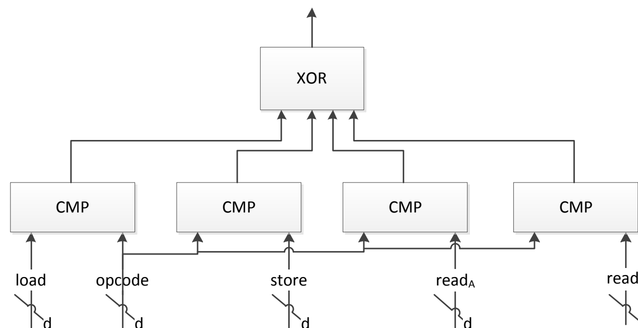


Figure 7: High-level view of the **special-instruction processing unit** (SPU). It outputs 1 if and only if the opcode is one of the "special" instructions (namely, read_A, read_B, load, and store). Its depth is $\log(d)$.

C.2 Constraint Circuit for De-Bruijn Graphs

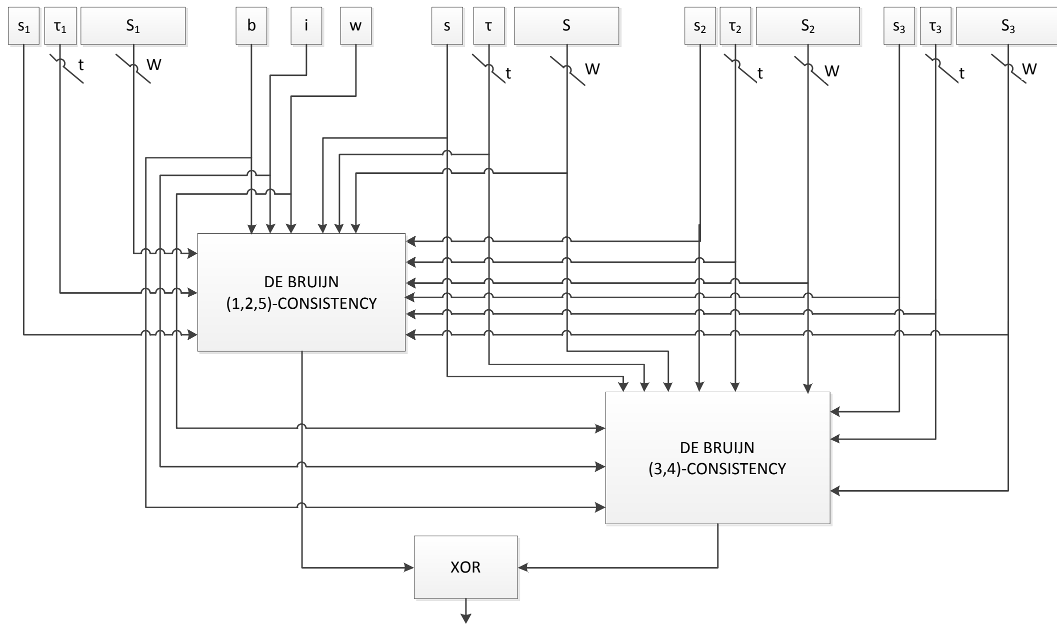


Figure 8: This circuit outputs 1 if and only if Equation 2 is satisfied, i.e., the colors of neighbors in a De Bruijn graph satisfy the requirements of Definition 9.12. It has two main components, the first checking items 1,2,5 of the definition (see Figure 9), and the second checking items 3 and 4 of the definition (see Figure 10).

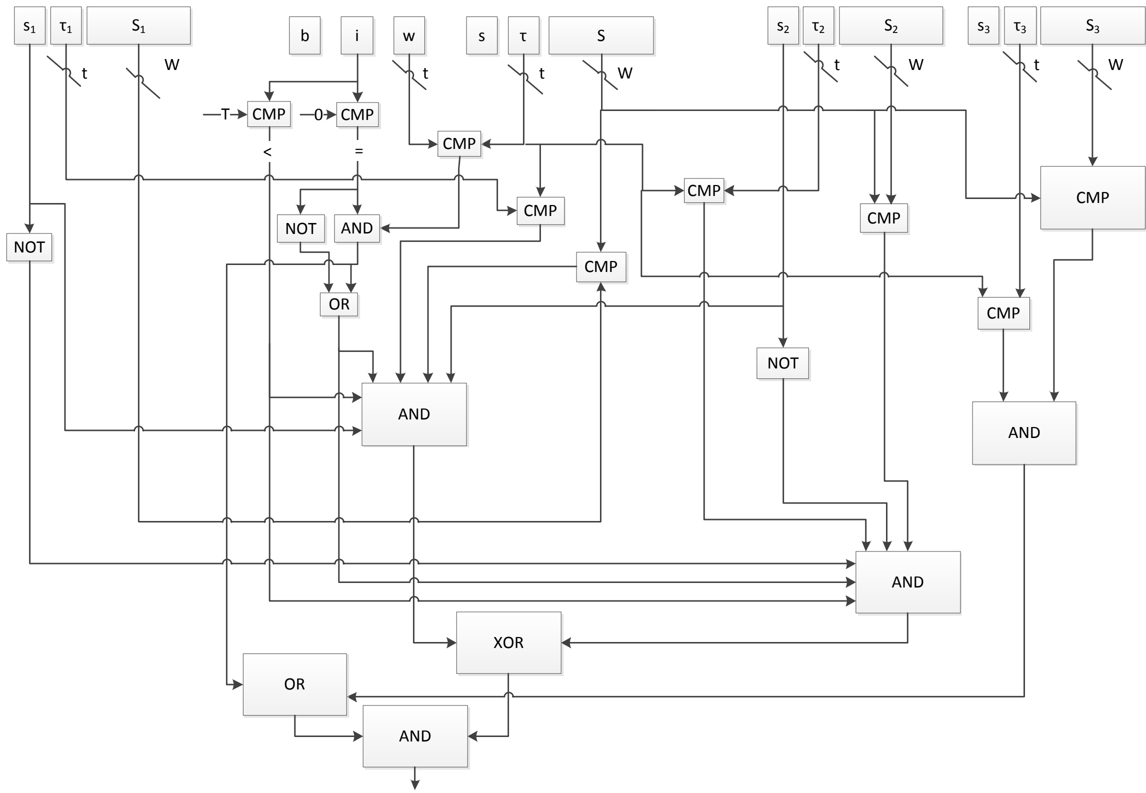


Figure 9: Outputs 1 if and only if two colors of neighbors in a De Bruijn graph satisfy the first, second and fifth requirements of the validity part in Definition 9.12.

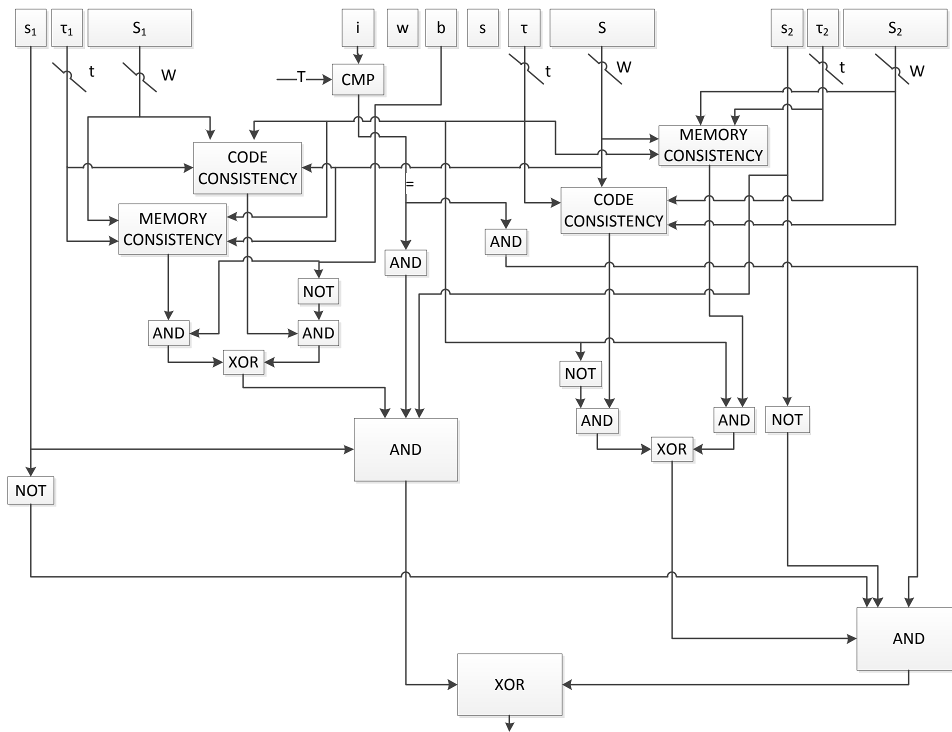


Figure 10: This circuit outputs 1 if and only if two colors of neighbors in a De Bruijn graph satisfy the third and fourth requirements of the validity part in Definition 9.12. The module of code consistency is given in Figure 11 and the module of memory consistency is given in Figure 12.

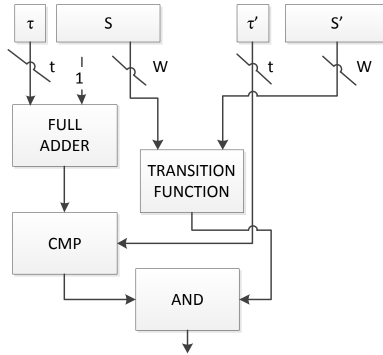


Figure 11: This circuit outputs 1 if and only if $\tau' = \tau + 1$ and $S \rightsquigarrow S'$.

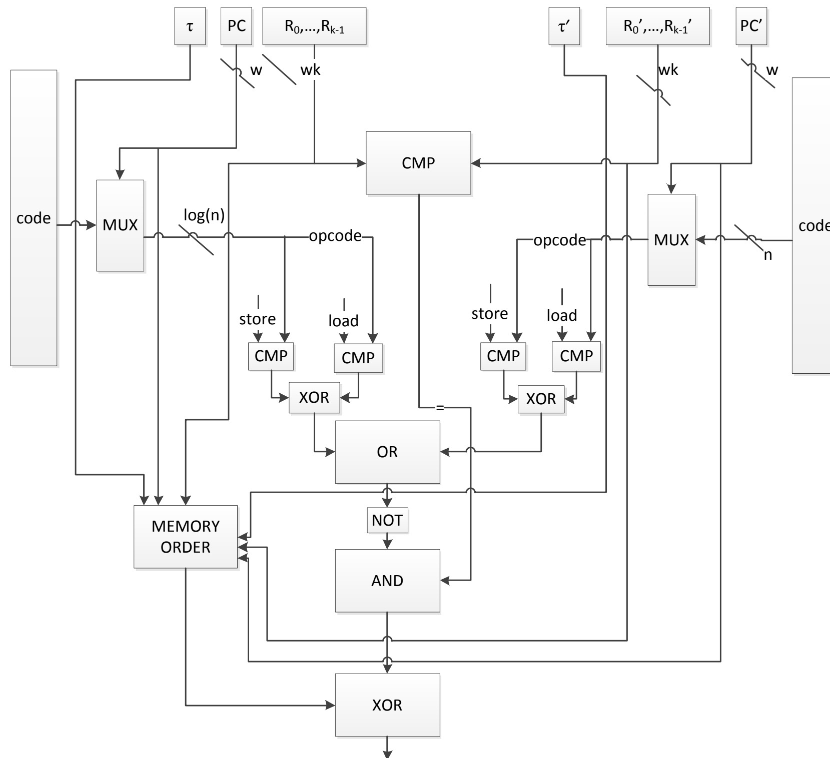


Figure 12: This circuit outputs 1 if and only if both S and S' contain memory instructions and (τ', S) precedes (τ'', S') in memory or $(\tau'' = \tau' \;\wedge\; S' = S)$.

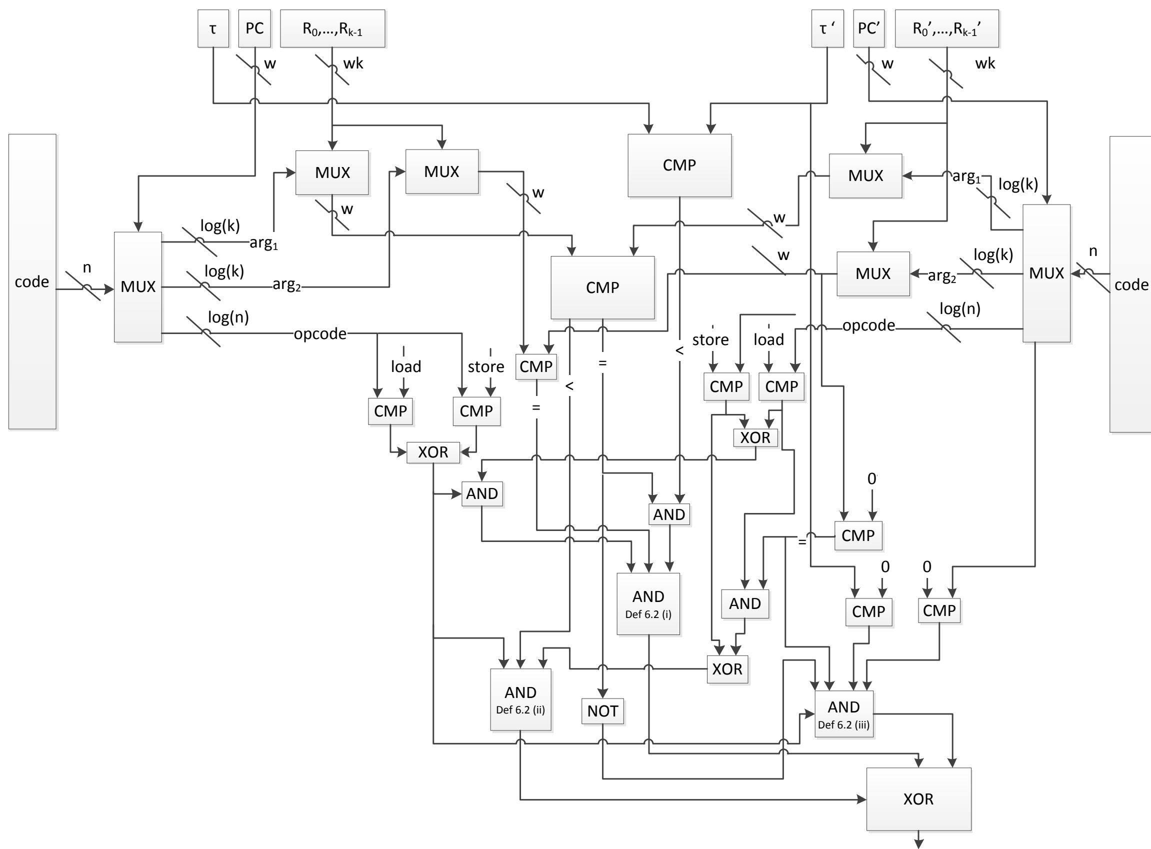


Figure 13: This circuit outputs 1 if and only if the two input configurations are memory ordered as defined inDefinition 9.2.

C.3 Constraint Circuit for Cyclic Graphs

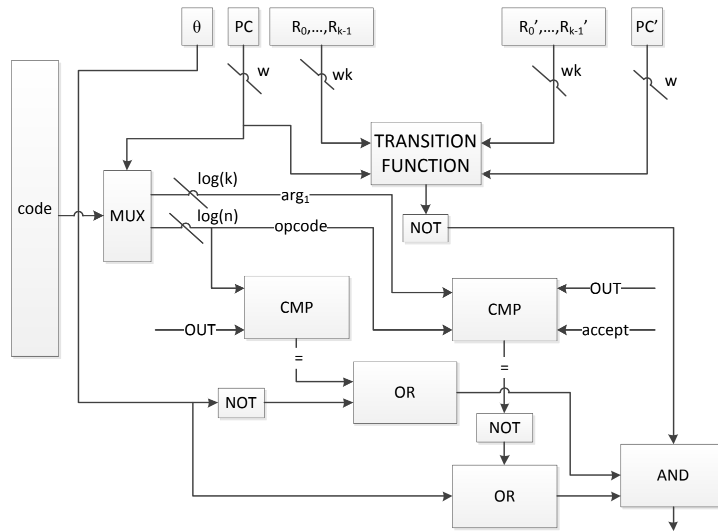


Figure 14: Outputs 1 if and only if Equation 4 is satisfied.

D Untrusted RAM Lemmas

We show how collision-resistant hash functions can be used in three quite useful computational Levin reductions from bounded-halting problems on random-access machines to bounded-halting problems on related random-access machines. Namely:

- In Section D.2, we explain how a random-access machine can be left to "check its own memory" by simply dynamically maintaining a Meklre tree over untrusted storage. In particular, in order to verify the correctness of such a computation, it suffices to verify the code consistency of the machine and not also its memory consistency. This lemma is what allows us to not rely on routing techniques in Section 9.2. Also, this lemma is a simple but crucial tool in the construction of (both publicly-verifiable and privately-verifiable) SNARKs and proof-carrying data in [BCCT12].

