A Fiat–Shamir Transformation From Duplex Sponges

Alessandro Chiesa
`alessandro.chiesa@epfl.ch`
EPFL

Michele Orrù
`m@orru.net`
CNRS

Abstract

The Fiat–Shamir transformation underlies numerous non-interactive arguments, with variants that differ in important ways. This paper addresses a gap between variants analyzed by theoreticians and variants implemented (and deployed) by practitioners. Specifically, theoretical analyses typically assume parties have access to random oracles with sufficiently large input and output size, while cryptographic hash functions in practice have fixed input and output sizes (pushing practitioners towards other variants).

In this paper we propose and analyze a variant of the Fiat–Shamir transformation that is based on an ideal permutation of fixed size. The transformation relies on the popular duplex sponge paradigm, and minimizes the number of calls to the permutation (given the amount of information to absorb and to squeeze). Our variant closely models deployed variants of the Fiat–Shamir transformation, and our analysis provides concrete security bounds that can be used to set security parameters in practice.

We additionally contribute `spongefish`, an open-source Rust library implementing our Fiat–Shamir transformation. The library is interoperable across multiple cryptographic frameworks, and works with any choice of permutation. The library comes equipped with Keccak and Poseidon permutations, as well as several "codecs" for re-mapping prover and verifier messages to the permutation's domain.

Keywords: Fiat–Shamir transformation; duplex sponge

Contents

1 Introduction **3**
1.1 Our results . . . 4
1.2 Application: a BCS transformation from duplex sponges . . . 6
1.3 Open problems . . . 7
1.4 Related work . . . 7

2 Techniques **9**
2.1 Review: starting point and the basic variant . . . 9
2.2 DSFS: a Fiat–Shamir transformation based on the ideal duplex sponge . . . 10
2.3 Soundness and knowledge soundness . . . 12
2.4 Zero knowledge . . . 15
2.5 Application: BCS transformation from a duplex sponge . . . 16

3 Preliminaries **17**
3.1 Notation . . . 17
3.2 Basics . . . 17
3.3 Duplex sponge in overwrite mode . . . 18
3.4 Non-interactive arguments in oracle models . . . 19
3.5 Interactive proofs . . . 21
3.6 Fiat–Shamir transformation . . . 23

4 Duplex-sponge Fiat–Shamir transformation **27**

5 Security analysis **29**
5.1 Backtracking procedure . . . 30
5.2 Lookahead procedure . . . 31
5.3 Prover transformation . . . 32
5.4 Trace transformation . . . 34
5.5 Analysis of aborts . . . 35
5.6 Proof of Lemma 5.1 . . . 41

6 Soundness and knowledge soundness **46**
6.1 Proof of Theorem 6.1 . . . 47
6.2 Proof of Theorem 6.2 . . . 47

7 Zero knowledge **50**
7.1 Definitions for zero knowledge . . . 50
7.2 The simulator . . . 51
7.3 Proof of Theorem 7.1 . . . 51

8 Implementation **59**
8.1 Core library and software stack . . . 59
8.2 Concrete security and ergonomics . . . 60

A Example: codecs for Schnorr's protocol **62**

B Bias of modular reduction **63**

Acknowledgments **64**

References **64**

1 Introduction

The Fiat–Shamir transformation [FS86] is a technique that uses a hash function to convert a public-coin interactive protocol between a prover and a verifier into a corresponding non-interactive protocol. The beautiful idea underlying the transformation is that one can replace the verifier's random messages with suitable outputs of the hash function (if sufficiently "secure"), eliminating the need for interaction. This technique, in various incarnations, has numerous and diverse applications across cryptography.

The Fiat–Shamir transformation is often studied in the *random oracle model* (ROM), where the hash function is modeled as a random function (every input is mapped to a random output). This is setting is an idealization, as any hash function that one would use in the real world would not be a random function. Nevertheless, this idealization provides an elegant model where security can be precisely quantified and then, in practice, one replaces the random oracle with a "random-looking" hash function such as SHA256, making the heuristic assumption that this replacement does not compromise security. This is an example of the *random oracle methodology* that, despite suffering from notable limitations,[1] has had an immense impact on the construction and analysis of highly-efficient cryptographic primitives for practical use.

In this paper *we propose and analyze a variant of the Fiat–Shamir transformation, aimed at closely modeling deployed variants of the Fiat–Shamir transformation*. We motivate this in the next few paragraphs, highlighting why prior work does not adequately capture features of practical interest.

Different shades of Fiat–Shamir. "Fiat–Shamir transformation" is an umbrella term that refers to a class of *related but distinct* transformations that apply to different classes of interactive protocols. Differences arise due to round complexity (one verifier message or multiple verifier messages), salts for zero knowledge (how many and where they are introduced), and oracle setting (which random oracles are available to parties). We review the main variants in Section 2.

A security gap. The motivation for this paper is a gap between variants analyzed by theoreticians and variants implemented (and deployed) by practitioners. On the one hand, theoreticians assume access to a random oracle (or multiple random oracles) *with sufficiently large input and output size*. For example, one assumes that a prover message (or even all prover messages up to a certain round) can be an input to the random oracle. On the other hand, cryptographic hash functions in practice *act over blocks of fixed length*, so practitioners consider (more complex) variants that process sufficiently large inputs via multiple calls to the fixed-size hash function. These variants *have not been analyzed in the appropriate oracle models* (where parties have access to, say, a random oracle with fixed input size). This open security gap should be closed (or at least reduced) in light of the widespread adoption in practice of the Fiat–Shamir transformation (e.g., in zkSNARK constructions).

Permutation-based cryptography. Permutation-based cryptography has developed into a viable alternative to traditional block-cipher-based cryptography, leading to simple and diverse constructions now used in practice. These include duplex sponges for lightweight authenticated encryption [BDPVA12], eXtensible Output Functions (XOFs) [MF21], deck functions such as Farfalle for high-speed authenticated encryption [BDHPVAVK18], compression modes such as Jive [Bou+23], and more. Most notably, the SHA-3 standard [Sha] based on Keccak is an example of a hash function designed according to this methodology: it is a duplex sponge that enables "absorbing" arbitrary-length inputs and "squeeze" arbitrary-length outputs.

[1] For example, there are interactive protocols for which the Fiat–Shamir transformation yields a secure protocol if the hash function is a random oracle but not if the hash function is any efficient function [Bar01; GK03; CGH04; BBHMR19; KRS25].

1.1 Our results

We propose and analyze the security of a Fiat–Shamir transformation **based on a duplex sponge**, aimed at closely modeling deployed variants of the Fiat–Shamir transformation. We accompany our analysis with a flexible Rust library that enables practitioners to conveniently and efficiently use our variant. We elaborate on our results below, and then in Section 2 we summarize the ideas for proving our results.

The ideal permutation model. We consider a setting where parties have access to a random permutation $p\colon \Sigma^{r+c} \to \Sigma^{r+c}$ where Σ is a finite alphabet, c is the *capacity*, r is the *rate*, and $r + c$ is the *permutation length*.[2] These parameters are fixed (independent of any specific interactive protocol). In practice, p would be heuristically instantiated by a binary permutation (e.g., Keccak [Sha]) or an "algebraic" permutation (e.g., MiMC [AGRRT16], Poseidon [GKRRS21], Anemoi [Bou+23]). This is the aforementioned setting of permutation-based cryptography. In particular, p directly gives rise to a corresponding duplex sponge construction [BDPVA12], a mode of operation that enables absorbing and squeezing strings over Σ.

Our transformation. We transform a given public-coin interactive proof $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ for a relation $\mathcal{R}$ into a corresponding non-interactive argument $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ for $\mathcal{R}$, in the ideal permutation model described above. The transformation additionally depends on a *codec* cdc that bridges the (typically different) representation of IP's messages and p's alphabet Σ: cdc specifies maps $(\varphi_i)_{i\in[\mathsf{k}]}$ to encode prover messages into strings over Σ and maps $(\psi_i)_{i\in[\mathsf{k}]}$ to decode strings over Σ into verifier messages. The transformation hardcodes a salt size $\delta \in \mathbb{N}$ used to ensure (preservation of) zero knowledge. Overall, our transformation is a function $\mathbf{DSFS}$ ("duplex-sponge Fiat–Shamir") of the following form:

$$\mathsf{NARG} := \mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta] \,.$$

We outline the transformation in Section 2, and formally describe it in Section 4. Briefly, the permutation p is used to absorb a random salt in Σ^{δ} and then to alternately absorb prover messages and squeeze verifier messages (suitably relying on the encoding maps and decoding maps in the codec cdc of the given IP).

The transformation *minimizes the number of calls to the given permutation* p. Specifically, when using p in a duplex sponge, each call to p absorbs r elements of Σ or squeezes r elements of Σ. For every $i \in [\mathsf{k}]$, let $\ell_{\mathbf{P}}(i)$ and $\ell_{\mathbf{V}}(i)$ be the length of prover and verifier messages for round i when represented over Σ according to the codec's maps. Then (up to rounding effects that we ignore here):

- the minimum number of calls to p to absorb all the relevant information is $\frac{1}{r} \cdot (\delta + \sum_{i\in[\mathsf{k}]} \ell_{\mathbf{P}}(i))$, and
- the minimum number of calls to p to squeeze all the relevant information is $\frac{1}{r} \cdot \sum_{i\in[\mathsf{k}]} \ell_{\mathbf{V}}(i)$.

This is essentially what our transformation achieves.

One can view $\mathbf{DSFS}$ as a "cleaned up" variant of several deployed transformation variants. These include: (i) Trail of Bit's Decree [Wri], internally relying on Merlin [Val] which in turn is based on the STROBE protocol framework [Ham17]; (ii) POKSHO [Sig], the library used in Signal for the generation of credentials in group chats; and (iii) ad-hoc, non-interoperable implementations used in the wild, such as the ones from arkworks.rs [Ark], Aztec [Azt], Microsoft Research [Set], StarkWare [Sta], and Zcash [BLHG].

Soundness and knowledge soundness. The soundness (resp., knowledge soundness) of NARG is, as with other Fiat–Shamir transformations, primarily determined by the state-restoration soundness (resp., knowledge soundness) of IP; see [CY24] for discussions on why state-restoration security intuitively captures the security of the technique underlying Fiat–Shamir transformations. Here it suffices to note that state-restoration soundness is implied by sufficiently strong notions of soundness such as round-by-round soundness and special soundness (and similarly for state-restoration knowledge soundness); see [CY24] (and [AFK22]).

[2] In security analyses, adversaries also have access to p^{-1}, as in practice permutations are not designed to be hard to invert.

The informal theorem below puts this in quantitative terms. The take-away is that security against t-query adversaries is determined by: (i) the state-restoration security of IP against t-move adversaries; plus (ii) a term $\frac{4t^2+\max_{i\in[\mathsf{k}]}\lceil \ell_{\mathbf{P}}(i)/r\rceil\left(t+\max_{i\in[\mathsf{k}]}\ell_{\mathbf{P}}(i)/r\right)}{|\Sigma|^c}$ that represents the security of a duplex sponge based on the ideal permutation $p\colon \Sigma^{r+c}\to\Sigma^{r+c}$. (There is also another additive error term, relevant in practice, that is incurred due to decoding biases. We ignore this term in the informal theorem, and postpone its discussion till further below.)

Theorem 1 (informal)**.** *If* IP *has state-restoration soundness error* $\varepsilon^{\mathrm{sr}}_{\mathsf{IP}}$ *then* NARG *has soundness error* $\varepsilon_{\mathsf{NARG}}$ *such that*

$$\varepsilon_{\mathsf{NARG}}(t)\le\varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(t)+\frac{2t^2}{|\Sigma|^c}\,.$$

Moreover, a similar statement holds for knowledge soundness: if IP *has state-restoration knowledge soundness error* $\kappa^{\mathrm{sr}}_{\mathsf{IP}}$ *then* NARG *has knowledge soundness error* κ_{NARG} *such that*

$$\kappa_{\mathsf{NARG}}(t)\le\kappa^{\mathrm{sr}}_{\mathsf{IP}}(t)+\frac{2t^2}{|\Sigma|^c}\,.$$

We outline the ideas underlying the theorem in Section 2.3, and formally state and prove it in Section 6.