- In Section D.3, we explain how we can always move the input of a random-access machine from the input tape to the witness tape, leaving only a hash of the input on the input tape. This is quite useful when the original input is very long, and one happens to use protocols where there is a significant dependence on the input size.

  For example, applying this reduction at the top of any of the reduction stacks we discuss in this paper allows one to avoid paying in the input size, e.g., when using PCPs for SuccinctACSPs, without having to resort to the more expensive PCPs of Proximity. (See [BSCGT12] for more.) Also, as pointed out by [BCCT11], this lemma is a tool in the construction of quite simple delegation of memory and stream schemes based on succinct arguments of knowledge.

- In Section D.4, we explain how we can always move the program of a random-access machine to the witness tape, and only rely on an a *small interpreter machine* by leaving a hash of the program in the input tape of the machine. This is quite useful because the larger a program is the more expensive instruction fetch becomes, thereby increasing the depth of the transition function as a Boolean circuit, ultimately making reductions from that machine to other models (such as SuccinctACSPs considered in this work) more expensive.

  For example, applying this reduction at the top of any of the reduction stacks we discuss in this paper allows allows one to have a a somewhat less expensive arithmetization.

We now proceed to discuss the above lemmas in more detail in each of the following subsections; throughout, it will be useful to be familiar with the definitions of Section 7.

While there is nothing too deep in the proof of any of these lemmas, we feel the need to be quite explicit about their proof, especially the construction aspect, given their potential great practical value, so that given our proofs it will not be hard to implement any of these transformations.

Remark D.1. An aspect of the above Levin reductions that we shall not discuss explicitly is the "witness reductions". These are efficient, and exist in both directions of the reduction.

D.1 Untrusted Memory Lemma

We present a transformation f_{m} that enables a random-access machine to check its own memory consistency, incurring only in a blowup in running time that is logarithmic in the memory size.

Traditionally, a random-access machine operates under the assumption that *memory is consistent*: the computational model makes sure that the value stored in a given memory address is equal to the value loaded from this address in the following load from the same memory address. The transformation f_{m} that we seek enables a random-access machine to correctly operate *without* this assumption.

More precisely, f_{m}, on input a random-access machine M and a seed s, generates a new random-access machine M_s that only "trusts" a finite number of memory cells, namely its registers. At high level, if the value loaded from an address a is not the same as the value stored last in a, M_s will be able to recognize this is the case and abort the computation.

The main idea behind the construction of f_{m} is a follows.

First, M_s assumes that memory is initialized to the value 0 at every address, and will thus begin its computation by computing a Merkle hash tree over the entire memory using a collision-resistant hash function h_s, using only its (trusted!) registers to store intermediate hash values; since the leaves at the bottom of such

a hash tree all have the same value (namely, 0) this computation can be done in time that is logarithmic in the memory size and using only a constant number of registers. Next, M_s will store the root of the obtained tree in a dedicated root register.

After that M_s executes the program of M, but will replace every load and store instruction with "macros" implementing "secure load" and "secure store" instructions. Specifically:

- On every load instruction from address a in the program of M, M_s will verify the hash values on the path in the tree from a to the root; since the root of the hash tree is stored in a (trusted) register, because of collision resistance of the hash used, the loaded values have to be "consistent" with the root and thus have to be the value stored last in this address to begin with.
- On every store instruction to address a in the program of M, M_s must update the root register or the subsequent load instruction will fail; M_s can do so by recomputing the entire path form a to the root in the Merkle hash tree and update the root register.

Roughly, both of the above require time that is only logarithmic in the memory size.

We now proceed to the formal statement and its proof.

Lemma D.2 (Untrusted Memory)**.** *Let $\mathcal{H} = \{h_s \colon \{0,1\}^* \to \{0,1\}^{|s|}\}_{s \in \{0,1\}^{|s|}}$ be a CRH function family. There exists a deterministic polynomial-time function $f_{\mathsf{m}} \colon \{0,1\}^* \to \{0,1\}^*$ such that:*

1. Syntax: *For every random-access machine M and every seed $s \in \{0,1\}^*$, $M' := f_{\mathsf{m}}(s, M)$ is a random-access machine.*
2. Completeness: *There exists a function $T_{\mathsf{m}} \colon \{0,1\}^* \times \{0,1\}^* \times \mathbb{N} \to \mathbb{N}$ such that for every random-access machine M, for every seed $s \in \{0,1\}^*$, for every instance $(x, 1^t)$, if $(x, 1^t) \in \text{BH}\mathrm{RAM}(M)$ then $(x, 1^{T_{\mathsf{m}}(s,M,t)}) \in \text{BH}\mathrm{URAM}(f_{\mathsf{m}}(s, M))$.*

   Moreover, $T_{\mathsf{m}}(s, M, t) \stackrel{\text{def}}{=} O(2^t \cdot T_{h_s} \cdot \max\{w, |s|\})$, where T_{h_s} is the time it takes to compute h_s and w is the register size of M.
3. Soundness: *For every polynomial-size circuit family $\{C_\kappa\}_{\kappa \in \mathbb{N}}$, for every non-negative c, there exists a $K \in \mathbb{N}$ such that, for every $\kappa \in \mathbb{N}$ with $\kappa > K$,*

$$\Pr_{s \leftarrow \{0,1\}^\kappa}\left[\begin{array}{c} (x, 1^t) \notin \text{BH}\mathrm{RAM}(M) \\ \textit{and} \\ (w, \vec{S}) \textit{ is witness to } (x, 1^{T_{\mathsf{m}}(s,M,t)}) \in \text{BH}\mathrm{URAM}(f_{\mathsf{m}}(s, M)) \end{array} \middle| \, (x, t, M, (w, \vec{S})) \leftarrow C_\kappa(s) \right] \leq \frac{1}{\kappa^c} \ .$$

Proof. Define f_{m} and T_{m} as follows:

- $f_{\mathsf{m}}(s, M) \stackrel{\text{def}}{=} \textsc{SecureMem}(s, M)$, discussed below, and
- $T_{\mathsf{m}}(s, M, t) \stackrel{\text{def}}{=} O(2^t \cdot T_{h_s} \cdot w_s)$ where T_{h_s} is the time it takes to compute h_s and w_s is the register size of M_s as discussed below.

The function $\textsc{SecureMem}$ is the procedure that given a seed s and random-access machine $M = \langle w, k, \mathbb{A}, \mathbb{P} \rangle$ outputs a new random-access machine $M_s := \langle w_s, k + 10 + k', \mathbb{A}, \mathbb{P}_s \rangle$ defined as follows:

- The new register size is $w_s := \max\{w, |s|\} + 1$.[14]
- If k' denotes the number of registers needed to compute h_s, the new number of registers is $k + 10 + k'$.
- The arithmetic unit $\mathbb{A}$ remains unchanged.
- The new program $\mathbb{P}_s$ is constructed based on the old program $\mathbb{P}$, and we discuss this transformation next.

When describing the new program $\mathbb{P}_s$, we assume that we have access to the following basic procedures:

[14]The additional bit is needed in order to double the addressable memory size of M; the additional addressable memory will be used to store and update the Merkle hash tree.

- InitMerkle(s). On input a seed s for the family $\mathcal{H}$, this procedure computes the root of a Merkle tree over an all-zero memory with addresses from 0 to 2^{w_s-1} and stores the root of the constructed tree inside the $(k+1)$-th register (which we call the root register). Note that doing so takes time $O(w_s)$ because all of the values on each layer of the tree are equal and there are $O(w_s)$ different layers.

- GetSibling(a). Letting the $(k+2)$-th register be called the node register, on input an address a, this procedure writes into the $(k+3)$-th register (which we call the digest register), the hash value written in the sibling of the r_{node}-th node on the path from the node corresponding to the address a to the root of the Merkle hash tree; then, it sets the $(k+4)$-th register (which we call the $\mathsf{isRight}$ register) to be equal to 1 if it is the right sibling and 0 otherwise.

- SetPath(a). Letting the $(k+5)$-th register be called the newH register, on input an address a, this procedure writes the value stored in the newH register to the hash value written in the r_{root}-th node on the path from the node corresponding to the address a to the root of the Merkle hash tree.

At high level, this is how we are going to use the additional registers:

- The $(k+1)$-th register (i.e., the root register) always stores the root of a Merkle hash tree.
- The contents of the $(k+2)$-th register (i.e., the node register) are used by the GetSibling and SetPath procedures.
- The $(k+3)$-th register (i.e., the digest register) always stores the hash value output by the GetSibling procedure.
- The $(k+4)$-th register (i.e., the $\mathsf{isRight}$ register) always stores a flag to indicate if the GetSibling procedure output the hash value of the right sibling.
- The contents of the $(k+5)$-th register (i.e., the newH register) are used by the SetPath procedure.
- The $(k+6)$-th register (which we call the temp register) always stores intermediate results during the computations of the SecureLoad and SecureStore procedures. (Described below.)
- The $(k+7)$-th through $(k+10)$-th registers are always used for intermediate results of the InitMerkle, GetSibling, and SetPath procedures.
- The $(k+10+1)$-th through $(k+10+k')$-th registers are always used for evaluating the collision resistant hash function h.

We transform the program $\mathbb{P}$ into the new program $\mathbb{P}_s$ as follows:

- **Prep Memory.**

  The first sequence of instructions in $\mathbb{P}_s$ is the following sequence of instructions:

  1. InitMerkle(s).

- **Secure Loads.**

  Replace every load instruction in $\mathbb{P}$ with arguments a and i with the following sequence of instructions, which we denote SecureLoad(a, i):

  1. $r_{\mathsf{root}} \leftarrow 1$
  2. $r_i \leftarrow a \oplus 1$
  3. $\mathsf{load}\ i, \mathsf{temp}$
  4. $\mathsf{load}\ a, i$
  5. if $a \leq a \oplus 1$, then $r_{\mathsf{temp}} \leftarrow h_s(r_i, r_{\mathsf{temp}})$, otherwise $r_{\mathsf{temp}} \leftarrow h_s(r_{\mathsf{temp}}, r_i)$
  6. $r_{\mathsf{node}} \leftarrow 1$
  7. while $r_{\mathsf{node}} < t$ do:
     (a) GetSibling(a)
     (b) if r_{isRight} is 1, then $r_{\mathsf{temp}} \leftarrow h_s(r_{\mathsf{temp}}, r_{\mathsf{digest}})$, otherwise $r_{\mathsf{temp}} \leftarrow h_s(r_{\mathsf{digest}}, r_{\mathsf{temp}})$
     (c) $r_{\mathsf{node}} \leftarrow r_{\mathsf{node}} + 1$
  8. if $r_{\mathsf{temp}} = r_{\mathsf{root}}$ then $\mathsf{load}\ a, r_i$ else halt and reject

- **Secure Stores.**

  Replace every store instruction in $\mathbb{P}$ with arguments a and i with the following sequence of instructions, which we denote SecureStore(a, i):

  1. SecureLoad(a, temp)

2. $\textsc{SecureLoad}(a \oplus 1, \mathsf{temp})$
3. if $a \leq a \oplus 1$, then $r_{\mathsf{newH}} \leftarrow h_s(r_i, r_{\mathsf{temp}})$, otherwise $r_{\mathsf{newH}} \leftarrow h_s(r_{\mathsf{temp}}, r_i)$
4. $r_{\mathsf{node}} \leftarrow 1$
5. while $r_{\mathsf{node}} < t$ do:
   (a) $\textsc{SetPath}(a)$
   (b) $\textsc{GetSibling}(a)$
   (c) if r_{isRight} is 1, then $r_{\mathsf{newH}} \leftarrow h_s(r_{\mathsf{newH}}, r_{\mathsf{digest}})$, otherwise $r_{\mathsf{newH}} \leftarrow h_s(r_{\mathsf{digest}}, r_{\mathsf{newH}})$
   (d) $r_{\mathsf{node}} \leftarrow r_{\mathsf{node}} + 1$
6. $\mathsf{store}\ a, i$
7. $r_{\mathsf{root}} \leftarrow r_{\mathsf{newH}}$

As for the running time bound of M_s, note that every memory instruction in M is replaced with its secure version whose running time is $O(T_{h_s} \cdot w_s)$ where T_{h_s} is the time it takes to compute a single invocation of h_s. Thus, the overall running time is bounded by $O(2^t \cdot T_{h_s} \cdot w_s)$ as the number of load and store instructions is bounded by the time bound 2^t.

We now briefly explain why the lemma holds so long as $\mathcal{H}$ is a collision-resistant hash function family. Assume by way of contradiction that there exist a polynomial-size circuit family $\{C_\kappa\}_{\kappa\in\mathbb{N}}$ and non-negative c such that, for any $K \in \mathbb{N}$, there exists $\kappa \in \mathbb{N}$ with $\kappa > K$ such that

$$\Pr_{s\leftarrow\{0,1\}^\kappa}\left[\begin{array}{c}(x, 1^t) \notin \text{BH}\mathrm{RAM}(M)\\ \text{and}\\ (w, \vec{S}) \text{ is witness to } (x, 1^{T_{\mathsf{m}}(s,M,t)}) \in \text{BH}\mathrm{URAM}(f_{\mathsf{m}}(s, M))\end{array}\middle|\, (x, t, M, (w, \vec{S})) \leftarrow C_\kappa(s)\right] > \frac{1}{\kappa^c} \ .$$

It must be the case that, during the run $\vec{S} = (S_0, \ldots, S_{2^t-1})$ of M_s on input x, there exists an address a and a configuration S' in $\vec{S}$ that loads a value from a that is not consistent with the value last stored in the address a by some other configuration S''. For this instruction not to result in M_s outputting reject, it must be the case that the root register as updated after executing S'' is equal to the last hash value on the path from a to the root of the Merkle hash tree, as computed while executing the $\textsc{SecureLoad}$ instruction in S'. Thus it must be the case that somewhere on the path from a to the root, the circuit $C_\kappa(s)$ found a collision with probability (taken over a choice of s) better than $\frac{1}{\kappa^c}$, which is a contradiction to the collision resistance of $\mathcal{H}$. □

D.2 Untrusted Input Lemma

We present a transformation f_{i} that enables a random-access machine to check its own input consistency.

Traditionally, the input to a random access machine is given on a dedicated "trusted" input tape. The transformation f_{i} that we seek enables a random-access machine to receive the input on the "untrusted" witness tape, while only being given on the trusted input tape a short digest of the (possibly much longer) input.

More precisely, f_{i}, on input a random-access machine M and a seed s, generates a new random-access machine M_s that only gets as input a short string that is a commitment to the (possibly much longer) input string that is given on the witness type.

The main idea behind the construction of f_{i} is as follows.

First, M_s reads all the alleged input given to it on the witness tape and stores it into a dedicated segment of memory; at the same time, M_s also computes the hash digest (using the seed s) of the input and compares it to the digest given to it on the input tape; if the digests are different then M_s halts and outputs reject. Next, M_s simulates every read_A instruction (i.e., read to the input tape) in the program of M by loading the corresponding input symbol from memory instead. (Note that here we *are* assuming that memory is trusted! Of course, one can always apply the transformation f_{m} from Section D.1 after the transformation f_{i} from this section.)

We now proceed to the formal statement and its proof:

Lemma D.3 (Untrusted Input). *Let $\mathcal{H} = \{h_s \colon \{0,1\}^* \to \{0,1\}^{|s|}\}_{s\in\{0,1\}^*}$ be a CRH function family. There exists a deterministic polynomial-time function $f_{\mathsf{i}} \colon \{0,1\}^* \to \{0,1\}^*$ such that:*

1. Syntax: *For every random-access machine M and every seed $s \in \{0,1\}^*$, $M' := f_{\mathsf{i}}(s, M)$ is a random-access machine.*

2. Completeness: *There exists a function* $T_{\mathsf{i}} \colon \{0,1\}^* \times \mathbb{N} \to \mathbb{N}$ *such that for every random-access machine* M*, for every seed* $s \in \{0,1\}^*$*, for every instance* $(x, 1^t)$*, if* $(x, 1^t) \in \textsc{bh}\mathrm{RAM}(M)$ *then* $(h_s(x), 1^{T_{\mathsf{i}}(s,x,t)}) \in \textsc{bh}\mathrm{RAM}(f_{\mathsf{i}}(s, M))$*.*

   Moreover, $T_{\mathsf{i}}(s, x, t) \stackrel{\text{def}}{=} O(|x| \cdot T_{\textsc{Hash}} + 2^t)$*, where* $T_{\textsc{Hash}}$ *is the time it takes to compute "one step" of* h_s*. (See proof.)*

3. Soundness: *For every polynomial-size circuit family* $\{C_\kappa\}_{\kappa \in \mathbb{N}}$*, for every non-negative* c*, there exists a* $K \in \mathbb{N}$ *such that, for every* $\kappa \in \mathbb{N}$ *with* $\kappa > K$*,*

$$\Pr_{s \leftarrow \{0,1\}^\kappa}\left[\begin{array}{c} (x, 1^t) \notin \textsc{bh}\mathrm{RAM}(M) \\ \textit{and} \\ (w, \vec{S}) \textit{ is witness to } (h_s(x), 1^{T_{\mathsf{i}}(s,x,t)}) \in \textsc{bh}\mathrm{RAM}(f_{\mathsf{i}}(s, M)) \end{array} \middle| \, (x, t, M, (w, \vec{S})) \leftarrow C_\kappa(s) \right] \leq \frac{1}{\kappa^c} \ .$$

Proof. Define f_{i} and T_{i} as follows:

- $f_{\mathsf{i}}(s, M) \stackrel{\text{def}}{=} \textsc{SecureInput}(s, M)$, discussed below, and
- $T_{\mathsf{i}}(s, x, t) \stackrel{\text{def}}{=} O(|x| \cdot T_{\textsc{Hash}} + 2^t)$, where $T_{\textsc{Hash}}$ is the time it takes to compute a single invocation of the $\textsc{Hash}$ procedure discussed below.