Our analysis in fact shows better upper bounds than stated above: the state-restoration terms $\varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(t)$ and $\kappa^{\mathrm{sr}}_{\mathsf{IP}}(t)$ can be replaced by the better terms $\varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(\theta_\star(t))$ and $\kappa^{\mathrm{sr}}_{\mathsf{IP}}(\theta_\star(t))$ with the smaller move budget $\theta_\star(t)\approx t/\min_{i\in[\mathsf{k}]}\frac{\ell_{\mathbf{P}}(i)}{r}$. Reducing t by a multiplicative "scale-down" factor represents the (intuitive) fact that one needs $\frac{\ell_{\mathbf{P}}(i)}{r}$ queries to the permutation p to fully absorb an encoded prover message in $\Sigma^{\ell_{\mathbf{P}}(i)}$.

Zero knowledge. The zero-knowledge error of NARG is, as with other Fiat–Shamir transformations, primarily determined by the honest-verifier zero-knowledge error of IP. The main difference compared to prior variants is the fact that we use only *a single random salt in* Σ^δ (independent of the number of rounds). Establishing zero knowledge of NARG with this improvement demands a significantly more delicate analysis.

Specifically, the distinguishing advantage against t-query adversaries is the sum of: (i) the honest-verifier zero-knowledge error of IP; (ii) a term $\frac{t}{|\Sigma|^{\min(\delta,c)}}$ representing the probability of guessing the salt; and (iii) a term $\frac{t\cdot\sum_{i\in[\mathsf{k}]}\lceil \ell_{\mathbf{V}}(i)/r\rceil}{|\Sigma|^{r+c}}$ representing the probability of guessing intermediate permutation states while squeezing. (Similar to Theorem 1, there is an additional error term due to decoding biases, which we discuss later.)

Theorem 2 (informal)**.** *If* IP *has honest-verifier zero-knowledge error* z_{IP} *then* NARG *has adaptive zero-knowledge error* z_{NARG} *such that*

$$z_{\mathsf{NARG}}(t)\le z_{\mathsf{IP}}+\frac{t}{|\Sigma|^{\min(\delta,c)}}+\frac{t\cdot\sum_{i\in[\mathsf{k}]}\lceil \ell_{\mathbf{V}}(i)/r\rceil}{|\Sigma|^{r+c}}\,.$$

We outline the ideas underlying the theorem in Section 2.4, and formally state and prove it in Section 7.

Decoding biases. All our analyses explicitly handle an additive error term coming from a mismatch that often arises in practice (and must be accounted for).

A codec for IP abstracts the process of encoding prover messages in IP to strings over Σ (to provide as input to p) and decoding verifier messages in IP from strings over Σ (obtained as outputs from p). For the decoding, *the distribution matters*: since IP is public coin, the verifier message for round $i\in[\mathsf{k}]$ is a uniform random element ρ_i sampled from a message space $\mathcal{M}_{\mathbf{V},i}$; in contrast, in the Fiat–Shamir transformation, the

verifier message is obtained as the decoding $\rho_i := \psi_i(\widehat{\boldsymbol{\rho}}_i)$ of a uniform random string $\widehat{\boldsymbol{\rho}}_i$ in $\Sigma^{\ell_{\mathbf{V}}(i)}$ (up to bad events that are already captured by the other error terms).

However, many decoding maps *do not preserve the uniform distribution*. For example, if $\Sigma = \{0, 1\}$ and $\ell_{\mathbf{V}}(i) = 100$ and $\mathcal{M}_{\mathbf{V},i}$ is a 100-bit prime field then every map from $\Sigma^{\ell_{\mathbf{V}}(i)}$ to $\mathcal{M}_{\mathbf{V},i}$ introduces some bias (i.e., does not map the uniform distribution on $\Sigma^{\ell_{\mathbf{V}}(i)}$ to the uniform distribution on $\mathcal{M}_{\mathbf{V},i}$).

We account for this by explicitly tracking decoding biases. We say that a codec has bias $\varepsilon_{\mathsf{cdc}}$ if, for every $i \in [\mathsf{k}]$, $\psi_i \colon \Sigma^{\ell_{\mathbf{V}}(i)} \to \mathcal{M}_{\mathbf{V},i}$ is a $\varepsilon_{\mathsf{cdc},i}$-biased map ($\psi_i$ maps the uniform distribution on $\Sigma^{\ell_{\mathbf{V}}(i)}$ to a distribution that is $\varepsilon_{\mathsf{cdc},i}$-close to the uniform distribution on $\mathcal{M}_{\mathbf{V},i}$). Our analyses show that the total bias

$$\sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}$$

appears as an additional additive error in Theorem 1 and Theorem 2. The takeaway is that in practice one must ensure that the total bias (for the chosen codec) is also suitably small.

To aid practitioners, we provide basic lemmas for establishing the bias of common decoding maps.

Implementation. We implement and evaluate our transform **DSFS** in Rust, and make it open-source under a BSD license. Our library, called `spongefish` (duplex sponge Fiat–Shamir), is freely available at `https://github.com/arkworks-rs/spongefish`. Features that stand out compared to other deployed variants of the Fiat–Shamir transformation include:

- **Any permutation.** The library supports arbitrary permutation functions over any alphabet. With the library, we include two permutation functions, `keccak` [Sha] and `poseidon` [GKRRS21], plus byte challenges with grinding (a proof of work integrated within the verifier messages).
- **Built-in codecs.** The library includes common encoding maps and decoding maps. We provide encoding maps for absorbing prover messages from field and elliptic curve points. We support traits defined in `arkworks-rs/algebra` and `zkcrypto/group` (two popular Rust frameworks for argument systems). The non-interactive argument string is interoperable across the two frameworks. We provide decoding maps, with negligible biases, for converting byte hash outputs into field elements, and converting field elements hash outputs into bytes (taking into account Lemma B.1). This avoids manual (and error-prone) conversions that are typically seen in the wild.

We elaborate on our library in Section 8. At present, our library does not target lattice-based argument systems, which often require rejection sampling and sampling from discrete Gaussian or Bernoulli distributions.

1.2 Application: a BCS transformation from duplex sponges

The Fiat–Shamir transformation is a building block of the BCS transformation [BCS16], which transforms a public-coin interactive oracle proof (IOP) into a corresponding non-interactive argument in the ROM. The BCS transformation is widely used in practice, enabling the construction of highly-efficient post-quantum zkSNARKs (e.g., zkSTARKs).

Our transformation can directly replace the relevant building block in the BCS transformation. This modified BCS transformation inherits the efficiency advantages of our transformation, benefitting from an efficient use of the underlying random permutation.

We elaborate on this application in Section 2.5.

1.3 Open problems

Post-quantum security. Establishing the security of **DSFS** against superposition queries to the permutation function remains a challenging open question. Such a result falls in an exciting line of work that establishes the (unconditional) security of notable non-interactive arguments in the QROM: this includes the Fiat–Shamir transformation for sigma protocols [DFMS19; LZ19], and the BCS transformation (which includes the multi-round Fiat–Shamir transformation [CMS19]). Such results are especially valuable given that, in general, security of a non-interactive argument in the ROM does *not* imply security in the QROM [YZ21].

UC security. In this paper we target (adaptive variants of) soundness, knowledge soundness, and zero-knowledge for single instances; this is for simplicity and should be taken as a valuable starting point. Applications may demand stronger variants, for example: security notions that consider multiple adaptively chosen instances; non-malleability; simulation-secure soundness; simulation-secure knowledge soundness; and so on. The gold standard, which in particular implies all of these security properties of "intermediate" strength, is *universally composable security* (UC security) in an appropriate global model (such as the global random oracle model [CJS14; CDGLN18]). The BCS transformation, which includes as a building block the Fiat–Shamir transformation, does satisfy UC security [CF24]. Establishing the UC security of **DSFS** within an appropriate context remains an important open question.

Other FS variants. Variants of the Fiat–Shamir transformation are typically proved secure via a security reduction to the "basic variant" (which we happen to discuss in Section 2.1). Examples of such reductions include those for the hash-chain variant in [CY24] and for the duplex-sponge variant in this paper. In practice one may need to modify these variants or consider yet other variants, which would require adapting a rather delicate argument to account for these modifications. Superficially, the notion of *indifferentiability* [MRH04] may help in reducing some of this recurring work, by proving that modifications are indifferentiable from the reference oracles. The notion of indifferentiability does not, in fact, suffice as is; nevertheless, we conjecture that a stronger notion of indifferentiability may suffice; see the discussion in Remark 2.2.

1.4 Related work

This paper is about the security of **DSFS**, a variant of the Fiat–Shamir transformation, in the ideal permutation model. In Section 2.1 we review prior variants that have been studied by theoreticians for other oracle models (i.e., variants in oracle models with a security analysis). Below we summarize two types of other prior work: (1) research on the Fiat–Shamir transformation in the plain model (no oracles); (2) implementations of the Fiat–Shamir transformation (without security analyses).

(1) Fiat–Shamir in the plain model. An oracle model is merely an idealization. A concrete (efficient) function replaces the oracle in the real world. Doing this securely is delicate, as there are examples of interactive protocols for which no efficient function can securely replace the oracle in the Fiat–Shamir transformation [GK03; CGH04; BBHMR19; KRS25]. On the other hand, there are classes of interactive protocols for which there is a way to securely instantiate a random oracle. Most notably the Fiat–Shamir transformation is secure in the plain model for IPs that are round-by-round sound when the oracle is replaced by *correlation-intractable hash functions* (for a suitable relation) [CCHLRR18]. This has led to a beautiful line of work exploring how to construct such hash functions and to understand for which IPs the Fiat–Shamir transformation "works" [CCR16; KRR17; CCRR18; HL18; CCHLRR18; Can+19; PS19; BKM20; JKKZ21; JJ21; HLR21; KLV23; CGJJZ23]. Separately, the Fiat–Shamir transformation can be extended with a suitable proof-of-work, which rules out many problematic cases [AY25].

(2) Fiat–Shamir in the wild. Differing variants of the Fiat–Shamir transformation can be found in the

wild. The STROBE [Ham17] framework is used by several implementations [Val; Ark; Wri], and internally relies on a fixed (not changeable) duplex binary sponge. Akworks [ark] supports arbitrary sponges (to be used in duplex mode). Some projects internally rely on compression functions, but expose an "absorb" and "squeeze" interface mimicking duplex sponges: Signal [Sig] internally relies on SHA-2, and Halo2 from Zcash [BLHG] internally relies on BLAKE2. Some IETF standards [DFHSW; CKGW; Hao; LKWL] perform the Fiat–Shamir transformation, each in a slightly different way, and targeting a specific set of binary hashes; no generalized approach or shared design structure is affirmed in the standards community.

2 Techniques

We outline the main ideas underlying our results. In Section 2.1 we review the basic variants of the Fiat–Shamir transformation and their limitations. In Section 2.2 we discuss our variant. In Section 2.3 we outline the proof of Theorem 1 (soundness and knowledge soundness) and Section 2.4 we outline the proof of Theorem 2 (zero knowledge). Finally, in Section 2.5 we discuss how our variant can be applied to the BCS transformation.

Throughout, we let $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ be a public-coin IP for a relation $\mathcal{R}$ with round complexity k. We denote by $(\mathcal{M}_{\mathbf{P},i})_{i\in[\mathsf{k}]}$ and $(\mathcal{M}_{\mathbf{V},i})_{i\in[\mathsf{k}]}$ the message spaces of prover messages and of verifier messages.

2.1 Review: starting point and the basic variant

A Fiat–Shamir transformation maps $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ to a non-interactive argument $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$, for the same relation $\mathcal{R}$, in a certain oracle model. Different transformation variants share a similar template.

- *Oracle setting.* A variant specifies the distribution $\mathcal{D}$ from which a list of oracles $\boldsymbol{f}$ is sampled.
- *Argument prover.* On input an instance $\mathbb{x}$ and witness $\mathbb{w}$, the argument prover $\mathcal{P}$ queries the oracles $\boldsymbol{f}$ and outputs an argument string π that includes IP prover messages $(\alpha_i \in \mathcal{M}_{\mathbf{P},i})_{i\in[\mathsf{k}]}$ (as well as some salts for the purpose of zero knowledge as we discuss in Section 2.4). Prover messages are obtained by emulating an interaction between the IP prover $\mathbf{P}(\mathbb{x}, \mathbb{w})$ and an imaginary IP verifier by somehow deriving each verifier message $\rho_i \in \mathcal{M}_{\mathbf{V},i}$ via queries to the oracles $\boldsymbol{f}$.
- *Argument verifier.* On input an instance $\mathbb{x}$ and argument string π, the argument verifier $\mathcal{V}$ queries the oracles $\boldsymbol{f}$ and outputs a decision bit. The decision corresponds to whether $\mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}\big)$ accepts, where $(\alpha_i)_{i\in[\mathsf{k}]}$ are the IP prover messages in π and $(\rho_i)_{i\in[\mathsf{k}]}$ are the IP verifier messages derived, using the oracles $\boldsymbol{f}$, the same way as the (honest) argument prover $\mathcal{P}$.

Variants of the Fiat–Shamir transformation differ in two main features:

- the choice of oracle distribution $\mathcal{D}$; and
- how IP verifier messages $(\rho_i)_{i\in[\mathsf{k}]}$ are derived.

We summarize the main variants of the Fiat–Shamir transformation, and explain their limitations. For simplicity, we ignore the matter of zero knowledge (and salts) until we discuss it explicitly in Section 2.4.

(1) The basic variant. The *basic variant* of the Fiat–Shamir transformation, which we denote as $\mathsf{FS}[\mathsf{IP}]$, derives each IP verifier message by hashing the instance and all prior IP prover messages. Specifically, for every $i \in [\mathsf{k}]$, the i-th verifier message ρ_i is derived as

$$\rho_i := f_i\big(\mathbb{x}, \alpha_1, \ldots, \alpha_i\big)$$

where f_i is a random oracle dedicated to round i. This variant typically appears in theory research papers, and its security is discussed in detail in [CY24]. Briefly, the soundness (and knowledge soundness) of the resulting non-interactive argument is determined by the state-restoration soundness (and knowledge soundness) of IP.

(2) The hash-chain variant. The *hash-chain variant* of the Fiat–Shamir transformation, which we denote as $\mathsf{HCFS}[\mathsf{IP}]$, derives each IP verifier message by hashing the current IP prover message and the prior hash. Specifically, for every $i \in [\mathsf{k}]$, the i-th verifier message ρ_i is derived as

$$\rho_i := \begin{cases} f_i\big(\mathbb{x}, \alpha_1\big) & \text{if } i = 1 \\ f_i\big(\rho_{i-1}, \alpha_i\big) & \text{if } i > 1 \end{cases} .$$

This variant is more common in practice-oriented papers, and its security is also discussed in detail in [CY24]. Informally, this variant incurs an additive error of $O(\frac{t^2}{\min_{i\in[\mathsf{k}]}|\mathcal{M}_{\mathbf{V},i}|})$ against t-query adversaries, representing the (small) probability that a t-adversary can "break" the hash chain.[3]

Limitations in practice. Neither variant described above is used in practice due to the same limitation. Hash functions in practice have fixed input and output sizes; instead both variants rely on *oracles whose input and output domains are not fixed*. In $\mathsf{FS}[\mathsf{IP}]$, for each $i \in [\mathsf{k}]$, the oracle f_i receives inputs in $\mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i}$ (plus the instance $\mathbb{x}$) and produces outputs in $\mathcal{M}_{\mathbf{V},i}$. In $\mathsf{HCFS}[\mathsf{IP}]$, for each $i \in [\mathsf{k}]$, the oracle f_i receives inputs in $\mathcal{M}_{\mathbf{P},i}$ (plus the instance $\mathbb{x}$ or verifier message ρ_{i-1}) and produces outputs in $\mathcal{M}_{\mathbf{V},i}$. These message spaces depend on the protocol IP and instance $\mathbb{x}$, so the oracles' domains and ranges are not fixed.

A separate, and typically ignored, issue is message encoding/decoding: there is often a mismatch between the domains/ranges of oracles (e.g., binary strings) and message spaces (e.g., field elements or group elements).

2.2 DSFS: a Fiat–Shamir transformation based on the ideal duplex sponge

Practitioners consider variants of the Fiat–Shamir transformation that, intuitively, support sufficiently large inputs and outputs via multiple calls to a fixed-input-size hash function. Roughly, one can divide variants based on whether one assumes access to an ideal compression function or an ideal permutation function.

In this paper we consider the case of a permutation function, in line with the trends in symmetric cryptography that we discussed. Let cdc be a codec for IP that specifies encoding maps $(\varphi_i \colon \mathcal{M}_{\mathbf{P},i} \to \Sigma^{\ell_{\mathbf{P}}(i)})_{i\in[\mathsf{k}]}$ and decoding maps $(\psi_i \colon \Sigma^{\ell_{\mathbf{V}}(i)} \to \mathcal{M}_{\mathbf{V},i})_{i\in[\mathsf{k}]}$. We denote by $\mathsf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$ the transformation variant for the interactive proof IP and codec cdc that we outline below; see Section 4 for the formal description.

Duplex sponge. The *duplex sponge* construction [BDPVA12] is a well-established mode of operation for a given permutation $p \colon \Sigma^{r+c} \to \Sigma^{r+c}$, which allows absorbing and squeezing strings over Σ, with each call to p handling at most r elements of Σ at a time. This idea has been foundational to the design of hash functions like SHA-3 and primitives like extensible output functions [MF21].

In Section 3.3 we recall in detail the duplex sponge construction based on an ideal permutation $p \colon \Sigma^{r+c} \to \Sigma^{r+c}$ (as well as an additional oracle $h \colon \{0,1\}^{\leq n} \to \Sigma^c$ for offline preprocessing as motivated in Remark 2.1).[4] Here it suffices to summarize the main interfaces.

- $\mathsf{st}_0 := \mathsf{DS.Start}^p(x)$. Initialize the sponge state st_0 given a binary string x.
- $\mathsf{st}' := \mathsf{DS.Absorb}^p(\mathsf{st}, \boldsymbol{\chi})$. Absorb the input string $\boldsymbol{\chi} \in \Sigma^*$, changing the sponge state from st to st'.
- $(\boldsymbol{\rho}, \mathsf{st}') := \mathsf{DS.Squeeze}^p(\mathsf{st}, \ell)$. Squeeze an output $\boldsymbol{\rho} \in \Sigma^\ell$, changing the sponge state from st to st'.

Warmup: the sigma-protocol case. For simplicity first we describe the use of the duplex sponge in the special case of a sigma protocol, which is a particularly simple public-coin IP.

In a sigma protocol, the prover sends a first message $\alpha_1 \in \mathcal{M}_{\mathbf{P},1}$, the verifier sends a random message $\rho_1 \in \mathcal{M}_{\mathbf{V},1}$, and the prover sends a second message $\alpha_2 \in \mathcal{M}_{\mathbf{P},2}$. To bridge the gap between message spaces and the permutation's alphabet, the codec specifies an encoding function $\varphi_1 \colon \mathcal{M}_{\mathbf{P},1} \to \Sigma^{\ell_{\mathbf{P}}(1)}$ (that is injective) and a decoding function $\psi_1 \colon \Sigma^{\ell_{\mathbf{V}}(1)} \to \mathcal{M}_{\mathbf{V},1}$ (that has small bias).[5]

---

[3] In particular, one must ensure that each $|\mathcal{M}_{\mathbf{V},i}|$ is super-polynomial in the security parameter λ. If not the case, one can straightforwardly modify IP to have enough additional dummy randomness per round.

[4] Specifically, we consider the duplex sponge *in overwrite mode* rather than canonical duplex sponges "in XOR mode" as it is simpler and suffices for us.

[5] An example for Schnorr's protocol can be found in Appendix A.

The argument prover $\mathcal{P}^p(\mathbb{x}, \mathbb{w})$ initializes the sponge state $\mathsf{st}_0 := \mathsf{DS.Start}^p(\mathbb{x})$, uses $\mathbf{P}(\mathbb{x}, \mathbb{w})$ to compute the first IP prover message $\alpha_1 \in \mathcal{M}_{\mathbf{P},1}$, and then derives the single IP verifier message as follows:

1. $\widehat{\boldsymbol{\alpha}}_1 := \varphi_1(\alpha_1) \in \Sigma^{\ell_{\mathbf{P}}(1)}$ (encode the IP prover message);
2. $\mathsf{st}_1' := \mathsf{DS.Absorb}^p(\mathsf{st}_0, \widehat{\boldsymbol{\alpha}}_1)$ (absorb the encoded IP prover message);
3. $(\widehat{\boldsymbol{\rho}}_1, \mathsf{st}_1) := \mathsf{DS.Squeeze}^p(\mathsf{st}_1', \ell_{\mathbf{V}}(1))$ (squeeze the encoded IP verifier message);
4. $\rho_1 := \psi_1(\widehat{\boldsymbol{\rho}}_1) \in \mathcal{M}_{\mathbf{V},1}$ (decode to obtain the IP verifier message).

Finally, the argument prover gives ρ_1 to $\mathbf{P}(\mathbb{x}, \mathbb{w})$ in order to obtain the second IP prover message $\alpha_2 \in \mathcal{M}_{\mathbf{P},2}$, and outputs the argument string $\pi := (\alpha_1, \alpha_2)$.

The argument verifier $\mathcal{V}^p(\mathbb{x}, \pi)$ checks that $\mathbf{V}\big(\mathbb{x}, (\alpha_1, \alpha_2), \rho_1\big)$ accepts, where ρ_1 is derived from $\mathbb{x}$ and α_1 following the same procedure used by the argument prover.

The multi-round case. The above extends to the general case where $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ is a k-round public-coin IP: for each round $i \in [\mathsf{k}]$, the IP prover sends a message $\alpha_i \in \mathcal{M}_{\mathbf{P},i}$ and then the IP verifier sends a random message $\rho_i \in \mathcal{M}_{\mathbf{V},i}$;[6] after the interaction, the IP verifier's decision is computed as $\mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}\big)$. In this case, to bridge the gap between message spaces and the permutation's alphabet, the IP's codec specifies encoding functions $(\varphi_i \colon \mathcal{M}_{\mathbf{P},i} \to \Sigma^{\ell_{\mathbf{P}}(i)})_{i\in[\mathsf{k}]}$ and decoding functions $(\psi_i \colon \Sigma^{\ell_{\mathbf{V}}(i)} \to \mathcal{M}_{\mathbf{V},i})_{i\in[\mathsf{k}]}$.

The argument prover $\mathcal{P}^p(\mathbb{x}, \mathbb{w})$ and argument verifier $\mathcal{V}^p(\mathbb{x}, \pi)$ each initialize the sponge state $\mathsf{st}_0 := \mathsf{DS.Start}^p(\mathbb{x})$ and then rely on the codec and duplex sponge to derive the relevant IP verifier messages.[7] Specifically, for every $i \in [\mathsf{k}]$, the IP verifier message $\rho_i \in \mathcal{M}_{\mathbf{V},i}$ is derived according to these steps:

1. $\widehat{\boldsymbol{\alpha}}_i := \varphi_i(\alpha_i) \in \Sigma^{\ell_{\mathbf{P}}(i)}$ (encode the IP prover message);
2. $\mathsf{st}_i' := \mathsf{DS.Absorb}^p(\mathsf{st}_{i-1}, \widehat{\boldsymbol{\alpha}}_i)$ (absorb the encoded IP prover message);
3. $(\widehat{\boldsymbol{\rho}}_i, \mathsf{st}_i) := \mathsf{DS.Squeeze}^p(\mathsf{st}_i', \ell_{\mathbf{V}}(i))$ (squeeze the encoded IP verifier message);
4. $\rho_i := \psi_i(\widehat{\boldsymbol{\rho}}_i) \in \mathcal{M}_{\mathbf{V},i}$ (decode to obtain the IP verifier message).