The function $\textsc{SecureInput}$ is the procedure that given a seed s and random-access machine $M = \langle w, k, \mathbb{A}, \mathbb{P} \rangle$, outputs a new random-access machine $M_s := \langle w_s, k + 5 + k', \mathbb{A}, \mathbb{P}_s \rangle$ defined as follows:

- The new register size is $w_s := \max\{w, |s|\} + 1$.
- The new number of registers is $k + 5 + k'$, where k' comes from the following.

  We suppose that there is procedure called $\textsc{Hash}$ that uses k' registers (say registers $k+6$ through $k+k'+5$) and takes as input a w_s-bit string, and for any string x it holds that, after $|x|/w_s$ iterations of loading a fresh w_s-bit block from x into the $(k+5)$-th register (which we call the input register) and invoking $\textsc{Hash}$, the $(k+4)$-th register (which we call the digest register) contains the value of $h_s(x)$. For example, if h_s uses a Merkle-Damgård chain then $\textsc{Hash}$ will compute a single step in this chain.

- The new program $\mathbb{P}_s$ is constructed based on the old program $\mathbb{P}$, and we discuss this transformation next.

At high level, this is how we are going to use the additional registers:

- The $(k+1)$-th register (which we call the len register) always stores the total input length.
- The $(k+2)$-th register (which we call the eof register) always stores whether the current input symbol is # or not. Later on, it will store the number of symbols read form the input.
- The $(k+3)$-th register (which we call the addr register) always stores the memory address of the current input symbol.
- The $(k+4)$-th register (i.e., the digest register) always stores the current hash digest computed by $\textsc{Hash}$.
- The $(k+5)$-th register (i.e., the input register) always stores the current input to the $\textsc{Hash}$ procedure.
- The $(k+5+1)$-th through $(k+5+k')$-th registers are always used for computing $\textsc{Hash}$.

We transform the program $\mathbb{P}$ into the new program $\mathbb{P}_s$ as follows:

- **Prep input.**

  The first sequence of instructions in $\mathbb{P}_s$ is the following sequence of instructions:

  1. read_B input
  2. $r_{\mathsf{len}} \leftarrow 1$
  3. iseof $\mathsf{eof}, \mathsf{input}$
  4. $r_{\mathsf{addr}} \leftarrow 2^{w_s - 1}$
  5. while $r_{\mathsf{eof}} \neq 1$ do:
     (a) store $\mathsf{addr}, \mathsf{input}$
     (b) $\textsc{Hash}(s)$
     (c) read_B input
     (d) $r_{\mathsf{len}} \leftarrow r_{\mathsf{len}} + 1$

(e) $r_{\mathsf{addr}} \leftarrow r_{\mathsf{addr}} + 1$
(f) $\mathsf{iseof}\ \mathsf{eof}, \mathsf{input}$

6. read_A input
7. $r_{\mathsf{addr}} \leftarrow 2^{w_s - 1}$
8. $r_{\mathsf{eof}} \leftarrow 1$
9. if $r_{\mathsf{digest}} \neq r_{\mathsf{input}}$ then halt and reject

- **Secure** read_A**.**

  Replace every read_A instruction in $\mathbb{P}$ with argument i with the following sequence of instructions:

  1. if $r_{\mathsf{eof}} = r_{\mathsf{len}}$ then $r_i \leftarrow \#$
  2. load addr, i
  3. $r_{\mathsf{addr}} \leftarrow r_{\mathsf{addr}} + 1$
  4. $r_{\mathsf{eof}} \leftarrow r_{\mathsf{eof}} + 1$

As for the running time of M_s, note that because each input symbol is hashed the running time of M_s is bounded by $O(|x| \cdot T_{\text{HASH}} + 2^t)$, where T_{HASH} is the time it takes to compute a single invocation of HASH.

We now briefly explain why the lemma holds so long as $\mathcal{H}$ is a collision-resistant hash-function family. Assume by way of contradiction that there exist a polynomial-size circuit family $\{C_\kappa\}_{\kappa \in \mathbb{N}}$ and non-negative c such that, for any $K \in \mathbb{N}$, there exists $\kappa \in \mathbb{N}$ with $\kappa > K$ such that

$$\Pr_{s \leftarrow \{0,1\}^\kappa} \left[\begin{array}{c} (x, 1^t) \notin \text{BHRAM}(M) \\ \text{and} \\ (w, \vec{S}) \text{ is witness to } (h_s(x), 1^{T_{\mathsf{i}}(s,x,t)}) \in \text{BHRAM}(f_{\mathsf{i}}(s, M)) \end{array} \middle| (x, t, M, (w, \vec{S})) \leftarrow C_\kappa(s) \right] > \frac{1}{\kappa^c} .$$

Thus, it must be the case that during the run $\vec{S} = (S_0, \ldots, S_{2^t - 1})$ of M_s on input $h_s(x)$, M_s reads an input $x' \neq x$ from the witness tape such that $h_s(x) = h_s(x')$, because otherwise M_s would have halted and output reject. Thus it must be the case that $C_\kappa(s)$ found a collision with probability (taken over a choice of s) better than $\frac{1}{\kappa^c}$, which is a contradiction to the collision resistance of $\mathcal{H}$. □

D.3 Untrusted Code Lemma

We present a transformation f_{c} that enables a random-access machine to check its own code consistency.

Traditionally, the code (also called *program*) of a random-access machine is given as part of its definition. The transformation f_{c} that we seek produces a *universal* random-access machine M_s that gets as input a short commitment to the (possibly much longer) code given on the witness tape (code that allegedly is the one of M).

The main idea behind the construction of f_{c} is as follows.

First, M_s reads all the alleged code given to it on the witness type and stores in into a dedicated segment of memory; at the same time, M_s also computes the hash digest (using the seed s) of the code and compares it to the digest given to it on the input tape; if the digests are different then M_s halts and outputs reject. Next, M_s simulates every instruction of M by loading the corresponding instruction from memory instead and executing the instruction. (Note that here we *are* assuming that memory is trusted! Of course, one can always apply the transformation f_{m} from Section D.1 after the transformation f_{c} from this section.)

We now proceed to the formal statement and its proof:

Lemma D.4 (Untrusted Code)**.** *Let $\mathcal{H} = \{h_s \colon \{0,1\}^* \to \{0,1\}^{|s|}\}_{s \in \{0,1\}^{|s|}}$ be a CRH function family. There exists a deterministic polynomial-time function $f_{\mathsf{c}} \colon \{0,1\}^* \to \{0,1\}^*$ such that:*

1. Syntax: *For every random-access machine M and every seed $s \in \{0,1\}^*$, $M' := f_{\mathsf{c}}(s, M)$ is a random-access machine.*

2. Completeness: *There exists a function $T_{\mathsf{c}} \colon \mathbb{N} \to \mathbb{N}$ such that for every random-access machine M, for every seed $s \in \{0,1\}^*$, for every instance $(x, 1^t)$, if $(x, 1^t) \in \text{BHRAM}(M)$ then $(h_s(\mathbb{P}), 1^{T_{\mathsf{c}}(s,M,t)}) \in \text{BHRAM}(f_{\mathsf{c}}(s, M))$.*

   Moreover, $T_{\mathsf{c}}(s, x, t) \stackrel{\text{def}}{=} O(T_{\text{HASH}} \cdot |\mathbb{P}| + L \cdot k^3 \cdot 2^t)$, where L is the number of possible instructions in $M = \langle w, k, \mathbb{A}, \mathbb{P} \rangle$ and T_{HASH} is the time it takes to compute "one step" of h_s. (See proof.)

3. Soundness: *For every polynomial-size circuit family $\{C_\kappa\}_{\kappa\in\mathbb{N}}$, for every non-negative c, there exists a $K \in \mathbb{N}$ such that, for every $\kappa \in \mathbb{N}$ with $\kappa > K$,*

$$\Pr_{s\leftarrow\{0,1\}^\kappa}\left[\begin{array}{c}(x,1^t)\notin \textsc{bh}\mathrm{RAM}(M)\\ \textit{and}\\ (w,\vec{S}) \textit{ is witness to } (h_s(\mathbb{P}),1^{T_{\mathsf{c}}(s,M,t)})\in \textsc{bh}\mathrm{RAM}(f_{\mathsf{c}}(s,M))\end{array}\,\middle|\,(x,t,M,(w,\vec{S}))\leftarrow C_\kappa(s)\right]\leq\frac{1}{\kappa^c}\ ,$$

where $M = \langle w, k, \mathbb{A}, \mathbb{P}\rangle$.

Proof. Define f_{c} and T_{c} as follows:

- $f_{\mathsf{c}}(s, M) \stackrel{\text{def}}{=} \textsc{SecureProgram}(s, M)$, discussed below, and
- $T_{\mathsf{c}}(s, M, t) \stackrel{\text{def}}{=} O(T_{\textsc{Hash}} \cdot |\mathbb{P}| + L \cdot k^3 \cdot 2^t)$, where L is the number of possible instructions in $M = \langle w, k, \mathbb{A}, \mathbb{P}\rangle$ and $T_{\textsc{Hash}}$ is the time it takes to compute a single invocation of the Hash procedure discussed below.

The function SecureProgram is the procedure that given a seed s and random-access machine $M = \langle w, k, \mathbb{A}, \mathbb{P}\rangle$, outputs a new random-access machine $M_s := \langle w_s, k + 5 + k', \mathbb{A}, \mathbb{P}_s\rangle$ that is defined as follows:

- The new register size is $w_s := \max\{w, |s|\} + 1$.
- The new number of registers is $k + 5 + k'$, where k' comes from the following.

  We suppose that there is procedure called Hash that uses k' registers (say registers $k + 5 + 1$ through $k + 5 + k'$) that is able to compute $h_s(x)$, as we assumed in the proof of Lemma D.3.

  We further suppose that there is a procedure called Step that uses $k+2$ registers (say registers 1 through $k + 2$) in order to simulate M. That is, Step executes the instruction stored in the $(k + 1)$-th register (which we call the inst register) and updates the first k registers as needed. In addition, we assume that any time an instruction updates the pc register, Step updates the $(k+2)$-th register (which we call the pc' register). Note that the code for Step is of length $O(L\cdot k^3)$, where L is the number of possible instructions in M.
- The new program $\mathbb{P}_s$ is constructed based on the old program $\mathbb{P}$, and we discuss this transformation next.

At high level, this is how we are going to use the additional registers:

- The $(k+1)$-th register (i.e., the inst register) always stores the current instruction to be executed by Step.
- The $(k + 2)$-th register (i.e., the pc' register) always stores whether the current instruction symbol is # or not. Later on, it will store the simulated pc register.
- The $(k + 3)$-th register (which we call the addr register) always stores the memory address of the current input symbol.
- The $(k + 4)$-th register (which we call the digest register) always stores the current hash digest computed by Hash.
- The $(k + 5)$-th register (which we call the input register) always stores the current input to the Hash procedure.
- The $(k + 5 + 1)$-th through $(k + 5 + k')$-th registers are always used for computing Hash.

We transform the program $\mathbb{P}$ into the new program $\mathbb{P}_s$ as follows:

- **Prep Code.**

  The first sequence of instructions in $\mathbb{P}_s$ is the following sequence of instructions:

  1. read_B input
  2. iseof $\mathsf{pc}', \mathsf{input}$
  3. $r_{\mathsf{addr}} \leftarrow 2^{w_s-1}$
  4. while $r_{\mathsf{pc}'} \neq 1$ do
     (a) store $\mathsf{addr}, \mathsf{input}$
     (b) $\textsc{Hash}(s)$
     (c) read_B input
     (d) $r_{\mathsf{addr}} \leftarrow r_{\mathsf{addr}} + 1$

(e) $\mathsf{iseof}\ \mathsf{pc}', \mathsf{input}$

5. $\mathsf{read}_A\ \mathsf{input}$
6. $r_{pc'} \leftarrow 1$
7. if $r_{k+4} \neq r_{k+5}$ then halt and reject

- **Run.**

  Replace $\mathbb{P}$ with the following sequence of instructions:

  1. $r_{\mathsf{pc}'} \leftarrow 1$
  2. if $r_{\mathsf{pc}'} > |\mathbb{P}|$ then halt and reject
  3. $r_{\mathsf{addr}} \leftarrow 2^{w_s-1} - 1$
  4. $r_{\mathsf{addr}} \leftarrow r_{\mathsf{addr}} + r_{\mathsf{pc}'}$
  5. $\mathsf{load}\ \mathsf{addr}, \mathsf{inst}$
  6. STEP
  7. $\mathsf{jmp}\ 2$

As for the running time of M_s, note that because each instruction in $\mathbb{P}$ is hashed the running time of the **PrepCode** phase is bounded by $O(T_{\text{HASH}} \cdot |\mathbb{P}|)$ where T_{HASH} is the time it takes to compute a single execution of HASH; next, each instruction of M is simulated using the procedure STEP, which consists of $O(L \cdot k^3)$ instructions, where L is the number of possible instructions in M. Thus, the total running time is $O(T_{\text{HASH}} \cdot |\mathbb{P}| + L \cdot k^3 \cdot 2^t)$.

We now briefly explain why the lemma holds so long as $\mathcal{H}$ is a collision-resistant hash-function family. Assume by way of contradiction that there exist a polynomial-size circuit family $\{C_\kappa\}_{\kappa \in \mathbb{N}}$ and non-negative c such that, for any $K \in \mathbb{N}$, there exists $\kappa \in \mathbb{N}$ with $\kappa > K$ such that

$$\Pr_{s \leftarrow \{0,1\}^\kappa}\left[\begin{array}{c}(x, 1^t) \notin \text{BHRAM}(M) \\ \text{and} \\ (w, \vec{S}) \text{ is witness to } (h_s(\mathbb{P}), 1^{T_{\mathsf{c}}(s,M,t)}) \in \text{BHRAM}(f_{\mathsf{c}}(s, M))\end{array} \middle| (x, t, M, (w, \vec{S})) \leftarrow C_\kappa(s)\right] > \frac{1}{\kappa^c} .$$

Thus, it must be the case that during the run $\vec{S} = (S_0, \ldots, S_{2^t-1})$ of M_s on input $h_s(x)$, M_s reads a code $\mathbb{P}' \neq \mathbb{P}$ from the witness tape such that $h_s(\mathbb{P}') = h_s(\mathbb{P})$, because otherwise M_s would have halted and output reject. Thus it must be the case that $C_\kappa(s)$ found a collision in $\mathcal{H}$ with probability (taken over a choice of s) better than $\frac{1}{\kappa^c}$, which is a contradiction to the collision resistance of $\mathcal{H}$. $\square$

E Finite Fields and Efficient Computation

We review some computational properties of finite fields; we also develop a number of algorithmic results that are crucial for "arithmetizing" graph coloring problems in Section 11 in a way that ensures that certain high-degree polynomials have small arithmetic circuits computing them.

Throughout, *linearized polynomials* (see [LN97, Section 2.5]) will play a crucial role both for speeding up further already efficient computations (e.g., faster interpolation and evaluation algorithms in Section E.3 and Section E.4) as well as for ensuring that certain high-degree polynomials can be computed efficiently (in Section E.6 and Section E.7).

Whenever possible, we state results for a generic field characteristic p; we will be explicit when we must take $p = 2$ (which ultimately is the special case of our interest). Also, whenever we make statements about the time or space complexity of an algorithm, we will make the simplifying assumption that basic field operations can be performed in unit time and field elements can be stored at unit cost.

E.1 Irreducible and Primitive Polynomials

We represent elements of a finite field as polynomials modulo an irreducible polynomial of the appropriate degree. Specifically, to represent elements of $\mathbb{F}_q$, where $q = p^f$ and p is the characteristic of $\mathbb{F}_q$: we consider any *irreducible* polynomial I over $\mathbb{F}_p$ of degree f; I has a root x in $\mathbb{F}_q$ and thus $\mathbb{F}_q = \mathbb{F}_p(\mathsf{x})$, so that every element of $\mathbb{F}_q$ can be uniquely expressed as a polynomial in x over $\mathbb{F}_p$ of degree less than f. See [LN97, Section 2.5] for more details. In particular, this representation will allow us to perform field operations in time that is polylogarithmic in the field size and space that is logarithmic in the field size.