Efficiency. **DSFS** is essentially optimal in terms of number of calls to the (fixed-size) permutation p. Specifically, putting aside the initialization based on the instance $\mathbb{x}$ (typically an offline computation), the argument prover makes $\sum_{i\in[\mathsf{k}-1]} \Big(\Big\lceil \frac{\ell_{\mathbf{P}}(i)}{r} \Big\rceil + \Big\lceil \frac{\ell_{\mathbf{V}}(i)}{r} \Big\rceil \Big)$ queries to p and the argument verifier makes $\sum_{i\in[\mathsf{k}]} \Big(\Big\lceil \frac{\ell_{\mathbf{P}}(i)}{r} \Big\rceil + \Big\lceil \frac{\ell_{\mathbf{V}}(i)}{r} \Big\rceil \Big)$ queries to p. This is essentially optimal for the information that must be absorbed/squeezed, given that p can absorb/squeeze at most r elements of Σ per query.

Unlike other Fiat–Shamir libraries used in the wild [Wri; Ark], **DSFS** does not hash lengths or labels of the prover and verifier messages. Moreover, differently from **HCFS**, **DSFS** uses only a single oracle and, after squeezing a verifier message, r symbols can be immediately absorbed without calling the permutation function. These efficiency details save precious (and unnecessary) queries to p.

Remark 2.1 (alternative instance hashing)**.** The argument prover $\mathcal{P}$ and argument verifier $\mathcal{V}$ initialize the sponge state as $\mathsf{st}_0 := \mathsf{DS.Start}^p(\mathbb{x})$, which is a computation that solely depends on the instance $\mathbb{x}$. The natural implementation of this step is $\mathsf{st}_0 := \mathsf{DS.Absorb}^p(\mathsf{st}_*, \varphi_0(\mathbb{x}))$ where st_* is a fixed sponge state and φ_0 is an encoding for binary strings. However, there are settings where initializing st_0 given $\mathbb{x}$ can be viewed as a distinct offline computation with possibly different priorities compared to the other duplex sponge computations (that depend on prover messages), which may favor alternative methods to hash $\mathbb{x}$, rather than

[6] We adopt the convention that each round consists of a prover message followed by a verifier message (in particular, the prover moves first); other conventions can be supported with straightforward changes.

[7] The argument prover derives $(\rho_i)_{i\in[\mathsf{k}-1]}$ in order to compute the IP prover messages $(\alpha_i)_{i\in[\mathsf{k}]}$, and the argument verifier derives $(\rho_i)_{i\in[\mathsf{k}]}$ in order to compute the decision bit.

$\mathsf{DS.Absorb}^p(\mathsf{st}_*, \varphi_0(\mathbb{x}))$.[8] Hence, in Section 3.3 (and subsequent technical sections) we consider a more flexible oracle setting: for instances $\mathbb{x}$ of size at most n, we consider an additional oracle $h\colon \{0,1\}^{\leq n} \to \Sigma^c$ to be used *only* for the sponge state initialization $\mathsf{st}_0 := \mathsf{DS.Start}^h(\mathbb{x})$ via a computation that stores $h(\mathbb{x})$ in st_0. Accordingly, our analysis separately keeps track of an adversary's queries to h and to p. The analysis then covers, as a special case, the particular choice where $\mathsf{DS.Start}^h(\mathbb{x}) = \mathsf{DS.Absorb}^p(\mathsf{st}_*, \varphi_0(\mathbb{x}))$ (with $h = p$).

2.3 Soundness and knowledge soundness

We outline the ideas behind Theorem 1, which establishes the soundness and knowledge soundness of $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$. The key technical lemma is a reduction that relates the security of $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$ to the security of $\mathbf{FS}[\mathsf{IP}]$ in a precise sense that we explain below. Intuitively this reduction suffices to prove Theorem 1 because the security of $\mathbf{FS}[\mathsf{IP}]$ is determined by the state-restoration security of IP (see [CY24]).

Below we let $(\mathcal{P}, \mathcal{V})$ be the argument prover and argument verifier of $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$, and $(\mathcal{P}_{\mathsf{std}}, \mathcal{V}_{\mathsf{std}})$ be the argument prover and argument verifier of $\mathbf{FS}[\mathsf{IP}]$. The oracle model for $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$ is a random permutation $p\colon \Sigma^{r+c} \to \Sigma^{r+c}$ and its inverse p^{-1} (we refer to this oracle distribution as $\mathcal{D}_{\mathfrak{S}}$), and the oracle model for $\mathbf{FS}[\mathsf{IP}]$ is random oracles $(f_i)_{i \in [\mathsf{k}]}$ where each f_i receives inputs in $\mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i}$ (plus the instance $\mathbb{x}$) and produces outputs in $\mathcal{M}_{\mathbf{V},i}$ (we refer to this oracle distribution as $\mathcal{D}_{\mathsf{IP}}$). Recall that, for security, we must consider adversaries that also have access to the inverse of p because, in practice, permutations are typically not designed to be hard to invert.

On the security reduction. We seek a procedure $\mathsf{D2SAlgo}$ that converts a query-efficient malicious argument prover $\tilde{\mathcal{P}}$ for $\mathcal{V}$ into a query-efficient malicious argument prover $\tilde{\mathcal{P}}_{\mathsf{std}}$ for $\mathcal{V}_{\mathsf{std}}$ that "behaves the same" as $\tilde{\mathcal{P}}$ up to a certain additive error $\eta_\star$ (which quantifies the security loss of $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$ over $\mathbf{FS}[\mathsf{IP}]$).

The precise meaning of "behaves the same" that suffices for our goals is not obvious. Below we motivate the specific form of our key technical lemma, which provides the desired precise meaning.

First we discuss the goal of reducing the soundness of $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$ to the soundness of $\mathbf{FS}[\mathsf{IP}]$. Informally, the (adaptive) soundness property upper bounds the probability, over the choice of oracles, that a query-efficient argument prover outputs a pair $(\mathbb{x}, \pi)$ such that $\mathbb{x}$ is not in the language and the argument verifier accepts $(\mathbb{x}, \pi)$. The key quantities in this experiment are the instance $\mathbb{x}$, argument string π, and decision bit b. Showing that the following two distributions are $\eta_\star$-close in statistical distance suffices:

$$\left\{ (\mathbb{x}, \pi, b) \;\middle|\; \begin{array}{c} (p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}} \\ (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}^{p, p^{-1}} \\ b \leftarrow \mathcal{V}^p(\mathbb{x}, \pi) \end{array} \right\} \quad \text{and} \quad \left\{ (\mathbb{x}, \pi, b) \;\middle|\; \begin{array}{r} (f_i)_{i \in [\mathsf{k}]} \leftarrow \mathcal{D}_{\mathsf{IP}} \\ (\mathbb{x}, \pi) \leftarrow \mathsf{D2SAlgo}^{(f_i)_{i \in [\mathsf{k}]}}(\tilde{\mathcal{P}}) \\ b \leftarrow \mathcal{V}_{\mathsf{std}}^{(f_i)_{i \in [\mathsf{k}]}}(\mathbb{x}, \pi) \end{array} \right\} .$$

However, showing that these two distributions are close does *not* suffice for knowledge soundness, as we now explain. Informally, the (adaptive) knowledge soundness property upper bounds the probability, over the choice of oracles, that a query-efficient argument prover outputs a pair $(\mathbb{x}, \pi)$ such that $(\mathbb{x}, \mathbb{w})$ is not in the relation and the argument verifier accepts $(\mathbb{x}, \pi)$, where $\mathbb{w}$ is the output of the knowledge extractor. The knowledge extractor receives the instance $\mathbb{x}$, argument string π, black-box access to the argument prover, and the query-answer trace of the argument prover. The query-answer trace is not part of the above statement, so it does not suffice to reduce the knowledge soundness of $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$ to the knowledge soundness of $\mathbf{FS}[\mathsf{IP}]$.

[8] For example, in recursive composition of zkSNARKs, the recursive circuit may directly receive as input st_0 (rather than having to compute st_0 from $\mathbb{x}$); this means that a different, possibly convenient, hash function can be used to hash $\mathbb{x}$ without affecting the recursive circuit's efficiency.

We fix this by following the pattern, developed in [CY24], of relying on an additional component: a procedure $\mathsf{D2STrace}$ that converts a query-answer trace for the permutation function p into a corresponding query-answer trace for the oracles $(f_i)_{i\in[\mathsf{k}]}$, in a way that is consistent with the prover conversion by $\mathsf{D2SAlgo}$.

This leads to our key technical lemma, which considers two distributions over quadruples $(\mathbb{x}, \pi, b, \mathsf{tr})$.

Lemma 1. *The following two distributions are $\eta_\star$-close in statistical distance:*

$$\left\{ (\mathbb{x}, \pi, b, \mathsf{tr}) \middle| \begin{array}{r} (p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}} \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{p}(\mathbb{x}, \pi) \\ \mathsf{tr} := \mathsf{D2STrace}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \end{array} \right\} \text{ and } \left\{ (\mathbb{x}, \pi, b, \mathsf{tr}) \middle| \begin{array}{r} (f_i)_{i\in[\mathsf{k}]} \leftarrow \mathcal{D}_{\mathsf{IP}} \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \mathsf{D2SAlgo}^{(f_i)_{i\in[\mathsf{k}]}}(\tilde{\mathcal{P}}) \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}_{\mathsf{std}}}} \mathcal{V}_{\mathsf{std}}^{(f_i)_{i\in[\mathsf{k}]}}(\mathbb{x}, \pi) \\ \mathsf{tr} := \mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}_{\mathsf{std}}} \end{array} \right\} . \tag{1}$$

The formal statement is Lemma 5.1, and in Section 6 we prove that the lemma enables us to straightforwardly establish soundness and knowledge soundness of $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$.

Establishing the key lemma. The technical core of the security reduction is establishing that the two distributions in Equation 1 are statistically close. Specifically, we prove that if $\tilde{\mathcal{P}}$ makes at most t queries then the statistical distance is at most

$$\eta_\star(t) := O\left(\frac{\left(t + \sum_{i\in[\mathsf{k}]} \left\lceil \frac{\ell_{\mathbf{V}}(i)}{r} \right\rceil \right)^2}{2\,|\Sigma^c|} \right) + \sum_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i} \,.$$

We outline the salient points of our proof, which works across several hybrids.

- First, we replace access to the random permutation p with access to the random oracles

  $$\boldsymbol{g} = \left(g_i \colon \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)} \to \Sigma^{\ell_{\mathbf{V}}(i)}\right)_{i\in[\mathsf{k}]} ,$$

  where each g_i takes as input encoded prover messages and outputs an encoded verifier message. We seek a reduction R that converts an argument prover $\tilde{\mathcal{P}}$ with oracle access to p into an "equivalent" argument prover $R(\tilde{\mathcal{P}})$ with oracle access to $\boldsymbol{g}$. Informally, R is tasked with using $\boldsymbol{g}$ to generate encoded verifier messages, rather than the duplex sponge based on p. This entails that R must do all the necessary book-keeping: it responds to queries in Σ^{r+c} with random answers in Σ^{r+c} and, whenever a sequence of queries to p can be recognized to be absorbing encoded prover messages in $\Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)}$, for some $i \in [\mathsf{k}]$, it queries g_i to obtain a corresponding output in $\Sigma^{\ell_{\mathbf{V}}(i)}$ to later use when another sequence of queries to p can be recognized as the corresponding squeezing operations. The challenge is that, in the world with p, encoded messages are absorbed/squeezed across multiple queries whereas, in the world with $\boldsymbol{g}$, a single query to the relevant oracle g_i with encoded prover messages yields the corresponding encoded verifier message.

  Intuitively, each query to p has input $(\boldsymbol{s}_{\mathrm{R,in}}, \boldsymbol{s}_{\mathrm{C,in}}) \in \Sigma^{r+c}$ and, by the duplex sponge construction, the capacity segment $\boldsymbol{s}_{\mathrm{C,in}}$ "points" to a previous output in the query-answer trace (it matches a previous output capacity segment). This allows us to recover a sequence of rate segments. We refer to this as *backtracking*. If backtracking yields a list of encoded prover messages $(\widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i)$, R can query g_i with it.

  However, a malicious argument prover might attempt at make backtracking ambiguous. For instance, if a collision in the capacity segment is found, backtracking might not find a single sequence of rate segments. We carefully prove that this event happens with probability at most $O(t^2 / |\Sigma^c|)$.