Irreducible polynomials of a given degree over a finite field can be found deterministically, in polynomial time if the characteristic is small [Sho88, Corollary 3.2]:

Theorem E.1 (Finding Irreducible Polynomials)**.** *There exists a deterministic algorithm* FINDIRRPOLY *that, on input* $(p, 1^f)$ *where p is a prime and f is a positive integer, computes an irreducible polynomial I of degree f over $\mathbb{F}_p$ in time* $\mathsf{poly}(p, f)$*. Specifically, there exists a universal constant $c > 0$ such that the running time of* FINDIRRPOLY$(p, 1^f)$ *is*

$$O\Big(p^{1/2+c} f^{3+c} + (\log p)^2 f^{4+c}\Big) \enspace ;$$

in particular, FINDIRRPOLY$(1^f) :=$ FINDIRRPOLY$(2, 1^f)$ *runs in polynomial time. (For convenience, we denote by* t_{IRR} *and* s_{IRR} *the time and space complexities of this algorithm.)*

We will only deal with (finite) fields of characteristic 2, so, for simpliciy, every time we invoke FINDIRRPOLY, we will leave implicit the first input $p = 2$, and write only FINDIRRPOLY(1^f).

Next, we recall the definition of a *primitive* polynomial; for more details about primitive polynomials, see [LN97, Section 3.1].

Definition E.2. *Let p be a prime and ℓ a positive integer. A polynomial Ξ of degree ℓ over $\mathbb{F}_p$ is* **primitive** *over $\mathbb{F}_p$ if Ξ is the minimal polynomial over $\mathbb{F}_p$ of a primitive element of $\mathbb{F}_{p^\ell}$.*

Finding primitive polynomials is not known to be easy:

Theorem E.3 (Finding Primitive Polynomials)**.** *There exists a deterministic algorithm* FINDPRIMPOLY *that, on input* $(p, 1^\ell)$ *where where p is a prime and ℓ is a positive integer, computes a primitive polynomial Ξ of degree ℓ over $\mathbb{F}_p$ in time* $\mathsf{poly}(p^\ell)$*. We define* FINDPRIMPOLY$(1^\ell) :=$ FINDPRIMPOLY$(2, 1^\ell)$*. (For convenience, we denote by* t_{PRIM} *and* s_{PRIM} *the time and space complexities of this algorithm.)*

Proof. Shoup [Sho99] shows how to compute a minimal polynomial of degree ℓ in time $\mathsf{poly}(\ell)$. Hence, the "hard" part is to find a primitive element of $\mathbb{F}_{p^\ell}$. Shparlinski [Shp96] shows that a primitive element of $\mathbb{F}_{p^\ell}$ can be found in time approximately $O(p^{\ell/4})$. □

Nonetheless, we will usually be interested in primitive polynomials of low degree, so the inefficiency of finding them will not matter. (And, in practice, one always relies on pre-computed tables anyways; see Remark E.6.)

We use primitive polynomials in Section 11 to create "artificial" cyclic groups inside a finite field of characteristic 2, in order to embed a certain graph into an affine graph over the finite field. The ability to create cyclic structure is given by the following claim:

Claim E.4. *Let ℓ be a positive integer, and let Ξ be a primitive polynomial of degree ℓ over $\mathbb{F}_2$. Then, for any non-negative integers i and j, x^i and x^j are congruent modulo the polynomial Ξ if and only if i and j are congruent modulo $2^\ell - 1$.*

Proof. Recall the following fact:

> **Theorem** ([LN97, Theorem 3.18]). *Let p be a prime. The monic polynomial P of positive degree m over $\mathbb{F}_p$ is a primitive polynomial over $\mathbb{F}_p$ if and only if $(-1)^m P(0)$ is a primitive element of $\mathbb{F}_p$ and the least positive integer r for which x^r is congruent modulo P to some element of $\mathbb{F}_p$ is $r = \frac{p^m-1}{p-1}$. In case P is primitive over $\mathbb{F}_q$, we have $x^r \equiv (-1)^m P(0) \bmod P(x)$.*

We invoke the above theorem with $p := 2$, $P := \Xi$, and $m := \ell$. We obtain that $r = 2^\ell - 1$. Moreover, since Ξ is primitive, $\Xi(0) \neq 0$ (by [LN97, Theorem 3.16]) and thus, over a field of characteristic 2, we have that $(-1)^m \Xi(0) = \Xi(0) = 1$. We deduce that $1, x, \ldots, x^{2^\ell-2} \not\equiv 0, 1 \bmod \Xi(x)$ and $x^{2^\ell-1} \equiv 1 \bmod \Xi(x)$, thereby proving the claim. □

The above claim gives us the following simple corollary:

Corollary E.5. *Let ℓ and Ξ as in Claim E.4. Define $\xi_i(x) := x^i \bmod \Xi(x)$ for $i = 0, \ldots, 2^\ell - 2$. Then, for any $i \in \{0, \ldots, 2^\ell - 2\}$ and positive integer c,*

$$\xi_{(i+c \bmod (2^\ell-1))}(x) = x^c \cdot \xi_i(x) + Q(x) \cdot \Xi(x) \ ,$$

where $Q(x)$ is the (unique) polynomial quotient in $\mathbb{F}_2[x]$ when dividing $x^c \cdot \xi_i(x)$ by $\Xi(x)$.

Proof. The corollary easily follows from the definition of the $\xi_0(x), \ldots, \xi_{(2^\ell-2)}(x)$, Claim E.4, and the definition of congruence under division of univariate polynomials:

$$\begin{aligned}
\xi_{(i+c \bmod (2^\ell-1))}(x) &= x^{(i+c \bmod (2^\ell-1))} \mod \Xi(x) && \text{(by definition of } \xi_{(i+c \bmod (2^\ell-1))}(x)\text{)} \\
&\equiv x^{i+c} \mod \Xi(x) && \text{(since } x^i \equiv x^j \bmod \Xi(x) \Leftrightarrow i \equiv j \bmod (2^\ell - 1)\text{)} \\
&\equiv x^c \cdot x^i \mod \Xi(x) \\
&\equiv x^c \cdot \xi_i(x) \mod \Xi(x) \ ,
\end{aligned}$$

so that $\xi_{(i+c \bmod (2^\ell-1))}(x) = x^c \cdot \xi_i(x) + Q(x) \cdot \Xi(x)$, as desired. □

Remark E.6. Ultimately we are interested in practical implementations of the Levin reductions discussed in this paper. In practice, one keeps tables of irreducible and primitive polynomials, so that we will not worry much about the time needed to compute such polynomials of the correct degree.

E.2 Linear Maps and Sparse Polynomials

Ben-Sasson et al. [BSGH+05, Section 5] discuss the computational advantages of working with linear subspaces of finite fields; while some of the underlying *algebraic* facts were already used in [BSS08] and [BSGH+06], their *computational* properties were only used in [BSGH+05], where they are critical to argue for an efficient verifier.

In this section, we recall some of the results of Ben-Sasson et al. [BSGH+05, Section 5], and complement them with further details and new results altogether (which ultimately are needed for an explicit description of an efficient PCP verifier, as constructed by [BSCGT12]). For a more in-depth discussion of the theory of linear algebra for vector spaces over finite fields, see [LN97, Chapter 3.4].

Throughout this section, we let $\mathbb{B} \subseteq \mathbb{F}$ be two fields of sizes $|\mathbb{B}| = p$ and p^f respectively. For a linear subset $H \subseteq \mathbb{F}$ of dimension h over the (smaller) field $\mathbb{B}$ (i.e., there is a basis $(\alpha_1, \ldots, \alpha_h)$ of elements in $\mathbb{F}$ such that every element of H can be expressed as $\sum_{i=1}^h c_i \alpha_i$ with $c_1, \ldots, c_h \in \mathbb{B}$), we say that a function $g \colon H \to \mathbb{F}$ is a $\mathbb{B}$-*linear map* if $g(ax + by) = ag(x) + bg(y)$ for every $x, y \in H$ and $a, b \in \mathbb{B}$.

A first observation is that $\mathbb{B}$-linear maps can be represented as sparse low-degree polynomials. (In other words, they have sparse *low-degree extensions*; see Section E.4.)

Claim E.7 ([BSGH+05, Proposition 5.1])**.** *Let $H \subseteq \mathbb{F}$ be a vector space of dimension h over the (smaller) field $\mathbb{B}$ and $g\colon H \to \mathbb{F}$ a $\mathbb{B}$-linear map. Then there exists a (unique) polynomial $\hat{g}\colon \mathbb{F} \to \mathbb{F}$ of the form*

$$\hat{g}(x) = \sum_{i=0}^{h-1} c_i x^{p^i} \enspace ,$$

where $c_0, \ldots, c_{h-1} \in \mathbb{F}$, such that $\hat{g}$ agrees with g on all of H. Moreover, given the evaluations of g on any basis $\mathcal{B}_H$ for H, the coefficients $c_0, \ldots, c_{h-1}$ can be computed with $\mathsf{poly}(h, \log p)$ arithmetic operations over $\mathbb{F}$.

Remark E.8. By inspecting the proof of Claim E.7, we see that the $\mathbb{B}$-linearity of f implies that the coefficients $c_0, \ldots, c_{h-1}$ are the (unique) solution to the following linear system: letting $\mathcal{B}_H = (e_1, \ldots, e_h)$,

$$\begin{pmatrix} e_1 & e_1^p & e_1^{p^2} & \cdots & e_1^{p^{h-1}} \\ e_2 & e_2^p & e_2^{p^2} & \cdots & e_2^{p^{h-1}} \\ \vdots & \vdots & \ddots & \vdots & \\ e_h & e_h^p & e_h^{p^2} & \cdots & e_h^{p^{h-1}} \end{pmatrix} \cdot \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{h-1} \end{pmatrix} = \begin{pmatrix} g(e_1) \\ g(e_2) \\ \vdots \\ g(e_h) \end{pmatrix} \enspace . \tag{29}$$

Hence, given $g(e_1), \ldots, g(e_h)$, the coefficients $c_0, \ldots, c_{h-1}$ can indeed be found using $O(h^2 \log p + h^3)$ arithmetic operations over $\mathbb{F}$, because finding the entries of the matrix from $\mathcal{B}_H$ requires $O(h^2 \log p)$ arithmetic operations via repeated squaring, and then $O(h^3)$ arithmetic operations are needed for Gaussian elimination. Later, in Section E.4.4, we shall discuss an $O(h^2 \log h \cdot \log p)$ recursive algorithm for finding the coefficients $c_0, \ldots, c_{h-1}$.

Once the coefficients $c_0, \ldots, c_{h-1}$ have been computed, however, evaluating $\hat{g}$ at a given point $\alpha \in \mathbb{F}$ is a *very simple* arithmetic circuit, involving only the use of repeated squaring to find $\alpha, \alpha^p, \ldots, \alpha^{p^{h-1}}$, multiplying each α_i by the respective coefficient c_i, and adding up the results.

A very important class of polynomials are *vanishing polynomials* [BSGH+05, Definition 5.2]:

Definition E.9. *For any subset S of $\mathbb{F}$, the S-***vanishing polynomial in** $\mathbb{F}$*, denoted $Z_S(x)$, is defined to be the polynomial in $\mathbb{F}[x]$ whose zeros are precisely the elements of S, that is,*

$$Z_S(x) = \prod_{s \in S} (x - s) \enspace .$$

It is easy to see that when S is a vector space over the base field $\mathbb{B}$, then $Z_S\colon \mathbb{F} \to \mathbb{F}$ is a $\mathbb{B}$-linear map, and we call Z_S a *subspace polynomial*:

Claim E.10 ([BSGH+05, Proposition 5.3])**.** *If S is a vector space over the base field $\mathbb{B}$ then $Z_S\colon \mathbb{F} \to \mathbb{F}$ is a $\mathbb{B}$-linear map, that is,*

- *for all $u, v \in \mathbb{F}$, $Z_S(u + v) = Z_S(u) + Z_s(v)$, and*
- *for all $a \in \mathbb{B}$ and $v \in \mathbb{F}$, $Z_S(a \cdot v) = a \cdot Z_S(v)$.*

Because of the above properties, a subspace polynomial is also known as a *linearized polynomial.* In particular, one can show, using Claim E.7, that, whenever S is a vector space over the base field $\mathbb{B}$, Z_S is sparse and its coefficients can be found fast:

Claim E.11 ([BSGH+05, Proposition 5.4])**.** *If S is a d-dimensional vector space over the base field $\mathbb{B}$, then there exist $c_1, \ldots, c_{d-1} \in \mathbb{F}$ such that*

$$Z_S(x) = x^{p^d} + \sum_{i=0}^{d-1} c_i x^{p^i} \enspace .$$

Moreover, the coefficients $c_0, \ldots, c_{d-1}$ can be computed with $\mathsf{poly}(d, \log p)$ arithmetic operations over $\mathbb{F}$, when given as input a basis $\mathcal{B}_S$ for S.

Example E.1. There are important special cases where finding a subspace polynomial is *very* easy (and one need not even solve any linear system to find the coefficients of the polynomial):

- If $S = \mathbb{F}$, then $Z_S(x) = x^{p^f} - x$.
- If p divides f, then $Z_S(x) = x^{p^{f/p}} - x$.

Furthermore, if the characteristic of the field is $p = 2$, then $x^{p^{f/p}} - x = x^{p^{f/p}} + x$, which is the field trace function from $\mathbb{F}_{p^f}$ to $\mathbb{F}_{p^{f/p}}$.

Whenever possible, we will try to work with such special cases, to take advantage of "super-sparse" subspace polynomials, and thus greatly speed up computations.

Remark E.12. For notational convenience, we denote by FindSubspPoly the algorithm that, on input (an irreducible polynomial representing) $\mathbb{F}$ and a basis $(e_1, \ldots, e_d)$ for S, outputs an arithmetic circuit $[Z_S]^{\mathsf{A}}$ for computing Z_S; note that $[Z_S]^{\mathsf{A}}$ has size $O(d)$.

Since Claim E.11 relies on Claim E.7, FindSubspPoly could in principle require $O(h^2 \log p + h^3)$ field operations (see Remark E.8). Furthermore, we are not "allowed" to benefit from the even faster $O(h^2 \log h \cdot \log p)$-time $O(h)$-space algorithm mentioned in Remark E.8 (and discussed in Section E.4.4) for finding the desired coefficients in Claim E.7, because FindSubspPoly is a subroutine of that faster algorithm.

Indeed, in this special case, there is instead a simple recursive algorithm, which requires only $O(d^2 \log p)$ running time and $O(d \log p)$ space.

Specifically, first note that, for $d > 1$,

$$\begin{aligned}
Z_S(x) &= Z_{\mathrm{span}(e_1,\ldots,e_d)}(x) \\
&= \prod_{i=1}^{p} Z_{\mathrm{span}(e_1,\ldots,e_{d-1})+(i-1)e_d}(x) \\
&= \prod_{i=1}^{p} Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x-(i-1)e_d) \\
&= \prod_{i=1}^{p} \left(Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x) - (i-1)\cdot Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(e_d)\right) \\
&= \prod_{i=1}^{p} \left(Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x) + (i-1)\cdot Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(e_d)\right) \qquad \text{(by rearranging)} \\
&= \sum_{i=1}^{p} \left(\sum_{\substack{r_1<\cdots<r_i \\ r_1,\ldots,r_i\in\{1,\ldots,i\}}} r_1\cdots r_i \right) Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(e_d)^{p-i} Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x)^i \\
&= Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x)^p + (p-1) Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(e_d)^{p-1} Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x) \qquad \text{(since } p \text{ is prime)} \\
&= Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x^p) + (p-1) Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(e_d)^{p-1} Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x) \ . \qquad (30)
\end{aligned}$$

Therefore, the following algorithm computes the coefficients for $Z_S(x)$: on input an irreducible polynomial I for representing $\mathbb{F}$ and the basis $(e_1, \ldots, e_d)$ for S,

FindSubspPoly$(I, e_1, \ldots, e_d) \equiv$

1. If $d = 1$, output $x^p + (p-1)e_1^{p-1}x$.
2. If $d > 1$, do the following:
   (a) Run FindSubspPoly$(e_1, \ldots, e_{d-1})$ to generate $[Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}]^{\mathsf{A}}$.
   (b) Using $[Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}]^{\mathsf{A}}$, compute $[Z_{\mathrm{span}(e_1,\ldots,e_d)}]^{\mathsf{A}}$ by following Equation 30.
   (c) Output $[Z_{\mathrm{span}(e_1,\ldots,e_d)}]^{\mathsf{A}}$.

The correctness of FindSubspPoly easily follows from the derivation of Equation 30 above, and its time complexity of $O(d^2 \log p)$ and space complexity of $O(d \log p)$ easily follows from its simple recursive structure.

E.3 Polynomial Evaluation

The problem of *polynomial evaluation* is the following:

- **input:** a field $\mathbb{F}$, a subset $S \subseteq \mathbb{F}$, and a polynomial $P(x) \in \mathbb{F}[x]$ of degree at most $|S| - 1$;
- **output:** a function $p \colon S \to \mathbb{F}$ such that $p(\alpha) = P(\alpha)$ for every $\alpha \in S$; p is known as the *evaluation* of P over S.

Note that the problem of polynomial evaluation easily generalizes to finding multi-variate evaluations of multi-variate polynomials; this generalization will not be of interest to us for the purpose of constructing Levin reductions (though will briefly arise in [BSCGT12] for optimizing the speed of the PCP prover).

A naïve evaluation of a polynomial of degree d at a point takes $O(d^2)$ field operations; using Horner's method, only $O(d)$ field operations are required. Still, if d is on the order of $|S|$, then evaluation of the polynomial over S would take $O(|S|^2)$ field operations.

Not surprisingly better algorithms are known:

Theorem E.13 ([vzGG03, Corollary 10.8]). *A polynomial evaluation over S of a polynomial of degree at most $|S| - 1$ can be computed in $O(\mathsf{M}(|S|) \log |S|)$ field operations, where $\mathsf{M}(n)$ is the time to multiply two polynomials of degree at most n. (Recall that, ithout additional assumptions on S, the best upper bound on $\mathsf{M}(n)$ is $O(n^{\log 3}) \approx O(n^{1.585})$, via Karatsuba's algorithm.)*

Fortunately, in the applications that we have in mind, the sets S in which we will be interested do satisfy additional properties: they will be linear subsets of the field. Therefore, we will be able to benefit from faster evaluation algorithms that use additive FFT methods to obtain great *quasilinear* running times.

Specifically, in the special case (of our interest) where $\mathbb{F} = \mathbb{F}_{2^f}$ for some $f \in \mathbb{N}$ and S is a linear subset of $\mathbb{F}$, much faster algorithms are known. This is the problem of *polynomial evaluation over linear subsets.*

For example, the additive FFT of the von zur Gathen and Gerhard [vzGG96] has complexity $O(|S|(\log |S|)^2)$; this algorithm derives its speed from the use of linearized polynomials. (This algorithm is given explicitly in [BSCGT12].)

Algorithms with even better asymptotic complexity are known; see for example [Mat08, Chapters 3]. Nonetheless, the von zur Gathen and Gerhard additive FFT can be implemented very easily and does not "hide" any big constants in its practical running time.

We note that Bhattacharyya [Bha05, Section 2.1] also developed a fast algorithm for evaluation over linear subsets, but his algorithm in practice performs worse than the von zur Gathen and Gerhard additive FFT.

E.4 Polynomial Interpolation

The problem of *polynomial interpolation* is the following:

- **input:** a field $\mathbb{F}$, a function $p \colon S \to \mathbb{F}$ with $S \subseteq \mathbb{F}$;
- **output:** a polynomial $P(x) \in \mathbb{F}[x]$ of degree at most $|S| - 1$ that agrees with p on S; P is known as the *low-degree extension* of p in $\mathbb{F}$.

Note that the problem of polynomial interpolation easily generalizes to finding multi-variate low-degree extensions of multi-variate functions.

In this section we discuss computational properties of the problem of polynomial interpolation, especially in some special cases that play a particularly important role in the applications that we consider.

E.4.1 Existence and uniqueness of low-degree extensions

We begin by recalling that low-degree extensions exist and are unique; for completeness, we shall review the proof of this fact.

Theorem E.14. *Let $\mathbb{F}$ be a finite field, m a positive integer, $s_1, \dots, s_m$ positive integers, and $S_1, \dots, S_m$ subsets of $\mathbb{F}$ with respective cardinalities $s_1, \dots, s_m$.*

For any function $f \colon S_1 \times \cdots \times S_m \to \mathbb{F}$ there exists a unique m-variate polynomial $\hat{f} \colon \mathbb{F}^m \to \mathbb{F}$ such that the following two conditions are satisfied:

(i) Low Degree: *for $i = 1, \dots, m$, the degree of $\hat{f}$ in the i-th variable is less than s_i, and*

(ii) Consistency: *$\hat{f}$ agrees with f on $S_1 \times \cdots \times S_m$, that is, for every $(\alpha_1, \dots, \alpha_m) \in S_1 \times \cdots \times S_m$, it is the case that $\hat{f}(\alpha_1, \dots, \alpha_m) = f(\alpha_1, \dots, \alpha_m)$.*

We call $\hat{f}$ the **low-degree extension of f in $\mathbb{F}$**, *and denote it* $\mathrm{LDE}_{\mathbb{F},m,(S_1,\ldots,S_m)}(f)$.

In other words, the low-degree extension of a function is simply a polynomial that has the function "embedded" into it; moreover, this polynomial is unique.

The first step in the proof of Theorem E.14 is to establish that the low-degree extension of the zero function (which certainly exists) is unique:

Lemma E.15. *Let $\mathbb{F}$, m, $s_1,\ldots,s_m$, and $S_1,\ldots,S_m$ be as in Theorem E.14, and let $z\colon S_1\times\cdots\times S_m\to\mathbb{F}$ be the zero function on $S_1\times\cdots\times S_m$. Then $\mathrm{LDE}_{\mathbb{F},m,(S_1,\ldots,S_m)}(z)$ is the unique (m-variate) identically-zero polynomial over $\mathbb{F}$.*

Proof. By induction on m. In the case $m=1$, the lemma follows from the fact that a univariate polynomial with positive degree d cannot have more than d roots: indeed, since the polynomial $\mathrm{LDE}_{\mathbb{F},1,(S_1)}(z)$ is required to vanish on all s_1 elements of S_1, and yet have degree less than s_1, it must be the case that $\mathrm{LDE}_{\mathbb{F},1,(S_1)}(z)$ is the (unique) identically-zero polynomial. In the case $m>1$, assume the lemma holds for all positive integers m' less than m, and consider any m-variate polynomial $\hat{z}$ that is a low-degree extension of z in $\mathbb{F}$. Let α be any element in $\mathbb{F}$, and consider the $(m-1)$-variate polynomial $\hat{z}_\alpha$ over $\mathbb{F}$ defined by $\hat{z}_\alpha(x_2,\ldots,x_m):=\hat{z}(\alpha,x_2,\ldots,x_m)$. By the inductive assumption, $\hat{z}_\alpha$ is the unique $(m-1)$-variate identically-zero polynomial over $\mathbb{F}$, because it is a low-degree extension in $\mathbb{F}$ of the function z_α, where $z_\alpha(x_2,\ldots,x_m):=z(\alpha,x_2,\ldots,x_m)$. Next, let $\vec{\beta}=(\beta_2,\ldots,\beta_m)$ be any element in $\mathbb{F}^{m-1}$, and consider the univariate polynomial $\hat{z}_{\vec{\beta}}$ defined by $\hat{z}_{\vec{\beta}}(x_1):=\hat{z}(x_1,\beta_2,\ldots,\beta_m)$. Again, by the inductive assumption, $\hat{z}_{\vec{\beta}}$ is the unique univariate identically-zero polynomial over $\mathbb{F}$, because it is a low-degree extension in $\mathbb{F}$ of the function $z_{\vec{\beta}}$, where $z_{\vec{\beta}}(x_1):=z(x_1,\beta_2,\ldots,\beta_m)$. We conclude that $\hat{z}$ vanishes everywhere on $\mathbb{F}^m$, and thus is the unique m-variate identically-zero polynomial over $\mathbb{F}$. $\square$

The next step is to use the previous lemma to deduce that low-degree extensions are unique, when they exist:

Corollary E.16. *Let $\mathbb{F}$, m, $s_1,\ldots,s_m$, and $S_1,\ldots,S_m$ be as in Theorem E.14. Given a function $f\colon S_1\times\cdots\times S_m\to\mathbb{F}$, if a low-degree extension of f in $\mathbb{F}$ exists, then it is unique.*

Proof. Suppose that two *distinct* m-variate polynomials $\hat{f}_1$ and $\hat{f}_2$ are low-degree extensions of f over $\mathbb{F}$. Then, the polynomial $\hat{z}':=\hat{f}_1-\hat{f}_2$ would be a not-identically-zero low-degree extension of the all-zero function on $S_1\times\cdots\times S_m$. This contradicts Lemma E.15, which guarantees that the only low-degree extension of the all-zero function on $S_1\times\cdots\times S_m$ is the m-variate identically-zero polynomial over $\mathbb{F}$. $\square$

We are left to show that low-degree extensions actually exist. We do this for point functions first:

Lemma E.17. *Let $\mathbb{F}$, m, $s_1,\ldots,s_m$, and $S_1,\ldots,S_m$ be as in Theorem E.14, and let $\vec{\alpha}=(\alpha_1,\ldots,\alpha_m)$ be an element in $S_1\times\cdots\times S_m$. There exists a (unique) m-variate polynomial $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}\colon\mathbb{F}^m\to\mathbb{F}$ such that:*

- *for $i=1,\ldots,m$, the degree of $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}$ in the i-th variable is s_i-1,*
- *$\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(\vec{\alpha})=1_{\mathbb{F}}$, and*
- *$\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(S_1\times\cdots\times S_m-\{\vec{\alpha}\})=\{0_{\mathbb{F}}\}$.*

Proof. Define the "Lagrange interpolant" polynomial $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}\colon\mathbb{F}^m\to\mathbb{F}$ by

$$\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(\vec{x}):=\prod_{i=1}^{m}\left(\prod_{\beta_i\in S_i-\{\alpha_i\}}\frac{x_i-\beta_i}{\alpha_i-\beta_i}\right) .$$

For every $i=1,\ldots,m$,

$$\deg_{x_i}\left(\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(\vec{x})\right)=\deg_{x_i}\left(\prod_{\beta_i\in S_i-\{\alpha_i\}}\frac{x_i-\beta_i}{\alpha_i-\beta_i}\right)=s_i-1 .$$

Moreover, $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(\vec{\alpha})=\prod_{i=1}^{m}\prod_{\beta_i\in S_i-\{\alpha_i\}}\frac{\alpha_i-\beta}{\alpha_i-\beta}=1$. Finally, for every $\vec{\gamma}\in S_1\times\cdots\times S_m-\{\vec{\alpha}\}$, $\vec{\gamma}$ and $\vec{\alpha}$ differ in at least one coordinate $i\in\{1,\ldots,m\}$, so that $\gamma_i\in S_i$; hence, $\prod_{\beta_i\in S_i-\{\alpha_i\}}\frac{\gamma_i-\beta_i}{\alpha_i-\beta_i}=0_{\mathbb{F}}$, because, when $\beta_i=\gamma_i$, the numerator of the ratio becomes zero. (Note that, since $\beta_i\in S_i-\{\alpha_i\}$, the denominator never vanishes!)

The uniqueness of $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}$ follows from Corollary E.16, because $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}$ is a low-degree extension of the "point" function that is equal to one on $\{\vec{\alpha}\}$ and zero everywhere else on $S_1\times\cdots\times S_m$. $\square$

Finally, constructing low-degree extensions for general functions easily follows from the previous lemma, by taking appropriate linear combinations of low-degree extensions of point functions:

Proof of Theorem E.14. For every $\vec{\alpha} = (\alpha_1, \dots, \alpha_m)$ in $S_1 \times \cdots \times S_m$, let $\delta_{\mathbb{F},m,(S_1,\dots,S_m),\vec{\alpha}} \colon \mathbb{F}^m \to \mathbb{F}$ be the polynomial guaranteed by Lemma E.17. Define the polynomial $\hat{f} \colon \mathbb{F}^m \to \mathbb{F}$ by

$$\hat{f}(x_1, \dots, x_m) = \sum_{\vec{\alpha} \in S_1 \times \cdots \times S_m} f(\vec{\alpha}) \cdot \delta_{\mathbb{F},m,(S_1,\dots,S_m),\vec{\alpha}}(x_1, \dots, x_m) \ .$$

Indeed, for every $i = 1, \dots, m$,

$$\deg_{x_i}\left(\hat{f}(x_1, \dots, x_m)\right) \leq \max_{\vec{\alpha} \in S_1 \times \cdots \times S_m} \left\{ \deg_{x_i}\left(\delta_{\mathbb{F},m,(S_1,\dots,S_m),\vec{\alpha}}(x_1, \dots, x_m)\right)\right\} = s_i - 1 < s_i \ ;$$

also, for every $\vec{\alpha}' \in S_1 \times \cdots \times S_m$,

$$\hat{f}(\vec{\alpha}') = \sum_{\vec{\alpha} \in S_1 \times \cdots \times S_m} f(\vec{\alpha}) \cdot \delta_{\mathbb{F},m,(S_1,\dots,S_m),\vec{\alpha}}(\vec{\alpha}) = f(\vec{\alpha}') \cdot 1_{\mathbb{F}} = f(\vec{\alpha}') \ .$$

The uniqueness of $\hat{f}$ follows from Corollary E.16, because $\hat{f}$ is, by construction, a low-degree extension of f. □

E.4.2 Complexity of the general case

We now briefly discuss the computational properties of low-degree extensions, in the general case.

Univariate case. One way to compute a low-degree extension is to simply follow the constructive proof of Theorem E.14, a process known as *Lagrange interpolation.* The Lagrange interpolant polynomials can be pre-computed for a given set of evaluating points S (and can thus be used for distinct functions f defined over S); this can be done in $|S|^2$ time. Then, the linear combination of the Lagrange interpolant polynomials can also be computed in $|S|^2$ time [vzGG03, Theorem 5.1]

An alternative to Lagrange interpolation, with the same time complexity, is *Newton interpolation* [vzGG03, Exercise 5.11].

Also, we recall that, in the univariate case, polynomial interpolation can alternatively be viewed as having to find the solution of a linear system of $|S|$ equations involving a Vandermonde matrix.

However, even when no additional properties are assumed about S, there are faster divide-and-conquer algorithms to compute a low-degree extension:

Theorem E.18 ([vzGG03, Corollary 10.12])**.** *A low-degree extension can be computed in $O(\mathsf{M}(|S|)\log|S|)$ field operations, where $\mathsf{M}(n)$ is the time to multiply two polynomials of degree at most n. (Recall that, ithout additional assumptions on S, the best upper bound on $\mathsf{M}(n)$ is $O(n^{\log 3}) \approx O(n^{1.585})$, via Karatsuba's algorithm.)*

Fortunately, in the applications that we have in mind, the sets S in which we will be interested do satisfy additional properties: they will be linear subsets of the field. Therefore, we will be able to benefit from faster interpolation algorithms that use additive inverse FFT methods to obtain great *quasilinear* running times. (See Section E.4.3.)

Multivariate case. Once again, one way to proceed is to follow the constructive proof of Theorem E.14: the Lagrange interpolant polynomials from the proof of Theorem E.14 can be pre-computed for a given set of evaluating points $S_1 \times \cdots \times S_m$ in $\prod_{i=1}^{m} |S_i|^2$ time; then, the linear combination of the Lagrange interpolant polynomials can also be found in $\prod_{i=1}^{m} |S_i|^2$ time.

E.4.3 Interpolation over linear subsets

In the special case (of our interest) where $\mathbb{F} = \mathbb{F}_{2^f}$ for some $f \in \mathbb{N}$ and S is a linear subset of $\mathbb{F}$, much faster algorithms are known. This is the problem of *polynomial interpolation over linear subsets.*

For example, the additive inverse FFT that is the "dual" of the von zur Gathen and Gerhard additive FFT [vzGG96] has complexity $O(|S|(\log|S|)^2)$; this algorithm also derives its speed from the use of linearized polynomials. (This algorithm is given explicitly in [BSCGT12].)

Algorithms with even better asymptotic complexity are known; see for example [Mat08, Chapters 4.4 and 4.5]. Nonetheless, the von zur Gathen and Gerhard additive inverse FFT can be implemented very easily and does not "hide" any big constants in its practical running time.

We note that Bhattacharyya [Bha05, Section 2.1] also developed a fast algorithm for interpolation over linear subsets, but his algorithm in practice performs worse than the von zur Gathen and Gerhard additive inverse FFT.

E.4.4 Linearized interpolation

Another special case (of our interest) is a problem that we call *linearized polynomial interpolation*:

- **input:** a basis $\mathcal{B} = (\alpha_1, \dots, \alpha_h)$ and a vector of values $\vec{\beta} = (\beta_1, \dots, \beta_h)$;
- **output:** a linearized polynomial $P(x) = \sum_{j=0}^{h-1} \gamma_j x^{2^j} \in \mathbb{F}[x]$ such that $P(\alpha_i) = \beta_i$.

In other words, we are asked to solve the following linear system:

$$\begin{pmatrix} \alpha_1 & \alpha_1^2 & \alpha_1^4 & \cdots & \alpha_1^{2^{h-1}} \\ \alpha_2 & \alpha_2^2 & \alpha_2^4 & \cdots & \alpha_2^{2^{h-1}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_h & \alpha_h^2 & \alpha_h^4 & \cdots & \alpha_h^{2^{h-1}} \end{pmatrix} \begin{pmatrix} \gamma_0 \\ \gamma_1 \\ \vdots \\ \gamma_{h-1} \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_h \end{pmatrix} . \tag{31}$$

Note that it is indeed important for $\alpha_1, \dots, \alpha_h$ to be linearly independent, otherwise the system may not always have a solution.

Recall from Remark E.8 that the problem of linearized polynomial interpolation arises when finding the coefficients of the (sparse) low-degree extensions of maps that are linear over the base field; finding these sparse low-degree extensions is important when we intend to carefully arithmetize Boolean circuits. See Section E.7 for more details.