Even if backtracking is successful, there remains another obstacle: if $\ell_{\mathbf{V}}(i) > r$, then R cannot just return g_i's output $\widehat{\boldsymbol{\rho}}_i$ (else $\tilde{\mathcal{P}}$ would trivially distinguish). The reduction R splits $\widehat{\boldsymbol{\rho}}_i$ into segments of length r, and builds a sequence of queries and answers that would the duplex sponge to yield the same squeezed output. However, each of these $\lceil \ell_{\mathbf{V}}(i)/r \rceil$ queries may have been already queried in the past and, in the unfortunate case where this happens, R cannot respond with a different answer than before (else $\tilde{\mathcal{P}}$ would trivially distinguish). We carefully prove that this event happens with probability at most $t \cdot \left(\sum_{i \in [\mathsf{k}]} \frac{\lceil \ell_{\mathbf{V}}(i)/r \rceil}{|\Sigma^{r+c}|} \right)$.

- We replace access to the random oracles $\boldsymbol{g} = (g_i)_{i \in [\mathsf{k}]}$ with access to the random oracles

$$\boldsymbol{f} = \left(f_i \colon \mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i} \to \mathcal{M}_{\mathbf{V},i} \right)_{i \in [\mathsf{k}]}$$

  incurring a statistical distance that is at most $\sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}$ (the sum of the biases of $(\psi_i)_{i \in [\mathsf{k}]}$).

  This entails showing that, for every $i \in [\mathsf{k}]$, messages can be converted from $\Sigma^{\ell_{\mathbf{P}}(i)}$ to $\mathcal{M}_{\mathbf{P},i}$ and from $\Sigma^{\ell_{\mathbf{V}}(i)}$ to $\mathcal{M}_{\mathbf{V},i}$. Specifically, we prove that oracle access to $\boldsymbol{f}$ is $(\sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i})$-close to oracle access to

$$\boldsymbol{\psi} \circ \boldsymbol{g} \circ \boldsymbol{\varphi}^{-1} := \left(\psi_i \circ g_i \circ \varphi_i^{-1} \right)_{i \in [\mathsf{k}]} .$$

  Fixing $i \in [\mathsf{k}]$, we sketch how f_i is $\varepsilon_{\mathsf{cdc},i}$-close to $\psi_i \circ g_i \circ \varphi_i^{-1}$.

  The inverse encoding map φ_i^{-1} does not "disturb" the oracle: φ_i is injective, so two images under φ_i^{-1} are different if and only if their corresponding inputs are different. We deduce that $g_i \circ \varphi_i^{-1}$ produces random (and consistent) outputs over $\Sigma^{\ell_{\mathbf{V}}(i)}$. (Technical remark: the running time of $\boldsymbol{\varphi}^{-1} = (\varphi_i^{-1})_{i \in [\mathsf{k}]}$ ultimately affects the running time of the security reduction, due to the need to compute $\boldsymbol{\varphi}^{-1}$ for the above translation.)

  The decoding map ψ_i may bias the distribution but we are expecting to account for this: by definition, the statistical distance between $\psi_i(\mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)}))$ and $\mathcal{U}(\mathcal{M}_{\mathbf{V},i})$ is at most $\varepsilon_{\mathsf{cdc}_i}$.

Throughout the execution, the inverse permutation p^{-1} does not help the adversary and the claimed result follows from the difference lemma, adding the error terms together.

Remark 2.2 (on indifferentiability)**.** We do not know how to obtain Lemma 1 from the fact that the duplex sponge construction is indifferentiable from a random oracle (in particular, our proof of Lemma 1 is from scratch), and we suspect that this is not possible. We elaborate on this here.

Indifferentiability is a relaxation of indistinguishability that facilitates flexible security reductions between systems [MRH04]. In particular, the duplex sponge construction is indifferentiable from the regular sponge construction [BDPVA12, Lemma 3] and, in turn, the latter is indifferentiable from a random oracle [BDPV08].[9] Lemma 1 appears stronger than indifferentiability because it also concerns the query-answer trace of the adversary with the oracles, which has size t. Our result also tackles the problem of encoding prover messages, re-mapping verifier messages, and re-constructing the query-answers given to the random oracles $(f_i)_{i \in [\mathsf{k}]}$. Nevertheless, we believe that a stronger notion of indifferentiability (that remains weaker than indistinguishability) may suffice, and leave distilling such a notion to future work.

[9]These references focus on the sponge in XOR mode and mention that similar results hold for the sponge in overwrite mode. Conversely, our Lemma 1 can similarly be proved for the duplex sponge in XOR mode with appropriate changes in our proof.

2.4 Zero knowledge

The desired privacy behavior for a Fiat–Shamir transformation is the following: if the interactive proof IP is honest-verifier zero knowledge then the corresponding non-interactive argument NARG is zero knowledge (in the relevant oracle model). Theorem 2 says that $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$ has this desired behavior. Here we summarize the efficiency improvement achieved over prior variants, and how this affects the proof of zero knowledge.

Programming for zero knowledge. The zero-knowledge property for a non-interactive argument considers an adversary that chooses an instance-witness pair (in the relation) and then receives an argument string that is either the output of the argument prover or the output of the simulator. As in other settings the simulator must have an edge over the adversary; in this oracle setting, the simulator can *program* oracles (necessarily so [Pas03; Wee09]). Briefly, in addition to a simulated argument string, the simulator outputs a list of query-answer pairs for programming oracles, and the adversary subsequently receives query access to oracles modified this way. (See Definition 7.4 for the definition of zero-knowledge for a non-interactive argument.)

In particular, the adversary sees oracles before programming (when it chooses the instance-witness pair) and after programming (when it receives a simulated argument string). We deduce that any programmed location must be *unpredictable*, or else an adversary could trivially distinguish programmed oracles from non-programmed ones (and thereby distinguish the real-world and simulated-world distributions).

Programming for Fiat–Shamir. Informally, in a Fiat–Shamir transformation, the zero-knowledge simulator samples a simulated IP transcript (by using the honest-verifier zero-knowledge simulator of the IP) and then programs the oracles so that certain query answers match the verifier messages in the simulated IP transcript. Details of this programming differ across variants due to the differing use of oracles to derive verifier messages; regardless, as noted above, in all cases programmed points must be unpredictable.

In simple variants of the Fiat–Shamir transformation this unpredictability is "for free". For example, in many sigma protocols the first prover message α_1 is uniformly sampled from a cryptographically-large group and the (single) verifier message is derived as $\rho := f(\mathbb{x}, \alpha_1)$, and the simulator for the sigma protocol also outputs a random element α_1. Hence, the programmed location $(\mathbb{x}, \alpha_1)$ is unpredictable.

In general, however, we cannot rely on unpredictability coming from the simulated IP transcript. For example, consider an honest-verifier zero-knowledge IP where the verifier moves first (equivalently, the first round consists of a fixed prover message followed by a random verifier message). In this case there is no entropy coming from a prover message prior to the verifier message.

Salts, salts, salts. The common solution is to introduce salt strings into random oracle queries in order to make them unpredictable. For example, [CY24] analyses the variants $\mathbf{FS}[\mathsf{IP}]$ and $\mathbf{HCFS}[\mathsf{IP}]$ of the Fiat–Shamir transformation where a *salt string per IP round is used* (i.e., a total of k rounds).

The same approach of "one salt per oracle query" would work for $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$ but, unfortunately, would result in *too many salt strings*, specifically $\sum_{i \in [\mathsf{k}]} \left(\left\lceil \frac{\ell_{\mathbf{P}}(i)}{r-\delta} \right\rceil + \left\lceil \frac{\ell_{\mathbf{V}}(i)}{r-\delta} \right\rceil \right)$ salt strings of length δ. Indeed, we consider a setting where the oracle has fixed size, which means that the number of oracle queries is proportional to the communication complexity, not round complexity. Including all these salts in the final argument string π would be unacceptable due to the significant increase in argument size.

One salt suffices. We show that, in $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$, absorbing a single salt at the beginning suffices.

The (honest) argument prover $\mathcal{P}^p(\mathbb{x}, \mathbb{w})$ initializes the sponge state $\mathsf{st}'_0 := \mathsf{DS.Start}^p(\mathbb{x})$ (as before), samples a random salt $\boldsymbol{\tau} \in \Sigma^\delta$, and absorbs it $\mathsf{st}_0 := \mathsf{Absorb}^p(\mathsf{st}'_0, \boldsymbol{\tau}) \in \Sigma^{r+c} \times [0, r] \times [0, r]$; the rest of $\mathcal{P}$ remains unchanged, except the salt is included in the argument string as $\pi := (\boldsymbol{\tau}, (\alpha_i)_{i \in [\mathsf{k}]})$. The argument verifier $\mathcal{V}^p(\mathbb{x}, \pi)$ similarly initializes the sponge state and absorbs the salt before proceeding as before.

Establishing zero knowledge entails showing that every absorb and squeeze query after initialization is unpredictable (over a random $\boldsymbol{\tau} \in \Sigma^\delta$). We do so via a direct proof (without reducing zero-knowledge

of $\mathsf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$ to zero knowledge of $\mathsf{FS}[\mathsf{IP}]$). We carefully quantify the statistical distance between the real-world distribution and simulated-world distribution, by taking into account not only the effect of the (single) salt, but also the biases incurred by the decoding maps.

The analysis is split in intermediate hybrids, isolating different distinguishing advantages: (i) arising from queries that absorb the salt $\boldsymbol{\tau}$, of which there are $\lceil \delta/r \rceil$; (ii) arising from queries that squeeze verifier messages, of which there are $\sum_{i \in [\mathsf{k}]} \lceil \ell_{\mathbf{V}}(i)/r \rceil$; (iii) arising from decoding biases. The latter involves measuring the statistical distance between

$$\mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)}) \quad \text{and} \quad (\psi_i^{-1} \circ \psi_i \circ \mathcal{U})(\Sigma^{\ell_{\mathbf{V}}(i)}) \, .$$

This can be done in two steps: first showing that $\psi_i(\mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)}))$ is $\varepsilon_{\mathsf{cdc},i}$-close to $\mathcal{U}(\mathcal{M}_{\mathbf{V},i})$, and then showing that $\psi_i^{-1}(\mathcal{U}(\mathcal{M}_{\mathbf{P},i}))$ is identically distributed to $\mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)})$.

Overall, our proof of Theorem 2 is significantly more delicate than corresponding ones in [CY24]. See Section 7 for the formal statement of Theorem 2 and its proof.

Remark 2.3 (one salt in prior variants)**.** Our analysis leads to an efficiency improvement in $\mathsf{FS}[\mathsf{IP}]$ and $\mathsf{HCFS}[\mathsf{IP}]$. Specifically, our analysis can be adapted to show that one salt would have sufficed for zero knowledge in both of those constructions, rather than k salts as done in [CY24].

2.5 Application: BCS transformation from a duplex sponge

The BCS transformation can be viewed as the composition of two separate transformations: (i) the iBCS (interactive BCS) transformation, which maps a public-coin IOP into a corresponding public-coin (succinct) interactive argument, by relying on a Merkle commitment based on a random oracle; (ii) the Fiat–Shamir transformation, which maps the public-coin interactive argument into a corresponding non-interactive argument, by relying on other random oracles. This can be informally summarized via the following "transformation equation" (and is formally discussed in [CY24]):

$$\mathsf{BCS}[\mathsf{IOP}] = \mathsf{HCFS}[\mathsf{iBCS}[\mathsf{IOP}]] \, . \tag{2}$$

We can replace the inner transformation with our variant to obtain a variant of the BCS transformation that internally relies on a (fixed-size) ideal permutation for the purpose of a Fiat–Shamir transformation. (The compression function for the Merkle commitment remains.) This can be informally summarized as follows:

$$\mathsf{BCS}[\mathsf{IOP}] = \mathsf{DSFS}[\mathsf{iBCS}[\mathsf{IOP}], \mathsf{cdc}] \, . \tag{3}$$

where cdc is an appropriate codec for encoding Merkle commitments (rather than IOP prover messages) and decoding IOP verifier messages for IOP.