Kopparty suggested to us an approach mimicking the Vandermonde algorithm [Kop10]:

Theorem E.19. *There exists an algorithm* SolveLPIP *such that, on input an irreducible polynomial I, basis $\mathcal{B} = (\alpha_1, \dots, \alpha_h)$, and vector $\vec{\beta} = (\beta_1, \dots, \beta_h)$, solves the linearized polynomial interpolation problem in $O(h^2 \log h \cdot \log p)$ time and $O(h)$ space.*

Proof. The idea is to use linearized polynomials and solve the problem recursively. We describe the desired algorithm SolveLPIP in three main steps:

1. Find two linearized polynomials $Q(x)$ and $R(x)$ of respective degrees at most $2^{\lfloor h/2 \rfloor}$ and $2^{\lceil h/2 \rceil}$ such that:

   $$Q(\alpha_1) = Q(\alpha_2) = \cdots = Q(\alpha_{\lfloor h/2 \rfloor}) = 0_{\mathbb{F}} \quad \text{and} \quad R(\alpha_{\lfloor h/2 \rfloor+1}) = R(\alpha_{\lfloor h/2 \rfloor+2}) = \cdots = R(\alpha_h) = 0_{\mathbb{F}} \ .$$

   Note that $Q(x)$ and $R(x)$ can be found in $O(h^2 \log p)$ field operations via divide and conquer, e.g., by setting $R(x) := \textsc{FindSubspPoly}(I, (\alpha_1, \dots, \alpha_{\lfloor h/2 \rfloor}))$ and $R(x) := \textsc{FindSubspPoly}(I, (\alpha_{\lfloor h/2 \rfloor+1}, \dots, \alpha_h))$. (See Remark E.12.)

2. Compute

   $$\alpha'_1 := R(\alpha_1), \alpha'_2 := R(\alpha_2), \dots, \alpha'_{\lfloor h/2 \rfloor} := R(\alpha_{\lfloor h/2 \rfloor}) \quad \text{and} \\ \alpha'_{\lfloor h/2 \rfloor+1} := Q(\alpha_{\lfloor h/2 \rfloor+1}), \alpha'_{\lfloor h/2 \rfloor+2} := Q(\alpha_{\lfloor h/2 \rfloor+2}), \dots, \alpha_h := Q(\alpha_h) \ .$$

3. Recursively solve two (smaller) linearized polynomial interpolation problems by computing

   $$R'(x) := \textsc{SolveLPIP}(I, \mathcal{B}^{(0)}, \vec{\beta}^{(0)}) \quad \text{and} \quad Q'(x) := \textsc{SolveLPIP}(I, \mathcal{B}^{(1)}, \vec{\beta}^{(1)})$$

   where

   $$\mathcal{B}^{(0)} := (\alpha'_1, \dots, \alpha'_{\lfloor h/2 \rfloor}) \quad \text{and} \quad \vec{\beta}^{(0)} := (\beta_1, \dots, \beta_{\lfloor h/2 \rfloor}) \\ \mathcal{B}^{(1)} := (\alpha'_{\lfloor h/2 \rfloor+1}, \dots, \alpha'_h) \quad \text{and} \quad \vec{\beta}^{(1)} := (\beta_{\lfloor h/2 \rfloor+1}, \dots, \beta_h) \ .$$

4. Compute the polynomial $P(x) := Q'(Q(x)) + R'(R(x))$, and output $P(x)$.

Note that:

- for $i = 1, \ldots, \lfloor h/2 \rfloor$, $P(\alpha_i) = Q'(Q(\alpha_i)) + R'(R(\alpha_i)) = Q'(0_{\mathbb{F}}) + R'(\alpha'_i) = 0_{\mathbb{F}} + \beta_i = \beta_i$, and
- for $i = \lceil h/2 \rceil, \ldots, h$, $P(\alpha_i) = Q'(Q(\alpha_i)) + R'(R(\alpha_i)) = Q'(\alpha'_i) + R'(0_{\mathbb{F}}) = \beta_i + 0_{\mathbb{F}} = \beta_i$.

Moreover, the composition and the addition of two linearized polynomials is still a linearized polynomial; in particular, since $Q(x)$, $Q'(x)$, $R(x)$, and $R'(x)$ are all linearized polynomials, so is $Q'(Q(x))$ and $R'(R(x))$, and thus $P(x)$ as well. By induction, the degrees of R' and Q' are respectively at most $2^{\lfloor h/2 \rfloor - 1}$ and $2^{\lceil h/2 \rceil - 1}$, so that $Q'(Q(x))$ and $R'(R(x))$ have degrees that are respectively at most at most $2^{\lfloor h/2 \rfloor} 2^{\lceil h/2 \rceil - 1} = 2^{h-1}$ and $2^{\lceil h/2 \rceil} 2^{\lfloor h/2 \rfloor - 1} = 2^{h-1}$.

As for the number of field operations, the recursion is $T(h) = h^2 \log p + 2T(h/2) + h^2$ so that the overall number of required field operations is $O(h^2 \log h \log p)$. A space complexity of $O(h)$ can be achieved by using an analogous iterative algorithm. □

E.5 A Canonical Embedding

We give a general purpose way of embedding a set into large enough finite extension fields. Crucial to us is the simple observation that if the set has cardinality that is a power of the characteristic of the field, then the image of the set is a *subspace* over the base field. The lemma also spells out the computational cost of performing and reversing this embedding.

Lemma E.20 (Canonical Embedding of Finite Sets into Finite Fields). *Let p be a prime, f a positive integer, and $\mathbb{F}$ a field extension of $\mathbb{F}_p$ of degree f, represented via an irreducible polynomial I with root x. Define $\Psi_f \colon \{0,1\}^f \to \mathbb{F}$ to be the function that maps a p-ary string $s_1 \cdots s_f$ to the element $\sum_{j=0}^{f-1} s_j \mathsf{x}^j$ in $\mathbb{F}$.*

Then, Ψ_f is a bijection from $\{0,1\}^f$ to $\mathbb{F}$, and we call Ψ_f the canonical embedding of $\{0,1\}^f$ into the finite field of degree f over $\mathbb{F}_p$.

In particular, for any set A with $|A| \leq p^r$ for some positive integer r not larger than f, Ψ_f injects A into $\mathbb{F}$. Furthermore, if $|A| = p^r$, then $\Psi_f(A)$ is a linear subset of $\mathbb{F}$ of dimension r over $\mathbb{F}_p$, specified by a basis $(1_{\mathbb{F}}, \mathsf{x}, \ldots, \mathsf{x}^{r-1})$.

Moreover, Ψ_f and Ψ_f^{-1} are efficiently computable: there exist linear-time algorithms Comp$\boldsymbol{\Psi}$ *and* Comp$\boldsymbol{\Psi}^{-1}$ *such that*

$$\Psi_f(s_1 \cdots s_f) = \textsc{Comp}\boldsymbol{\Psi}\big(1^f, s_1 \cdots s_f\big) \quad \textit{and} \quad \Psi_f^{-1}\left(\sum_{j=0}^{f-1} s_j \mathsf{x}^j\right) = \textsc{Comp}\boldsymbol{\Psi}^{-1}\left(1^f, \sum_{j=0}^{f-1} s_j \mathsf{x}^j\right) .$$

Alternatively: there exist Boolean circuit families $\{\textsc{Comp}\boldsymbol{\Psi}_f\}_{f \in \mathbb{N}}$ and $\{\textsc{Comp}\boldsymbol{\Psi}_f^{-1}\}_{f \in \mathbb{N}}$ for computing Ψ_f and Ψ_f^{-1}, and the circuits in both families have linear size, constant depth, and can be constructed in time linear in the circuit size.

Proof. That Ψ_f is a bijection is clear from its definition. If $|A| = p^r$, then

$$\Psi_f(A) = \left\{ \sum_{j=0}^{r-1} a_j \mathsf{x}^j \ : \ a_1, \ldots, a_{r-1} \in \{0, \ldots, p-1\} \right\} = \mathrm{span}_{\mathbb{F}_p}(1, \mathsf{x}, \ldots, \mathsf{x}^{r-1}) \ ,$$

so that $\Psi_f(A)$ is an r-dimensional linear subset of $\mathbb{F}$ over $\mathbb{F}_p$. Next, the algorithms that compute Ψ_f and Ψ_f^{-1} can be defined as follows:

> Comp$\boldsymbol{\Psi}(1^f, s_1 \cdots s_f) \equiv$
>
> 1. Compute $\alpha(\mathsf{x}) := \sum_{j=0}^{f-1} s_j \mathsf{x}^j$ in the finite field $\mathbb{F}$.
> 2. Output $\alpha(\mathsf{x})$.
>
> Comp$\boldsymbol{\Psi}^{-1}(1^f, \alpha(\mathsf{x})) \equiv$
>
> 1. For $i = 1, \ldots, f$: set s_i to be the coefficient of x^{i-1} in $\alpha(\mathsf{x})$.
> 2. Output $s_1 \cdots s_f$.

The corresponding circuit families can be easily deduced from the algorithms. □

E.6 Some Useful Families of Polynomials

We discuss here some (families of) polynomials, along with their complexities, that we shall find useful in our arithmetization constructions of Section 11.

We begin with *CMP polynomials*, which are direct arithmetizations of corresponding Boolean comparator circuits. For example, given a 3-bit string $\sigma = 100$, the σ-CMP polynomial over $\mathbb{F}$ is given by $\mathrm{CMP}_{\mathbb{F},\sigma}(x_1, x_2, x_3) = x_1(1_{\mathbb{F}} - x_2)(1_{\mathbb{F}} - x_3)$; for $\alpha_1(\mathsf{x}), \alpha_2(\mathsf{x}), \alpha_3(\mathsf{x}) \in \{0_{\mathbb{F}}, 1_{\mathbb{F}}\}$, $\mathrm{CMP}_{\mathbb{F},\sigma}(\alpha_1(\mathsf{x}), \alpha_2(\mathsf{x}), \alpha_3(\mathsf{x})) = 1_{\mathbb{F}}$ if and only if $\alpha_1(\mathsf{x}), \alpha_2(\mathsf{x}), \alpha_3(\mathsf{x})$ correspond to the bits of σ.

Definition E.21 (CMP Polynomial)**.** *Let $\mathbb{F}$ be a finite field, m a positive integer, and σ an m-bit string. The* **σ-CMP polynomial over** $\mathbb{F}$*, denoted $\mathrm{CMP}_{\mathbb{F},\sigma}$, is the m-variate polynomial over $\mathbb{F}$ defined by*

$$\mathrm{CMP}_{\mathbb{F},\sigma}(x_1, \ldots, x_m) := y_1^{(\sigma_1)} \cdots y_m^{(\sigma_m)} \ ,$$

where $y_i^{(\sigma_i)}$ is defined to be x_i if $\sigma_i = 1$ and $(1_{\mathbb{F}} - x_i)$ if $\sigma_i = 0$ for $i = 1, \ldots, m$. Note that $\mathrm{CMP}_{\mathbb{F},\sigma}$ is multilinear and, moreover, can be computed by a $\lceil \log(m) \rceil$-depth $\mathbb{F}$-arithmetic circuit with $m - \mathrm{weight}(\sigma)$ field subtractions and $m - 1$ field multiplications (and this circuit can be constructed in time that is linear in its size).

Next, we introduce *MUX polynomials* that, as the name suggests, are simply arithmetizations of corresponding Boolean MUX circuits. For example, the 2-bit multiplexer polynomial over $\mathbb{F}$ is the polynomial

$$\mathrm{MUX}_{\mathbb{F},2}\big(s_1, s_2, x_0, x_1, x_2, x_3\big) = (1 - s_1)(1 - s_2)x_0 + (1 - s_1)s_2x_1 + s_1(1 - s_2)x_2 + s_1s_2x_3 \ .$$

Definition E.22 (MUX Polynomial)**.** *Let $\mathbb{F}$ be a finite field and m a positive integer. The* **m-bit multiplexer polynomial over** $\mathbb{F}$*, denoted $\mathrm{MUX}_{\mathbb{F},m}$, is the $(m + 2^m)$-variate polynomial over $\mathbb{F}$ defined by*

$$\mathrm{MUX}_{\mathbb{F},m}\big(s_1, \ldots, s_m, x_0, \ldots, x_{(2^m-1)}\big) := \sum_{\sigma \in \{0,1\}^m} \mathrm{CMP}_{\mathbb{F},\sigma}(s_1, \ldots, s_m) \cdot x_\sigma \ .$$

Note that $\mathrm{MUX}_{\mathbb{F},m}$ is a multilinear homogeneous polynomial of degree $(m+1)$ and, moreover, can be computed by a $(\lceil \log(m) \rceil + 1 + m)$-depth $\mathbb{F}$-arithmetic circuit with m field subtractions, $2^m \cdot ((m-1)+1)$ field multiplications, and 2^m field additions; furthermore, this circuit can be constructed in time that is linear in its size by giving the input $(\mathbb{F}, m)$ to an algorithm FindMUX.

Another class of polynomials that we use are *alternator polynomials*; roughly, given two subsets $T, S \subseteq \mathbb{F}$ with $T \subseteq S$, we are interested in the polynomial that is equal to 1 on T but vanishes everywhere on $S - T$. Similarly to vanishing polynomials, when T and S are vector spaces, the corresponding alternator polynomial has a small arithmetic circuit that computes it (though, unlike in the case of subspace polynomials, the polynomial itself will not be sparse).

Theorem E.23. *Let $H \subseteq \mathbb{F}$ be a vector space of dimension h over the (smaller) field $\mathbb{B}$, and let $K \subseteq H$ be a vector space of dimension k over $\mathbb{B}$. Let $Y_{\mathbb{F},H,K} \colon \mathbb{F} \to \mathbb{F}$ the low-degree extension of the function over H that is equal to 1 over K but vanishes everywhere on $H - K$; we call $Y_{\mathbb{F},H,K}$ the* **alternator polynomial** *of K in H over $\mathbb{F}$.*

There exists a $O(\max\{k^2, (h-k)^2\} \log p)$-time $O(h \log p)$-space algorithm FindAlternator *that, on input (an irreducible polynomial representing) $\mathbb{F}$, a basis $(\mu_1, \ldots, \mu_h)$ for H, and a dimension k with $k \leq h$, computes a $O(h)$-size p^h-degree $\mathbb{F}$-arithmetic circuit $[Y_{\mathbb{F},H,K}]^{\mathsf{A}}$ that computes $Y_{\mathbb{F},H,K}$.*

Proof. If $K = H$, then the theorem trivially follows by letting $Y_{\mathbb{F},H,K}$ be the constant polynomial that is everywhere equal to 1.

So assume that $K \subsetneq H$ (so that $k < h$), in which case we argue as follows. Let $\mathcal{B}_K = (\mu_1, \ldots, \mu_k)$ be a basis for K, and let $\mathcal{B}_H = (\mu_1, \ldots, \mu_k, \mu_{k+1}, \ldots, \mu_h)$ be its completion to a basis for H. Let $Z_K(x) \in \mathbb{F}[x]$ be the vanishing polynomial for K (cf. Definition E.9), and recall that $Z_K(x)$ is $\mathbb{B}$-linear (cf. Claim E.10). Furthermore, by Lemma E.11, $Z_K(x)$ is sparse and its coefficients can be efficiently computed when given the basis $\mathcal{B}_K$; specifically, we know from Remark E.12 that there exists an algorithm FindSubspPoly that, on input $(\mathbb{F}, \mathcal{B}_K)$, in time $O(k^2 \log p) = \mathsf{poly}(k, \log p)$ outputs $c_0^{(K)}, \ldots, c_{k-1}^{(K)} \in \mathbb{F}$ such that

$$Z_K(x) = x^{p^k} + \sum_{i=0}^{k-1} c_i^{(K)} x^{p^i} \ .$$

For $i \in \{k+1, \ldots, h\}$, define $\nu_i := Z_K(\mu_i)$. Observe that $\nu_{k+1}, \ldots, \nu_h$ are linearly independent for, if not, then, using the $\mathbb{B}$-linearity of Z_K it is possible to show that Z_K has more than p^k roots, which is a contradiction because the degree of Z_K is only p^k.

So let L be the $(h-k)$-dimensional $\mathbb{B}$-linear subset spanned by $\nu_{k+1}, \ldots, \nu_h$, and let $Z_L(x) \in \mathbb{F}[x]$ be its vanishing polynomial. Again invoking Claim E.10 and then Lemma E.11, we deduce that $Z_L(x)$ has the form

$$Z_L(x) = x^{p^{h-k}} + \sum_{i=0}^{h-k-1} c_i^{(L)} x^{p^i} \quad ,$$

where the coefficients $c_0^{(L)}, \ldots, c_{h-k-1}^{(L)} \in \mathbb{F}$ may be computed in time $O((h-k)^2 \log p) = \mathsf{poly}(h-k, \log p)$ by the algorithm FindSubspPoly on input $(\mathbb{F}, \mathcal{B}_L)$.

Define the polynomial $P \in \mathbb{F}[x]$ as[15]

$$P(x) := \frac{Z_L(x)}{c_0^{(L)} x} = x^{p^{h-k}-1} + \sum_{i=0}^{h-k-1} \frac{c_i^{(L)}}{c_0^{(L)}} x^{p^i - 1} \quad .$$

Finally, define the polynomial $Q \in \mathbb{F}[x]$ as

$$Q(x) := P\big(Z_K(x)\big) = \big(Z_K(x)\big)^{p^{h-k}-1} + \sum_{i=0}^{h-k-1} \frac{c_i^{(L)}}{c_0^{(L)}} \big(Z_K(x)\big)^{p^i - 1} \quad .$$

We claim that $Y_{\mathbb{F},H,K}(x) = Q(x)$. Indeed,

$$\deg(Q) = \deg(Z_K) \cdot \deg(P) < \deg(Z_K) \cdot \deg(Z_L) = p^k \cdot p^{h-k} = p^h \quad .$$

Moreover,

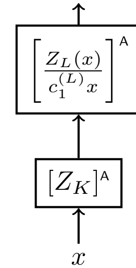
- for any $\alpha \in K$, $Q(\alpha) = P(Z_K(\alpha)) = P(0_{\mathbb{F}}) = c_0^{(L)}/c_0^{(L)} = 1_{\mathbb{F}}$; and
- for any $\alpha \in H - K$, then $Z_K(\alpha)$ is a non-zero element of L, so that $Z_L(\alpha) = 0$, and thus $Q(\alpha) = 0$.

So that indeed Q agrees on H with the function that is equal to 1 over K and is equal to 0 on $H - K$ and, moreover, is of degree at most $|H| = p^h$. Thus, by the uniqueness of low-degree extensions (cf. Theorem E.14), it must indeed be the case that $Y_{\mathbb{F},H,K}(x) = Q(x)$.

Now that we have an explicit form for $Y_{\mathbb{F},H,K}(x)$, we can say something about what kind of circuit is needed for computing $Y_{\mathbb{F},H,K}(x)$, and how fast such a circuit may be generated. Note that, unlike a subspace polynomial, $Y_{\mathbb{F},H,K}(x)$ is *not* sparse, because it involves raising a sum of terms (namely, $Z_K(x)$) to powers $p^i - 1$ for $i \in \{1, \ldots, h-k\}$, and $p^{h-k} - 1$ is large.[16] Nonetheless, given an element $\alpha \in \mathbb{F}$, $Y_{\mathbb{F},H,K}(\alpha)$ may be computed efficiently, by first computing $\beta := Z_K(\alpha)$, and then computing $P(\beta)$, because both Z_K and P are sparse. (And being able to compute the polynomial efficiently is all that we will need.) Thus, we can define the algorithm FindAlternator as follows:

FindAlternator$\big(\mathbb{F}, (\mu_1, \ldots, \mu_h), k\big) \equiv$

1. If $k = h$, output the constant circuit $1_{\mathbb{F}}$; otherwise continue.
2. Define $\mathcal{B}_K := (\mu_1, \ldots, \mu_k)$.
3. Compute $[Z_K]^{\mathsf{A}} :=$ FindSubspPoly$(\mathbb{F}, \mathcal{B}_K)$.
4. For $i = k+1, \ldots, h$, compute $\nu_i := Z_K(\mu_i)$.
5. Define $\mathcal{B}_L := (\nu_{k+1}, \ldots, \nu_h)$.
6. Compute $[Z_L]^{\mathsf{A}} :=$ FindSubspPoly$(\mathbb{F}, \mathcal{B}_L)$.
7. Compute $[\frac{Z_L(x)}{c_0^{(L)} x}]^{\mathsf{A}}$.
8. Compute $[Y_{\mathbb{F},H,K}]^{\mathsf{A}}$ as the composition of $[Z_K]^{\mathsf{A}}$ and $[\frac{Z_L(x)}{c_0^{(L)} x}]^{\mathsf{A}}$.
9. Output $[Y_{\mathbb{F},H,K}]^{\mathsf{A}}$.



Note that $[Y_{\mathbb{F},H,K}]^{\mathsf{A}}$ has size $O(h)$. Furthermore, the time complexity of FindAlternator is given by $O(k^2 \log p)+$

[15] Note that the coefficient of x in a subspace polynomial is never equal to zero. Indeed, suppose by way of contradiction that $Z_S(x)$ is a subspace polynomial where the coefficient of x is equal to zero. Then, $Z_S(x) = R(x^p) = (R'(x))^p$ for some linearized polynomials R and R'. Note that every root of $Z_S(x)$ is a root of $R'(x)$, but the degree of $R'(x)$ is smaller than the number of roots of $Z_S(x)$, which is a contradiction.

[16] Indeed, note that $(\alpha + \beta)^{p^i - 1} = \frac{1}{q-1} \prod_{j=0}^{i-1} (\alpha^{p^j} + \beta^{p^j})$ is a product of 2^i terms.

$O((h-k)k\log p)+O((h-k)^2\log p)$ and its space complexity is given by $O(h\log p)$. □

Finally, we discuss polynomials that represent *projection functions over linear subspaces.*

Definition E.24. *Let $H \subseteq \mathbb{F}$ be a vector space of dimension h over the base field $\mathbb{B}$, and consider the basis $\mathcal{B}_H = (1, \mathsf{x}, \ldots, \mathsf{x}^{h-1})$ for H. (Recall that x is a root of the irreducible polynomial I used to represent $\mathbb{F}$.) For $j \in \{1,\ldots,h\}$, define $p_{\mathbb{F},H,\mathbb{B},j} \colon H \to \mathbb{F}$ to be the function that, for any $\lambda = \sum_{i=1}^{h} \lambda_i \mathsf{x}^{i-1} \in H$, outputs $\lambda_j \in \mathbb{B}$.*

Let us first establish $\mathbb{B}$-linearity:

Lemma E.25. *Let $H \subseteq \mathbb{F}$ be a vector space of dimension h over the base field $\mathbb{B}$. Then, for every $j \in \{1,\ldots,h\}$, the function $p_{\mathbb{F},H,\mathbb{B},j}$ from Definition E.24 is $\mathbb{B}$-linear.*

Proof. Fix any two elements $\lambda = \sum_{i=1}^{h} \lambda_i \mathsf{x}^{i-1}$ and $\gamma = \sum_{i=1}^{h} \gamma_i \mathsf{x}^{i-1}$ in H. Then, for any two elements α and β in $\mathbb{B}$,

$$
\begin{aligned}
p_{\mathbb{F},H,\mathbb{B},j}(\alpha\lambda+\beta\gamma) &= p_{\mathbb{F},H,\mathbb{B},j}\left(\sum_{i=1}^{h}(\alpha\lambda_i+\beta\gamma_i)\mathsf{x}^{i-1}\right) \\
&= \alpha\lambda_j + \beta\gamma_j \\
&= \alpha \cdot p_{\mathbb{F},H,\mathbb{B},j}\left(\sum_{i=1}^{h}\lambda_i\mathsf{x}^{i-1}\right) + \beta \cdot p_{\mathbb{F},H,\mathbb{B},j}\left(\sum_{i=1}^{h}\gamma_i\mathsf{x}^{i-1}\right) \\
&= \alpha \cdot p_{\mathbb{F},H,\mathbb{B},j}(\lambda) + \beta \cdot p_{\mathbb{F},H,\mathbb{B},j}(\gamma) \ ,
\end{aligned}
$$

as desired. □

Because a projection function is $\mathbb{B}$-linear, we can express it as a sparse polynomial:

Lemma E.26. *Let $H \subseteq \mathbb{F}$ be a vector space of dimension h over the base field $\mathbb{B}$. Then, for every $j \in \{1,\ldots,h\}$, there exists a unique polynomial $\Pi_{\mathbb{F},H,\mathbb{B},j} \colon \mathbb{F} \to \mathbb{F}$ of the form*

$$
\Pi_{\mathbb{F},H,\mathbb{B},j}(x) := \sum_{i=0}^{h-1} \alpha_i x^{p^i} \ , \tag{32}
$$

where $\alpha_0,\ldots,\alpha_{h-1} \in \mathbb{F}$, such that $\Pi_{\mathbb{F},H,\mathbb{B},j}$ agrees on all of H with the function $p_{\mathbb{F},H,\mathbb{B},j}$ from Definition E.24; we call $\Pi_{\mathbb{F},H,\mathbb{B},j}$ the **projection polynomial** *for the j-th bit in H with respect to $\mathbb{B}$ over $\mathbb{F}$. Moreover, the coefficients $\alpha_0,\ldots,\alpha_{h-1}$ can be computed in $O(h^2\log h\cdot\log p) = \mathsf{poly}(h,\log p)$ time and $O(h)$ space.*

In particular, there exists a $\mathsf{poly}(h,\log p)$-time algorithm FIND**Π** *that, on input (an irreducible polynomial representing) $\mathbb{F}$, a basis $\mathcal{B}_H$ for H, and an index $j \in \{1,\ldots,h\}$, computes a $O(h\log p)$-size $\mathbb{F}$-arithmetic circuit $[\Pi_{\mathbb{F},H,\mathbb{B},j}]^{\mathsf{A}}$ that computes $\Pi_{\mathbb{F},H,\mathbb{B},j}$.*

Proof. Fix every $j \in \{1,\ldots,h\}$. By Lemma E.25, $p_{\mathbb{F},H,\mathbb{B},j}$ is $\mathbb{B}$-linear. Hence, by Claim E.7, the equality in Equation 32 follows. That the coefficients $\alpha_0,\ldots,\alpha_{h-1}$ may be computed in the claimed efficiency follows from Remark E.8 (and note that in this case, since $g = p_{\mathbb{F},H,\mathbb{B},j}$, $g(e_1),\ldots,g(e_h)$ can all be easily computed in $O(h^2)$ time because $e_i = \mathsf{x}^{i-1}$ and $g(e_j) = 1_{\mathbb{F}}$ and $g(e_j) = 0_{\mathbb{F}}$ for $i \neq j$). □

E.7 Efficient Algebraic Computation

Let $\mathbb{F}$ be an extension field of $\mathbb{F}_2 = \{0_{\mathbb{F}}, 1_{\mathbb{F}}\}$, H an h-dimensional linear subset of $\mathbb{F}$, and $\mathcal{B}_H = (e_1,\ldots,e_h)$ a basis for H.

Define $\mathrm{bit}\colon \mathbb{F} \to \{0,1\} \cup \{\perp\}$ to be the function that maps $\mathbb{F}_2$ to the two Boolean values and $\mathbb{F} - \mathbb{F}_2$ to $\{\perp\}$, that is, $\mathrm{bit}(0_{\mathbb{F}}) = 1$, $\mathrm{bit}(1_{\mathbb{F}}) = 1$, and $\mathrm{bit}(\alpha) = \perp$ for every $\alpha \in \mathbb{F} - \mathbb{F}_2$.

Define $\mathrm{bin}_{\mathcal{B}_H}\colon H \to \{0,1\}^h$ to be the function that gives the representation of elements in H in terms of the basis $\mathcal{B}_H$; that is, for any $\alpha = \sum_{j=1}^{h} \lambda_j e_j \in H$, with $\lambda_1,\ldots,\lambda_h \in \mathbb{F}_2$ and $e_1,\ldots,e_h \in \mathbb{F}$, it holds that $\mathrm{bin}_{\mathcal{B}_H}(\alpha) = (\mathrm{bit}(\lambda_1),\ldots,\mathrm{bit}(\lambda_h))$. Note that $\mathrm{bin}_{\mathcal{B}_H}$ is injective, so that referring to $\mathrm{bin}_{\mathcal{B}_H}$ as a representation of the elements in H (with respect to the basis $\mathcal{B}_H$) is indeed justified.

The function $\text{bin}_{\mathcal{B}_H}$ can naturally be extended to give the basis representation of elements in a *product* of linear subsets: for $i = 1, \ldots, m$, let H_i be an h_i-dimensional linear subset of $\mathbb{F}$, and let $\mathcal{B}_{H_i} = (e_1^{(i)}, \ldots, e_{h_i}^{(i)})$ be a basis for H_i; then the function

$$\text{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}} \colon H_1 \times \cdots \times H_m \to \{0,1\}^{h_1+\cdots+h_m}$$

is defined by

$$\text{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha_1, \ldots, \alpha_m) := \text{bin}_{\mathcal{B}_{H_1}}(\alpha_1) \circ \cdots \circ \text{bin}_{\mathcal{B}_{H_m}}(\alpha_m) \enspace ,$$

where $\circ$ is the string concatenation operator. Thus, it is natural to call $\text{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha_1, \ldots, \alpha_m)$ a *binary representation* of an element $(\alpha_1, \ldots, \alpha_m) \in H_1 \times \cdots \times H_m$.

Ben-Sasson and Sudan [BSGH+05, Theorem 5.5] showed that any small-depth, small-size, single-bit-output Boolean circuit operating on the binary representation of elements in a product space H^m, for some linear subset H of $\mathbb{F}$, can be converted into an equivalent arithmetic circuit of small size and moderate degree (exponential in the depth) over $\mathbb{F}$. (In particular, any bit of the binary representation of an element in H^m can be computed efficiently.)

An unfortunate yet seemingly inherent cost of arithmetizing Boolean circuits is that retrieving any single bit of a field element induces a polynomial of very high degree; thus we shall use this tool very sparingly in our reductions. Indeed, it is because of this cost that in Section 11, when arithmetizing SuccinctGCPs, we choose to "stripe" the bit of a color across many field elements instead of simply "packing" these in one field element and retrieve them later with a polynomial; we still have to invoke this result, though, to "know" where we are in the graph.

Here we prove a simple but useful generalization of [BSGH+05, Theorem 5.5] and take the opportunity to state its improved complexity in light of the fast algorithm of Section E.4.4. Specifically, we allow for the product space to consist of different (linear) subsets and we allow for the output of the Boolean circuit to consist of multiple bits. We also choose to state the theorem in terms of the multiplicative degree (see Section 6) of the Boolean circuit to be arithmetized, as multiplicative degree is in our setting a finer and more convenient complexity metric.

Theorem E.27 (Arithmetizing Boolean Circuits over Linear Spaces)**.** *Let $\mathbb{F}$ be an extension field of $\mathbb{F}_2$, m a positive integer, and, for $i = 1, \ldots, m$, H_i an h_i-dimensional linear subset of $\mathbb{F}$ with a basis $\mathcal{B}_{H_i} = (e_1^{(i)}, \ldots, e_{h_i}^{(i)})$.*

Fix a positive integer ρ. For any Boolean function $g \colon \{0,1\}^{h_1+\cdots+h_m} \to \{0,1\}^\rho$ computed by a Boolean circuit C of size S and (multiplicative) degree D, there exist ρ polynomials $\hat{g}_1, \ldots, \hat{g}_\rho \colon \mathbb{F}^m \to \mathbb{F}$ such that:

1. *For $i = 1, \ldots, m$ and $j = 1, \ldots, \rho$, the degree in the i-th variable of the j-th polynomial is*
$$\frac{|H_i|}{2} \max_{k - \sum_{r=0}^{i-1} h_r \in \{1,\ldots,h_i\}} \mathsf{D}[k \to j]$$
where we defined $h_0 := 0$;
2. *All of the polynomials are simultaneously computable by a multi-element-output $\mathbb{F}$-arithmetic circuit $\widehat{C}$ of size $O(\sum_{i=1}^m h_i^2 + \mathsf{S})$; and*
3. *For every $(\alpha_1, \ldots, \alpha_m)$ in $H_1 \times \cdots \times H_m$,*
$$\text{bit}\big(\hat{g}_1(\alpha_1, \ldots, \alpha_m)\big) \circ \cdots \circ \text{bit}\big(\hat{g}_\rho(\alpha_1, \ldots, \alpha_m)\big) = g\big(\text{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha_1, \ldots, \alpha_m)\big) \enspace .$$

Moreover, $\widehat{C}$ can be constructed in time $O(\sum_{l=1}^m h_l^3 \log h_l + \mathsf{S})$ and space $O(\sum_{i=1}^m h_i^2 + \mathsf{S})$ when given as input the basis $\mathcal{B}_{H_l}$ for each linear space H_l and C.

First, we prove that any bit of the binary representation of an element in $H_1 \times \cdots \times H_m$ can be computed efficiently; specifically, any individual bit, say a bit of an element in the subspace H_l, can be extracted by a polynomial of degree at most $2^{h_l - 1} = |H_l|/2$ that is computable by an $\mathbb{F}$-arithmetic circuit of size at most $O(h_l)$:

Lemma E.28. *Define $h_0 := 0$. Fix any $l \in \{1, \ldots, m\}$ and $\iota \in \{1, \ldots, h_l\}$, and consider the special case where g is the projection to the $(\iota + \sum_{i=0}^{l-1} h_i)$-th bit of the input. Then there exists a polynomial $\hat{g}_{l,\iota} \colon \mathbb{F}^m \to \mathbb{F}$ of degree at most $|H_l|/2$ computable by an $\mathbb{F}$-arithmetic circuit $\widehat{C}_{l,\iota}$ of size $O(h_l)$ such that, for every $(\alpha_1, \ldots, \alpha_m)$ in $H_1 \times \cdots \times H_m$,*

$$\text{bit}\big(\hat{g}_{l,\iota}(\alpha_1, \ldots, \alpha_m)\big) = g\big(\text{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha_1, \ldots, \alpha_m)\big) \enspace .$$

Moreover, given the basis $\mathcal{B}_{H_l} = (e_1^{(l)}, \ldots, e_{h_l}^{(l)})$, $\widehat{C}_{l,\iota}$ can be constructed in time $O(h_l^2 \log h_l)$.