Upper bounds on the soundness and knowledge soundness follow as a corollary of Theorem 1, replacing the error $\frac{t^2}{2^\lambda}$ of HCFS with $\frac{2t^2}{|\Sigma|^c}$ of DSFS. The multi-extraction error κ_{MT} that arises in iBCS remains.

Corollary 1 (informal)**.** *If* IOP *has state-restoration soundness error* $\varepsilon_{\mathrm{IOP}}^{\mathrm{sr}}$ *then* $\mathsf{NARG} := \mathsf{DSFS}[\mathsf{iBCS}[\mathsf{IOP}], \mathsf{cdc}]$ *has soundness error* $\varepsilon_{\mathsf{NARG}}$ *such that*

$$\varepsilon_{\mathsf{NARG}}(t) \le \varepsilon_{\mathrm{IOP}}^{\mathrm{sr}}(t) + \kappa_{\mathrm{MT}}(t, \mathsf{l}, (t+1) \cdot \mathsf{k}, \mathsf{k}) + \frac{2t^2}{|\Sigma|^c} \, .$$

Moreover, a similar statement holds for knowledge soundness: if IOP *has state-restoration knowledge soundness error* $\kappa_{\mathrm{IOP}}^{\mathrm{sr}}$ *then* NARG *has knowledge soundness error* κ_{NARG} *such that*

$$\kappa_{\mathsf{NARG}}(t) \le \kappa_{\mathrm{IOP}}^{\mathrm{sr}}(t) + \kappa_{\mathrm{MT}}(t, \mathsf{l}, (t+1) \cdot \mathsf{k}, \mathsf{k}) + \frac{2t^2}{|\Sigma|^c} \, .$$

3 Preliminaries

Several definitions in this section (most notably related to interactive proofs, non-interactive arguments, and the Fiat–Shamir transformation) are direct adaptations of definitions in [CY24].

3.1 Notation

With $[n]$ we denote the range $\{1, \ldots, n\}$. An alphabet Σ is a non-empty finite set[10], containing an element $0 \in \Sigma$ denoted *default value*. The Kleene closure Σ^* denotes all finite-length strings over Σ. Let A and B be algorithms. We say that A *uses* B *as a black-box*, denoted $A(B)$, to emphasize that A relies only on the input-output functionality of B, rather than also using its description. If A invokes B some number of times (on inputs of its choice) and otherwise does not "look" at B's description. More precisely, for every two algorithms B_1 and B_2 that represent the same function, $A(B_1) = A(B_2)$.

3.2 Basics

The preimage of a function $f\colon X \to Y$ is a map $f^{-1}\colon 2^Y \to 2^X : C \mapsto \{x \in X : f(x) \in C\}$. With a slight abuse of notation, for $x \in X$ we denote by $f^{-1}(x)$ the preimage of the singleton $\{x\}$ under f.

A ***relation*** $\mathcal{R}$ is a set of instance-witness pairs $(\mathbb{x}, \mathbb{w})$. The ***language*** associated to a relation $\mathcal{R}$ is the set $\mathcal{L}(\mathcal{R})$ of all instances $\mathbb{x}$ for which there exists a witness $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$. We denote by $|\mathbb{x}|$ the length of the instance $\mathbb{x}$ (according to some length measure, e.g., the number of bits used to describe $\mathbb{x}$).

We write $a \leftarrow \mathcal{D}$ to denote that the element a is sampled according to the distribution $\mathcal{D}$. We denote by $\mathcal{U}(S)$ the uniform distribution over a given non-empty finite set S. We denote by $X \to Y$ the set of functions from X to Y, so $\mathcal{U}(X \to Y)$ refers to the uniform distribution over all such functions. To ease notation, the act of sampling multiple functions $f_i\colon X_i \to Y_i$ is denoted as

$$\boldsymbol{f} = (f_i)_{i \in [\mathsf{k}]} \leftarrow \mathcal{U}((X_i \to Y_i)_{i \in [\mathsf{k}]}) ,$$

indicating that, for each $i \in [\mathsf{k}]$, $f_i \leftarrow \mathcal{U}(X_i \to Y_i)$ is sampled uniformly and independently at random.

We denote $\mathcal{A}^f$ the fact that algorithm $\mathcal{A}$ has query access to the function f, and $\mathcal{A}^{\boldsymbol{f}}$ the fact that $\mathcal{A}$ has query access to the functions $\boldsymbol{f} = (f_i)_{i \in [\mathsf{k}]}$. We say that $\mathcal{A}$ is t-query if it makes at most t queries (to any oracle), and is $(t_i)_{i \in [\mathsf{k}]}$-query if, for every $i \in [\mathsf{k}]$, it makes at most t_i queries to the i-th oracle. We write $a \overset{\mathsf{tr}}{\leftarrow} \mathcal{A}^f$ to denote the fact that the query-answer trace of $\mathcal{A}$ with respect to the oracle f is tr; we also write $a \overset{\mathsf{tr}}{\leftarrow} \mathcal{A}^{\boldsymbol{f}}$ for the case of multiple oracles, in which case tr additionally includes for each query-answer pair the information about which oracle it is for. When sampling from multiple oracles f_i for $i \in [\mathsf{k}]$, we denote query-answer trace as tr.

Definition 3.1. *We assume the random oracle is implemented via* **lazy sampling***, that is, via the following stateful algorithm:*

- *Set the internal state* tr *to be the empty mapping.*
- *Upon receiving a query* x*, if* $x \in \mathsf{tr}$*, return* $\mathsf{tr}[x]$*; otherwise, sample* $y \leftarrow \mathcal{U}(Y)$ *uniformly at random, set* $\mathsf{tr}[x] \coloneqq y$ *and return* y*.*

The list of key-value pairs of tr *is called the* **trace** *of the random oracle.*

[10] We will generally assume that Σ has *partial order*. This is only needed in order to perform dichotomic search in logarithmic time over a list of Σ-tuples (cf. Sections 5.1 and 5.2).

3.3 Duplex sponge in overwrite mode

We describe a duplex sponge (in overwrite mode) that is directly inspired from [BDPVA12, Sec. 6.2]. The setting is one where there are two random oracles:

- a random function $h\colon \{0,1\}^{\le n} \to \Sigma^c$, and
- a random permutation $p\colon \Sigma^{r+c} \to \Sigma^{r+c}$ (that we sometimes view as a function $p\colon \Sigma^r \times \Sigma^c \to \Sigma^r \times \Sigma^c$).

Here Σ is a finite (non-empty) alphabet, c is the ***capacity***, r is the ***rate***, and $r + c$ is the ***permutation length***.

Informally, the duplex sponge consists of three procedures that initialize and evolve a ***sponge state*** st, which is a tuple

$$\mathsf{st} = (\boldsymbol{s}, i_{\mathrm{A}}, i_{\mathrm{S}}) \in \Sigma^{r+c} \times [0, r] \times [0, r]$$

consisting of a permutation state $\boldsymbol{s} = (\boldsymbol{s}_{\mathrm{R}}, \boldsymbol{s}_{\mathrm{C}}) \in \Sigma^{r+c}$ (with a rate segment $\boldsymbol{s}_{\mathrm{R}}$ and a capacity segment $\boldsymbol{s}_{\mathrm{C}}$), an absorbing index $i_{\mathrm{A}} \in [0, r]$ for (over)writing parts of the permutation state, and a squeezing index $i_{\mathrm{S}} \in [0, r]$ for reading from parts of the permutation state; reading and writing is always to the rate segment.

- An *initialization phase*, given an input x, produces the initial sponge state $\mathsf{st}_0 = (\boldsymbol{s}, i_{\mathrm{A}}, i_{\mathrm{S}}) = \big((0^r, \boldsymbol{s}_{\mathrm{C}}), 0, r\big)$ where the rate segment is initialized to 0^r and the capacity segment is initialized to $\boldsymbol{s}_{\mathrm{C}} := h(x)$.
- An *absorbing phase*, where an input block $\boldsymbol{\chi} \in \Sigma^*$ is written into rate segment of the sponge state, interleaved with applications of the permutation p. (The overwrite mode refers to the fact that we overwrite the rate segment with the input block, rather than "xoring" the rate segment with the input block.)
- A *squeezing phase* where, for a desired output length $\ell \in \mathbb{N}$, yields an output $\boldsymbol{\rho} \in \Sigma^\ell$ that is obtained from the rate segment of the permutation state, in blocks of r elements interleaved with calls to the permutation p.

In more detail, the duplex sponge is constructed as follows.

Construction 3.2. *For $r, c, n \in \mathbb{N}$, let $h\colon \{0,1\}^{\le n} \to \Sigma^c$ and $p\colon \Sigma^{r+c} \to \Sigma^{r+c}$ be oracles. The* **duplex sponge (in overwrite mode)** *is a list of procedures* $\mathsf{DS} := (\mathsf{Start}, \mathsf{Absorb}, \mathsf{Squeeze})$ *that have query access to the oracles (h, p, p^{-1}) and work as follows.*

- $\mathsf{st}_0 := \mathsf{DS.Start}^h(x)$.
  Given as input $x \in \{0,1\}^{\le n}$, compute $\boldsymbol{s}_{\mathrm{C}} := h(x) \in \Sigma^c$, set the initial permutation state $\boldsymbol{s} := (0^r, \boldsymbol{s}_{\mathrm{C}}) \in \Sigma^{r+c}$, set the initial absorbing index $i_{A} := 0$, set the initial squeezing index $i_{S} := r$, and output the initial sponge state:

  $$\mathsf{st}_0 := (\boldsymbol{s}, i_{A}, i_{S}) = \big((0^r, \boldsymbol{s}_{\mathrm{C}}), 0, r\big) \in \Sigma^{r+c} \times [0, r] \times [0, r]\,.$$

- $\mathsf{st}' := \mathsf{DS.Absorb}^p(\mathsf{st}, \boldsymbol{\chi})$.
  Given as input $\mathsf{st} = (\boldsymbol{s}, i_{A}, i_{S})$ and $\boldsymbol{\chi} \in \Sigma^*$, output st' computed as follows.

  1. *Set $i'_{S} := r$.*
  2. *If $\boldsymbol{\chi}$ is the empty string, then return $\mathsf{st}' := (\boldsymbol{s}, i_{A}, i'_{S})$.*
  3. *Otherwise:*
     (a) *If $0 \le i_{A} < r$ then:*
       - *Let χ be the first element of $\boldsymbol{\chi}$, and let $\boldsymbol{\chi}'$ be the remaining elements of $\boldsymbol{\chi}$.*
       - *Let $\boldsymbol{s}'$ be the permutation state obtained by overwriting the i_{A}-th element of $\boldsymbol{s}$ with χ.*
       - *Set $i'_{A} := i_{A} + 1$ (increment the absorbing index).*
       - *Set $\mathsf{st}' := (\boldsymbol{s}', i'_{A}, i'_{S})$.*
       - *Output* $\mathsf{DS.Absorb}(\mathsf{st}', \boldsymbol{\chi}')$.

(b) *If* $i_{\mathsf{A}} = r$ *then:*
   - *Set* $\boldsymbol{s}' := p(\boldsymbol{s})$.
   - *Set* $i'_{\mathsf{A}} := 0$ *(reset the absorbing index).*
   - *Set* $\mathsf{st}' := (\boldsymbol{s}', i'_{\mathsf{A}}, i'_{\mathsf{S}})$.
   - *Output* $\mathsf{DS.Absorb}(\mathsf{st}', \boldsymbol{\chi})$.