Proof. Consider the function $\tilde{g} \colon H_1 \times \cdots \times H_m \to \mathbb{F}$ satisfying $(\text{bit} \circ \tilde{g}) = (g \circ \text{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}})$.[17] Observe that $(\text{bit} \circ \tilde{g})(H_1 \times \cdots \times H_m) = \{0,1\}$, so we deduce that $\tilde{g}(H_1 \times \cdots \times H_m) = \mathbb{F}_2$; moreover, for any two $\alpha = \sum_{i=1}^{m} \sum_{j=1}^{h_i} \lambda_j^{(i)} e_j^{(i)}$ and $\beta = \sum_{i=1}^{m} \sum_{j=1}^{h_i} \gamma_j^{(i)} e_j^{(i)}$ in $H_1 \times \cdots \times H_m$,

$$\begin{aligned}(\text{bit} \circ \tilde{g})(\alpha + \beta) &= g\big(\text{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha + \beta)\big) \\ &= \text{bit}(\lambda_\iota^{(l)}) + \text{bit}(\gamma_\iota^{(l)}) \\ &= g\big(\text{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha)\big) + g\big(\text{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\beta)\big) \\ &= (\text{bit} \circ \tilde{g})(\alpha) + (\text{bit} \circ \tilde{g})(\beta) \ ,\end{aligned}$$

implying that $\tilde{g}$ is a $\mathbb{F}_2$-linear map.

Invoking Claim E.7 with $\mathbb{F}$, $\mathbb{B} = \mathbb{F}_2$, $H = H_l$, $h = h_l$, and the (also $\mathbb{F}_2$-linear) function $\tilde{g}|_{H_l}$, we deduce that there exists a (unique) low-degree extension $\hat{g}'_{l,\iota} \colon \mathbb{F} \to \mathbb{F}$ of $\tilde{g}|_{H_l}$ that agrees with $\tilde{g}|_{H_l}$ on H_l and has the form $\hat{g}'_{l,\iota}(x_l) = \sum_{j=0}^{h_l-1} c_j x_l^{2^j}$, where $c_0, \ldots, c_{h_l-1} \in \mathbb{F}$ is a solution to the following system of h_l linear equations:

$$\left\{ \sum_{j=0}^{h_l-1} c_j \pi_j(e_s^{(l)}) = \tilde{g}|_{H_l}(e_s^{(l)}) \ : \ s = 1, \ldots, h_l \right\} \ , \tag{33}$$

where π_j is the mapping $x_l \mapsto x_l^{2^j}$. Given the basis $\mathcal{B}_{H_l} = (e_1^l, \ldots, e_{h_l}^{(l)})$, we can compute $\tilde{g}|_{H_l}(e_1^{(l)}), \ldots, \tilde{g}|_{H_l}(e_{h_l}^{(l)})$ in $O(h_l^2)$ time and then, by Remark E.8, compute the coefficients $c_0, \ldots, c_{h_l-1}$ in $O(h_l^2 \log h_l)$ time.

Therefore, $\hat{g}'_{l,\iota}$ can be computed by the $\mathbb{F}$-arithmetic circuit $\widehat{C}'_{l,\iota}$ of size $O(h_l)$ (computable in $O(h_l^2 \log h_l)$ time from $\mathcal{B}_{H_l}$) that, on input x_l, (1) computes the powers $x_l, x_l^2, \ldots, x_l^{2^{h_l-1}}$ (by repeated squaring); and (2) outputs the linear combination $\sum_{j=0}^{h_l-1} c_j x_l^{2^j}$.

To finish the proof of the lemma, we note that we can simply let $\widehat{C}$ be the circuit that works with that component of $\mathbb{F}^m$ in which the projected bit is present, and ignore the remaining components of $\mathbb{F}^m$; specifically, we let $\hat{g}_{l,\iota}(x_1, \ldots, x_m) := \hat{g}'_{l,\iota}(x_l)$ and $\widehat{C}_{l,\iota}(x_1, \ldots, x_m) := \widehat{C}'_{l,\iota}(x_l)$. ☐

The case for a general Boolean function g is obtained by constructing an arithmetic circuit that first extracts all the individual bits and then uses a straightforward arithmetization of the original Boolean circuit C:

Proof of Theorem E.27. For each $l \in \{1, \ldots, m\}$ and $\iota \in \{1, \ldots, h_l\}$, invoking Lemma E.28 with l and ι, we deduce that there exists a polynomial $\hat{g}_{l,\iota}$ of degree at most $|H_l|/2$ (which is computable by an $\mathbb{F}$-arithmetic circuit $\widehat{C}_{l,\iota}$ of $O(h_l)$ size) that agreees with $\pi_j^{(l)}$, the projection to the $(\iota + \sum_{i=0}^{l-1} h_i)$-th bit of the input, on $H_1 \times \cdots \times H_m$; moreover, $\widehat{C}_{l,\iota}$ can be constructed in $O(h_l^2 \log h_l)$ time from $\mathcal{B}_{H_l}$.

We deduce that there exists an $\mathbb{F}$-arithmetic circuit $\widehat{C}_{\text{bin}}$ of size $\sum_{i=1}^{m} h_i O(h_i) = O(\sum_{i=1}^{m} h_i^2)$ that extracts *all* the individual bits of the input: we define $\widehat{C}_{\text{bin}}$ to be the multi-output circuit that computes each $\widehat{C}_{l,\iota}$,

$$\widehat{C}_{\text{bin}}(x_1, \ldots, x_m) = \times_{l=1}^{m} \times_{\iota=1}^{h_l} \widehat{C}_{l,\iota}(x_1, \ldots, x_m) \ .$$

Note that $\widehat{C}_{\text{bin}}$ has $\sum_{l=1}^{m} h_l$ outputs, and, for $l \in \{1, \ldots, m\}$ and $\iota \in \{1, \ldots, h_l\}$, the $(\iota + \sum_{i=0}^{l-1} h_i)$-th output contains only variable x_l, with degree at most $|H_l|/2$. Moreover, from the definition of $\widehat{C}_{\text{bin}}$ and the constructibility of each $\widehat{C}_{l,\iota}$, we deduce that $\widehat{C}_{\text{bin}}$ can be constructed in time $\sum_{l=1}^{m} h_l \cdot O(h_l^2 \log h_l) = O(\sum_{l=1}^{m} h_l^3 \log h_l)$, when given as input the basis $\mathcal{B}_{H_l}$ for each linear space H_l.

Next, we argue that there exists a ρ-element-output $\mathbb{F}$-arithmetic circuit $\widehat{C}_g$ of size $O(\mathsf{S})$ that, on input the $\sum_{i=1}^{m} h_i$ individual bits[18] of an input in $H_1 \times \cdots \times H_m$, computes g and, moreover, the polynomial induced by $\widehat{C}_g$ has (multiplicative) degree (function) D and can be constructed in time $O(\mathsf{S})$ from C (which was the Boolean circuit of size s and depth d that computes g). Indeed, letting $\widehat{C}_g$ be the straightforward arithmetization of C does the job: for example, the arithmetization of the AND and NOT gate is performed as $\text{AND}(x, y) = x \cdot y$ and $\text{NOT}(x) = 1 - x$; the arithmetized gates compute the intended values whenever x and y take values in $\mathbb{F}_2$, which is the case.[19]

[17] Note that $\tilde{g}$ depends on $\mathcal{B}_{H_1}, \ldots, \mathcal{B}_{H_m}$, but we drop this subscript.

[18] Actually, bits represented as elements of $\mathbb{F}_2$.

[19] More generally, any binary gate $G(x, y)$ is replaced by an appropriate multilinear polynomial in x and y, which extends the corresponding mapping $\mathbb{F}_2 \times \mathbb{F}_2 \to \mathbb{F}_2$.

Finally, to finish the proof, we let $\widehat{C} = \widehat{C}_g \circ \widehat{C}_{\text{bin}}$, so that the overall size is $O(\sum_{i=1}^m h_i^2 + \mathsf{S})$. As for the multiplicative degree: letting $h_0 := 0$, for $i = 1, \ldots, m$ and $j = 1, \ldots, \rho$, the degree in the i-th variable of the j-th polynomial is

$$\frac{|H_i|}{2} \cdot \left(\max_{k - \sum_{r=0}^{i-1} h_r \in \{1, \ldots, h_i\}} \mathsf{D}[k \to j] \right) .$$

Moreover, from the constructibility of $\widehat{C}_g$ and $\widehat{C}_{\text{bin}}$, we deduce that $\widehat{C}$ can be constructed in time $O(\sum_{l=1}^m h_l^3 \log h_l + \mathsf{S})$ and space $O(\sum_{i=1}^m h_i^2 + \mathsf{S})$ when given as input the basis $\mathcal{B}_{H_l}$ for each linear space H_l and C. $\square$

References

[AF07] Masayuki Abe and Serge Fehr. Perfect nizk with adaptive soundness. In *TCC '07: Proceedings of the 4th Theory of Cryptography Conference on Theory of Cryptography*, pages 118–136, Berlin, Heidelberg, 2007. Springer-Verlag.

[AIK10] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: efficient verification via secure computation. In *ICALP '10: Proceedings of the 37th International Colloquium on Automata, Languages, and Programming*, pages 152–163, Berlin, Heidelberg, 2010. Springer-Verlag.

[AV77] Dana Angluin and Leslie G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. In *Proceedings on 9th Annual ACM Symposium on Theory of Computing*, STOC '77, pages 30–41, 1977.

[BA95] Amir M. Ben-Amram. What is a "pointer machine"? *SIGACT News*, 26:88–95, June 1995.

[BCCT11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. Cryptology ePrint Archive, Report 2011/443, 2011.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Composition and bootstrapping for SNARKs and proof-carrying data. Cryptology ePrint Archive, 2012.

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *STOC '91: Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 21–32, New York, NY, USA, 1991. ACM.

[Bha05] Arnab Bhattacharyya. Implementing probabilistically checkable proofs of proximity. Technical Report MIT-CSAIL-TR-2005-051, MIT, 2005. Available at `http://dspace.mit.edu/handle/1721.1/30562`.

[BSCGT12] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Towards practical PCPs, 2012. Electronic Colloquium on Computational Complexity.

[BSGH+04] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Robust PCPs of proximity, shorter PCPs and applications to coding. In *STOC '04: Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 1–10, New York, NY, USA, 2004. ACM. Full version available at `http://people.seas.harvard.edu/~salil/research/shortPCP.pdf`.

[BSGH+05] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Short PCPs verifiable in polylogarithmic time. In *CCC '05: Proceedings of the 20th Annual IEEE Conference on Computational Complexity*, pages 120–134, Washington, DC, USA, 2005. IEEE Computer Society.

[BSGH+06] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Robust PCPs of proximity, shorter PCPs, and applications to coding. *SIAM Journal on Computing*, 36(4):889–974, 2006. Preliminary versions of this paper have appeared in Proceedings of the 36th ACM Symposium on Theory of Computing and in Electronic Colloquium on Computational Complexity.

[BSS08] Eli Ben-Sasson and Madhu Sudan. Short PCPs with polylog query complexity. *SIAM Journal on Computing*, 38(2):551–607, 2008. Preliminary version appeared in STOC '05.

[CKLR11] Kai-Min Chung, Yael Kalai, Feng-Hao Liu, and Ran Raz. Memory delegation. In *Proceeding of the 31st Annual Cryptology Conference*, pages 151–168, 2011.

[CKV10] Kai-Min Chung, Yael Kalai, and Salil Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO '10: Proceedings of the 30th Annual International Cryptology Conference on Advances in Cryptology*, pages 483–501, Berlin, Heidelberg, 2010. Springer-Verlag. Full version online at `http://eprint.iacr.org/2010/241`.

[CR72] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*, STOC '72, pages 73–80, 1972.

[CRR11] Ran Canetti, Ben Riva, and Guy Rothblum. Two 1-round protocols for delegation of computation. Cryptology ePrint Archive, Report 2011/518, 2011.

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS '10: Proceedings of the 1st Symposium on Innovations in Computer Science*, pages 310–331, Beijing, China, 2010. Tsinghua University Press.

[CTY10] Graham Cormode, Justin Thaler, and Ke Yi. Verifying computations with streaming interactive proofs. ECCC, 2010. Available at `http://eccc.hpi-web.de/report/2010/159/`.

[DCL08] Giovanni Di Crescenzo and Helger Lipmaa. Succinct NP proofs from an extractability assumption. In *CiE '08: Logic and Theory of Algorithms, 4th Conference on Computability in Europe*, pages 175–185. Springer, 2008.

[DFH11] Ivan Damgård, Sebastian Faust, and Carmit Hazay. Secure two-party computation with low communication. Cryptology ePrint Archive, Report 2011/508, 2011.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *CRYPTO '10: Proceedings of the 30th Annual International Cryptology Conference on Advances in Cryptology*, pages 465–482, Berlin, Heidelberg, 2010. Springer-Verlag. `http://eprint.iacr.org/2009/547`.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *STOC '08: Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pages 113–122, New York, NY, USA, 2008. ACM.

[GL03] William F. Gilreath and Phillip A. Laplante. *Computer Architecture*. Kluwer Academic Publishers, 2003.

[GLR11] Shafi Goldwasser, Huijia Lin, and Aviad Rubinstein. Delegation of computation without rejection problem from designated verifier CS-proofs. Cryptology ePrint Archive, Report 2011/456, 2011.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43:431–473, May 1996.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '10, pages 321–340, 2010.

[GS89] Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Logic at Botik '89, Symposium on Logical Foundations of Computer Science*, pages 108–118, 1989.

[Gur88] Yuri Gurevich. On kolmogorov machines and related issues. *Bulletin of the European Association for Theoretical Computer Science*, 35:71–82, 1988.

[Har04] Prahladh Harsha. *Robust PCPs of Proximity and Shorter PCPs*. PhD thesis, MIT, EECS, September 2004.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[IKO07] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short PCPs. In *CCC '07: Proceedings of the Twenty-Second Annual IEEE Conference on Computational Complexity*, pages 278–291, Washington, DC, USA, 2007. IEEE Computer Society.

[IO11] Yuval Ishai and Rafail Ostrovsky. Linear interactive proofs and the complexity of verification, 2011. Unpublished manuscript.

[Jon88] Douglas W. Jones. The ultimate RISC. *ACM SIGARCH Computer Architecture News*, 16:48–55, June 1988.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *STOC '92: Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 723–732, New York, NY, USA, 1992. ACM.

[Kol53] Andrey N. Kolmogorov. To the definition of an algorithm. *Uspekhi Matematicheskikh Nauk*, 8(4):175–176, 1953.

[Kop10] Swastik Kopparty. Private communication, 2010.

[KU58] Andrey N. Kolmogorov and Vladimir A. Uspenskiĭ. To the definition of an algorithm. *Uspekhi Matematicheskikh Nauk*, 13(4):3–28, 1958. In Russian. English translation in in AMS Translations, ser. 2, vol. 21 (1963), 217Ð-245.

[Lei92] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[LN97] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. Cambridge University Press, Cambridge, UK, second edition edition, 1997.

[Mat08] Todd Mateer. *Fast Fourier Transform algorithms with applications*. PhD thesis, Clemson University, 2008.

[Mer89] Ralph C. Merkle. A certified digital signature. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference*, pages 218–238, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. Preliminary version appeared in FOCS '94.

[NRS94] Ashish V. Naik, Kenneth W. Regan, and D. Sivakumar. On quasilinear-time complexity theory. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '94, pages 325–349, 1994.

[Ofm65] Yuri P. Ofman. A universal automaton. *Transactions of the Moscow Mathematical Society*, 14:200–215, 1965.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, USA, 1994.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *Journal of the ACM*, 26:361–381, April 1979.

[PS94] Alexander Polishchuk and Daniel A. Spielman. Nearly-linear size holographic proofs. In *STOC '94: Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 194–203, New York, NY, USA, 1994. ACM.

[Rob91] J. M. Robson. An O(T log T) reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, May 1991.

[Sch78] Claus-Peter Schnorr. Satisfiability is quasilinear complete in NQL. *Journal of the ACM*, 25:136–145, January 1978.

[Sch80] Arnold Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, 1980.

[Sho88] Victor Shoup. New algorithms for finding irreducible polynomials over finite fields. In *FOCS '88: Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 283–290, Los Alamitos, CA, USA, 1988. IEEE Computer Society.

[Sho99] Victor Shoup. Efficient computation of minimal polynomials in algebraic extensions of finite fields. In *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation*, ISSAC '99, pages 53–58, New York, NY, USA, 1999. ACM.

[Shp96] Igor Shparlinski. On finding primitive roots in finite fields. *Theoretical Computer Science*, 157(2):273–275, 1996.

[Spi95] Daniel Spielman. *Computationally Efficient Error-Correcting Codes and Holographic Proofs*. PhD thesis, MIT, Mathamatics Department, May 1995.

[vzGG96] Joachim von zur Gathen and Jürgen Gerhard. Arithmetic and factorization of polynomial over f_2. In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, ISSAC '96, pages 1–9, New York, NY, USA, 1996. ACM.

[vzGG03] Joachim von zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.