- $(\boldsymbol{\rho}, \mathsf{st}') := \mathsf{DS.Squeeze}^{p}(\mathsf{st}, \ell)$.
  Given as input $\mathsf{st} = (\boldsymbol{s}, i_{\mathsf{A}}, i_{\mathsf{S}})$ *and* $\ell \in \mathbb{N}$, *output* $\boldsymbol{\rho} \in \Sigma^{\ell}$ *and* st' *computed as follows.*

1. *Set* $i'_{\mathsf{A}} := r$ *(the absorbing index is not used while squeezing).*
2. *If* $\ell = 0$ *set* $\boldsymbol{\rho}$ *to the empty string, set* $\mathsf{st}' := (\boldsymbol{s}, i'_{\mathsf{A}}, i_{\mathsf{S}})$, *and output* $(\boldsymbol{\rho}, \mathsf{st}')$.
3. *Otherwise:*
   (a) *If* $0 \le i_{\mathsf{S}} < r$ *then:*
   - *Set* $i'_{\mathsf{S}} := i_{\mathsf{S}} + 1$ *(increment the squeezing index).*
   - *Set* $\mathsf{st}' := (\boldsymbol{s}, i'_{\mathsf{A}}, i'_{\mathsf{S}})$.
   - *Compute* $(\boldsymbol{\rho}', \mathsf{st}') := \mathsf{DS.Squeeze}(\mathsf{st}', \ell - 1)$.
   - *Output* $(\boldsymbol{s}_{i_{\mathsf{S}}} \| \boldsymbol{\rho}', \mathsf{st}')$.

   (b) *If* $i_{\mathsf{S}} = r$ *then:*
   - *Set* $\boldsymbol{s}' := p(\boldsymbol{s})$.
   - *Set* $i'_{\mathsf{S}} := 0$ *(reset the squeezing index).*
   - *Set* $\mathsf{st}' := (\boldsymbol{s}', i'_{\mathsf{A}}, i'_{\mathsf{S}})$.
   - *Output* $\mathsf{DS.Squeeze}(\mathsf{st}', \ell)$.

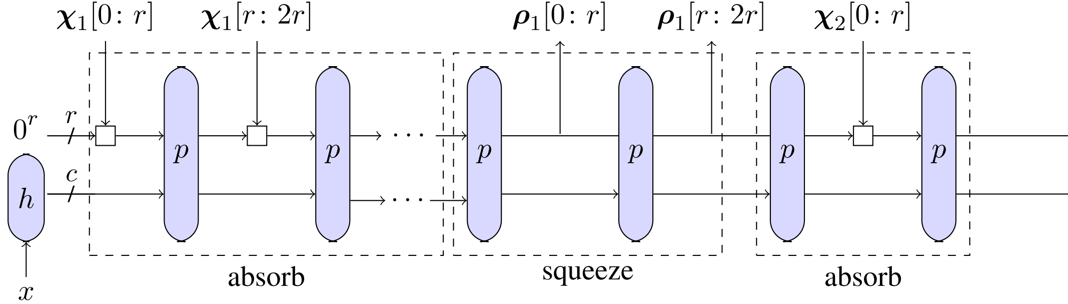

Figure 1: Diagram of the duplex sponge in Construction 3.2.

3.4 Non-interactive arguments in oracle models

An *oracle distribution* $\mathcal{D}$ receives as input a security parameter $\lambda \in \mathbb{N}$ and an instance size bound $n \in \mathbb{N}$, and samples a list $\boldsymbol{f}$ of functions. A *non-interactive argument* (NARG) in the $\mathcal{D}$-oracle model is a tuple $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$, where $\mathcal{P}$ is an oracle algorithm known as the *argument prover* and $\mathcal{V}$ is an oracle algorithm known as the *argument verifier*, that works as follows. For a given security parameter $\lambda \in \mathbb{N}$ and instance size bound $n \in \mathbb{N}$, a list of functions $\boldsymbol{f}$ is sampled according to $\mathcal{D}(\lambda, n)$ and is made public; anyone, including the argument prover $\mathcal{P}$ and the argument verifier $\mathcal{V}$, can query any function in $\boldsymbol{f}$. The argument prover $\mathcal{P}$ receives as input an instance $\mathbb{x}$ and witness $\mathbb{w}$, and outputs an argument string π. The argument verifier $\mathcal{V}$ receives as input the instance $\mathbb{x}$ and argument string π, and outputs a bit denoting whether to accept (the bit is 1) or reject

(the bit is 0). Both $\mathcal{P}$ and $\mathcal{V}$ additionally also receive as input λ and n, but we omit them for ease of notation. We consider several properties for a non-interactive argument, stated below. The definitions are from [CY24] (and straightforwardly adapted to any oracle model); we refer the reader to the relevant discussions there.

Definition 3.3. *A non-interactive argument* $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ *for relation* $\mathcal{R}$ *in the* $\mathcal{D}$*-oracle model has* **(perfect) completeness** *if for every security parameter* $\lambda \in \mathbb{N}$*, instance size bound* $n \in \mathbb{N}$*, and instance-witness pair* $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ *such that* $|\mathbb{x}| \leq n$*,*

$$\Pr\left[\mathcal{V}^{\boldsymbol{f}}(\mathbb{x}, \pi) = 1 \,\middle|\, \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n) \\ \pi \leftarrow \mathcal{P}^{\boldsymbol{f}}(\mathbb{x}, \mathbb{w}) \end{array}\right] = 1\,.$$

The probability is taken over $\boldsymbol{f}$ *and any randomness of the argument prover* $\mathcal{P}$ *and verifier* $\mathcal{V}$*.*

Definition 3.4. *A non-interactive argument* $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ *for relation* $\mathcal{R}$ *in the* $\mathcal{D}$*-oracle model has* **soundness error** $\varepsilon_{\mathsf{NARG}}$ *if for every security parameter* $\lambda \in \mathbb{N}$*, query bound* $t \in \mathbb{N}$*,* t*-query malicious argument prover* $\tilde{\mathcal{P}}$*, and instance size bound* $n \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} |\mathbb{x}| \leq n \\ \wedge\ \mathbb{x} \notin \mathcal{L}(\mathcal{R}) \\ \wedge\ \mathcal{V}^{\boldsymbol{f}}(\mathbb{x}, \pi) = 1 \end{array} \,\middle|\, \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n) \\ (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}^{\boldsymbol{f}} \end{array}\right] \leq \varepsilon_{\mathsf{NARG}}(\lambda, t, n)\,.$$

The probability is taken over $\boldsymbol{f}$ *and any randomness of the argument verifier* $\mathcal{V}$*.*

Definition 3.5. *A non-interactive argument* $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ *for a relation* $\mathcal{R}$ *in the* $\mathcal{D}$*-oracle model has* **straightline knowledge soundness error** $\kappa_{\mathsf{NARG}}\big(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n)\big)$ *if there exists a polynomial-time deterministic algorithm* $\mathcal{E}$ *(the* extractor*) such that for every security parameter* $\lambda \in \mathbb{N}$*, query bound* $t \in \mathbb{N}$*,* t*-query deterministic argument prover* $\tilde{\mathcal{P}}$*, and instance size bound* $n \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} |\mathbb{x}| \leq n \\ \wedge\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \\ \wedge\ \mathcal{V}^{\boldsymbol{f}}(\mathbb{x}, \pi) = 1 \end{array} \,\middle|\, \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}} \tilde{\mathcal{P}}^{\boldsymbol{f}} \\ \mathbb{w} \leftarrow \mathcal{E}(\mathbb{x}, \pi, \mathsf{tr}) \end{array}\right] \leq \kappa_{\mathsf{NARG}}(\lambda, t, n)\,.$$

Definition 3.6. *Let* $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ *be a non-interactive argument in the* $\mathcal{D}$*-oracle model. A deterministic argument prover* $\tilde{\mathcal{P}}$ *has* **failure probability** $\delta_{\tilde{\mathcal{P}}}$ *if for every security parameter* $\lambda \in \mathbb{N}$ *and instance size bound* $n \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} |\mathbb{x}| > n \\ \vee\ \mathcal{V}^{\boldsymbol{f}}(\mathbb{x}, \pi) = 0 \end{array} \,\middle|\, \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n) \\ (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}^{\boldsymbol{f}} \end{array}\right] \leq \delta_{\tilde{\mathcal{P}}}(\lambda, n)\,.$$

Definition 3.7. *A non-interactive argument* $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ *for a relation* $\mathcal{R}$ *in the* $\mathcal{D}$*-oracle model has* **rewinding knowledge soundness error** $\kappa_{\mathsf{NARG}}\big(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n)\big)$ **with extraction time** $\mathbf{et}_{\mathsf{NARG}}$ *if there exists a probabilistic algorithm* $\mathcal{E}$ *(the* extractor*) such that for every security parameter* $\lambda \in \mathbb{N}$*, query bound* $t \in \mathbb{N}$*,* t*-query deterministic argument prover* $\tilde{\mathcal{P}}$ *with failure probability* $\delta_{\tilde{\mathcal{P}}}$ *and running time* $\tau_{\tilde{\mathcal{P}}}$*, and instance size bound* $n \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} |\mathbb{x}| \leq n \\ \wedge\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \\ \wedge\ b = 1 \end{array} \,\middle|\, \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}} \tilde{\mathcal{P}}^{\boldsymbol{f}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{\boldsymbol{f}}(\mathbb{x}, \pi) \\ \mathbb{w} \leftarrow \mathcal{E}(\mathbb{x}, \pi, \mathsf{tr}, \mathsf{tr}_{\mathcal{V}}, \tilde{\mathcal{P}}) \end{array}\right] \leq \kappa_{\mathsf{NARG}}\big(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n)\big)\,.$$

Moreover, $\mathcal{E}$ *runs in expected time* $\mathbf{et}_{\mathsf{NARG}}\,\big(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n), \tau_{\tilde{\mathcal{P}}}(\lambda, n)\big)$ *(over the given inputs and internal randomness).*

3.5 Interactive proofs

An *interactive proof* (IP) for relation $\mathcal{R}$ is a tuple of interactive algorithms $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ that works as follows. The IP prover $\mathbf{P}$ receives as input an instance-witness pair $(\mathbb{x}, \mathbb{w})$ and the IP verifier $\mathbf{V}$ receives as input an instance $\mathbb{x}$. They interact across k rounds and, in each round $i \in [\mathsf{k}]$, the IP prover sends a message α_i and then the IP verifier sends a message ρ_i. After the interaction, the IP verifier outputs a decision bit $b \in \{0,1\}$. We denote by $\langle \mathbf{P}, \mathbf{V} \rangle_{\mathsf{IP}}$ the random variable that equals the output of $\mathbf{V}$ after interacting with $\mathbf{P}$ (the probability is taken over the randomness of $\mathbf{P}$ and of $\mathbf{V}$).

Definition 3.8. $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *for relation* $\mathcal{R}$ *is (perfectly)* **complete** *if for every instance-witness pair* $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$

$$\Pr[\langle \mathbf{P}(\mathbb{x}, \mathbb{w}), \mathbf{V}(\mathbb{x}) \rangle_{\mathsf{IP}} = 1] = 1 \, .$$

Definition 3.9. $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *for relation* $\mathcal{R}$ *has* **soundness error** $\varepsilon_{\mathsf{IP}}^{\mathrm{snd}}$ *if for every* $\mathbb{x} \notin \mathcal{L}(\mathcal{R})$ *and malicious IP prover* $\tilde{\mathbf{P}}$

$$\Pr\left[\langle \tilde{\mathbf{P}}, \mathbf{V}(\mathbb{x}) \rangle_{\mathsf{IP}} = 1\right] \leq \varepsilon_{\mathsf{IP}}^{\mathrm{snd}}(\mathbb{x}) \, .$$

We additionally define $\varepsilon_{\mathsf{IP}}^{\mathrm{snd}}(n) \coloneqq \max_{\substack{\mathbb{x} \notin \mathcal{L}(\mathcal{R}) \\ |\mathbb{x}| \leq n}} \varepsilon_{\mathsf{IP}}^{\mathrm{snd}}(\mathbb{x})$.

This work focuses on public-coin interactive proofs. An IP is ***public coin*** if every message sent by $\mathbf{V}$ is a random string in Σ^*, each sampled independently at random, and $\mathbf{V}$ has no other randomness. In this case, we can view the IP verifier as a deterministic algorithm $\mathbf{V}(\mathbb{x}, \boldsymbol{\alpha}, \boldsymbol{\rho})$ that outputs a decision bit (where $\boldsymbol{\alpha}$ is the sequence of messages sent by the prover and $\boldsymbol{\rho}$ is the sequence of random messages sent by the verifier).

Message spaces. For every $i \in [\mathsf{k}]$, we denote by $\mathcal{M}_{\mathbf{P},i}$ and $\mathcal{M}_{\mathbf{V},i}$ the prover message space and verifier message space for the i-th round. (These spaces may be functions of, e.g., the instance size.) The message spaces induce corresponding function spaces that we use later on.

Definition 3.10. *For every instance size bound* $n \in \mathbb{N}$ *and salt size* $\delta \in \mathbb{N}$*, we define the following function spaces: for every* $i \in [\mathsf{k}]$*,* $\mathcal{Z}_i(\delta, n) \coloneqq \{0,1\}^{\leq n} \times \{0,1\}^{\leq \delta} \times \mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i} \to \mathcal{M}_{\mathbf{V},i}$.

State restoration. We describe a state-restoration soundness and knowledge soundness, security notions that (informally) allow a malicious prover to obtain (up to a query bound t) verifier messages for the same prover input. Our formulation is more permissive than [CY24, Def. 12.1], as it allows the prover not to pick a salt for each round. This change does not affect the soundness results given in the book.

Definition 3.11. *The* **IP state-restoration game** *for* $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *with salt size* $\delta \in \mathbb{N}$*, random oracles* $\boldsymbol{f} = (f_i)_{i \in [\mathsf{k}]} \in \mathcal{U}((\mathcal{Z}_i(\delta, n))_{i \in [\mathsf{k}]})$*, and IP state-restoration prover* $\tilde{\mathbf{P}}^{\mathrm{sr}}$ *is defined below.*

$\mathrm{SR}_{\mathsf{IP}, \boldsymbol{f}, \tilde{\mathbf{P}}^{\mathrm{sr}}}(\delta)$*:*

1. *Repeat the following until* $\tilde{\mathbf{P}}^{\mathrm{sr}}$ *decides to exit the loop.*
   (a) $\tilde{\mathbf{P}}^{\mathrm{sr}}$ *outputs* $(\mathbb{x}, \boldsymbol{\tau}, (\alpha_1, \ldots, \alpha_i))$*, where* $\mathbb{x}$ *is an instance,* $(\alpha_1, \ldots, \alpha_i)$ *are IP prover messages, and* $\boldsymbol{\tau}$ *are salt strings in* $\{0,1\}^{\leq i \cdot \delta}$.
   (b) *Set* $\rho_i \coloneqq f_i(\mathbb{x}, \boldsymbol{\tau}, (\alpha_1, \ldots, \alpha_i))$.
   (c) *Send* ρ_i *to* $\tilde{\mathbf{P}}^{\mathrm{sr}}$.
2. $\tilde{\mathbf{P}}^{\mathrm{sr}}$ *outputs* $(\mathbb{x}, \boldsymbol{\tau}, (\alpha_1, \ldots, \alpha_{\mathsf{k}}))$*, where* $\mathbb{x}$ *is an instance,* $(\alpha_1, \ldots, \alpha_{\mathsf{k}})$ *are IP prover messages, and* $\boldsymbol{\tau} \in \{0,1\}^{\leq (\mathsf{k}-1)\delta}$ *are salt strings.*
3. *For every* $i \in [\mathsf{k}]$*, set* $\rho_i \coloneqq f_i(\mathbb{x}, \boldsymbol{\tau}, (\alpha_1, \ldots, \alpha_i))$.

4. *Output* $(\mathbb{x}, \boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]})$.

We denote by $\mathsf{tr}^{\mathrm{sr}}$ *the list of move-response pairs of the form* $((\mathbb{x}, \boldsymbol{\tau}, (\alpha_1, \ldots, \alpha_i)), \rho_i)$ *performed in the loop. We show* $\mathsf{tr}^{\mathrm{sr}}$ *in an execution of the IP state-restoration game using the following notation:*

$$(\mathbb{x}, \boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}) \xleftarrow{\mathsf{tr}^{\mathrm{sr}}} \mathrm{SR}_{\mathsf{IP},f,\tilde{\mathbf{P}}^{\mathrm{sr}}}(\delta)\,.$$

$\tilde{\mathbf{P}}^{\mathrm{sr}}$ *is* t**-move** *if* $\tilde{\mathbf{P}}^{\mathrm{sr}}$ *exits the loop after at most* t *iterations.*

The above game directly leads to the notion of state-restoration soundness error: it is an upper bound on the probability that any IP prover in the IP state-restoration game can find an instance not in the language and an accepting transcript for it.

Definition 3.12. $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *has* **state-restoration soundness error** $\varepsilon^{\mathrm{sr}}_{\mathsf{IP}}$ *if for every salt size* $\delta \in \mathbb{N}$*, move budget* $t \in \mathbb{N}$*,* t*-move malicious IP state-restoration prover* $\tilde{\mathbf{P}}^{\mathrm{sr}}$*, and instance size bound* $n \in \mathbb{N}$*:*

$$\Pr\left[\begin{array}{l} |\mathbb{x}| \le n \\ \wedge\ \mathbb{x} \notin \mathcal{L}(\mathcal{R}) \\ \wedge\ \mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}\big) = 1 \end{array} \,\middle|\, \begin{array}{l} \boldsymbol{f} = (f_i)_{i\in[\mathsf{k}]} \leftarrow \mathcal{U}((\mathcal{Z}_i(\delta,n))_{i\in[\mathsf{k}]}) \\ (\mathbb{x}, \boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}) \leftarrow \mathrm{SR}_{\mathsf{IP},\boldsymbol{f},\tilde{\mathbf{P}}^{\mathrm{sr}}}(\delta) \end{array}\right] \le \varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(\delta, t, n)\,.$$

The straightline variant of state-restoration knowledge soundness considers a (deterministic) knowledge extractor $\mathbf{E}^{\mathrm{sr}}$ that is tasked with finding a witness $\mathbb{w}$ while given the final output $(\mathbb{x}, \boldsymbol{\tau}, (\alpha_1, \ldots, \alpha_{\mathsf{k}}))$ of the state-restoration prover $\tilde{\mathbf{P}}^{\mathrm{sr}}$ and its move-response trace $\mathsf{tr}^{\mathrm{sr}}$. The knowledge extractor $\mathbf{E}^{\mathrm{sr}}$ does not receive the randomness $(\rho_i)_{i\in[\mathsf{k}]}$ used by the IP verifier.

Definition 3.13. $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *has* **straightline state-restoration knowledge soundness error** $\kappa^{\mathrm{sr}}_{\mathsf{IP}}$ *if there exists a polynomial-time deterministic algorithm* $\mathbf{E}^{\mathrm{sr}}$ *(the* extractor*) such that for every salt size* $\delta \in \mathbb{N}$*, move budget* $t \in \mathbb{N}$*,* t*-move deterministic IP state-restoration prover* $\tilde{\mathbf{P}}^{\mathrm{sr}}$*, and instance size bound* n*:*

$$\Pr\left[\begin{array}{l} |\mathbb{x}| \le n \\ \wedge\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \\ \wedge\ \mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}\big) = 1 \end{array} \,\middle|\, \begin{array}{l} \boldsymbol{f} = (f_i)_{i\in[\mathsf{k}]} \leftarrow \mathcal{U}((\mathcal{Z}_i(\delta,n))_{i\in[\mathsf{k}]}) \\ (\mathbb{x}, \boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}) \xleftarrow{\mathsf{tr}^{\mathrm{sr}}} \\ \quad \mathrm{SR}_{\mathsf{IP},f,\tilde{\mathbf{P}}^{\mathrm{sr}}}(\delta) \\ \mathbb{w} \leftarrow \mathbf{E}^{\mathrm{sr}}(\mathbb{x}, \boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]}, \mathsf{tr}^{\mathrm{sr}}) \end{array}\right] \le \kappa^{\mathrm{sr}}_{\mathsf{IP}}(\delta, t, n)\,.$$

The rewinding variant of state-restoration knowledge soundness relaxes the prior notion by considering a knowledge extractor that additionally receives the randomness $(\rho_i)_{i\in[\mathsf{k}]}$ used by the IP verifier and black-box access to the state-restoration prover $\tilde{\mathbf{P}}^{\mathrm{sr}}$. In this case, the error may additionally depend on the failure probability of $\tilde{\mathbf{P}}^{\mathrm{sr}}$ (an upper bound on the probability that the final output of $\tilde{\mathbf{P}}^{\mathrm{sr}}$ does *not* convince the IP verifier). Intuitively, as the failure probability increases, the error of extraction increases.

Definition 3.14. *Let* $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *be an IP. A deterministic IP state-restoration prover* $\tilde{\mathbf{P}}^{\mathrm{sr}}$ *has* **failure probability** $\delta_{\tilde{\mathbf{P}}^{\mathrm{sr}}}$ *if for every salt size* $\delta \in \mathbb{N}$ *and instance size bound* n*:*

$$\Pr\left[\begin{array}{l} |\mathbb{x}| \le n \\ \wedge\ \mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}\big) = 0 \end{array} \,\middle|\, \begin{array}{l} \boldsymbol{f} = (f_i)_{i\in[\mathsf{k}]} \leftarrow \mathcal{U}((\mathcal{Z}_i(\delta,n))_{i\in[\mathsf{k}]}) \\ (\mathbb{x}, \boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}) \leftarrow \\ \quad \mathrm{SR}_{\mathsf{IP},\boldsymbol{f},\tilde{\mathbf{P}}^{\mathrm{sr}}}(\delta) \end{array}\right] \le \delta_{\tilde{\mathbf{P}}^{\mathrm{sr}}}(\delta, n)\,.$$

Definition 3.15. $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *has* **rewinding state-restoration knowledge soundness error** $\kappa^{\text{sr}}_{\mathsf{IP}}$ **with extraction time** $\mathbf{et}^{\text{sr}}_{\mathsf{IP}}$ *if there exists a probabilistic algorithm* $\mathbf{E}^{\text{sr}}$ *(the* extractor*) such that for every salt size* $\delta \in \mathbb{N}$*, move budget* $t \in \mathbb{N}$*, t-move deterministic IP state-restoration prover* $\tilde{\mathbf{P}}^{\text{sr}}$ *with failure probability* $\delta_{\tilde{\mathbf{P}}^{\text{sr}}}$ *and running time* $\tau_{\tilde{\mathbf{P}}^{\text{sr}}}$*, and instance size bound* n*:*

$$\Pr\left[\begin{array}{l} |\mathbb{x}| \le n \\ \wedge\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \\ \wedge\ \mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}\big) = 1 \end{array} \middle| \begin{array}{l} \boldsymbol{f} = (f_i)_{i\in[\mathsf{k}]} \leftarrow \mathcal{U}((\mathcal{Z}_i(\delta, n))_{i\in[\mathsf{k}]}) \\ (\mathbb{x}, \boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}) \xleftarrow{\mathsf{tr}^{\text{sr}}} \\ \quad \mathrm{SR}_{\mathsf{IP}, \boldsymbol{f}, \tilde{\mathbf{P}}^{\text{sr}}}(\delta) \\ \mathbb{w} \leftarrow \mathbf{E}^{\text{sr}}(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, \boldsymbol{\tau}, (\rho_i)_{i\in[\mathsf{k}]}, \mathsf{tr}^{\text{sr}}, \tilde{\mathbf{P}}^{\text{sr}}) \end{array}\right]$$
$$\le \kappa^{\text{rsr}}_{\mathsf{IP}}(\delta, t, n, \delta_{\tilde{\mathbf{P}}^{\text{sr}}}(\delta, n)) \,.$$

Moreover, $\mathbf{E}^{\text{sr}}$ *runs in expected time* $\mathbf{et}^{\text{sr}}_{\mathsf{IP}}(\delta, t, n, \delta_{\tilde{\mathbf{P}}^{\text{sr}}}(\delta, n), \tau_{\tilde{\mathbf{P}}^{\text{sr}}}(\delta, n))$ *(over the given inputs and internal randomness).*

3.6 Fiat–Shamir transformation

Let $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ be a public-coin IP with round complexity k and message spaces $((\mathcal{M}_{\mathbf{P},i_i}, \mathcal{M}_{\mathbf{V},i}))_{i\in[\mathsf{k}]}$. Define the oracle distribution $\mathcal{D}_{\mathsf{IP}}$ with salt size δ as follows: for every security parameter $\lambda \in \mathbb{N}$ and an instance size bound $n \in \mathbb{N}$,

$$\mathcal{D}_{\mathsf{IP}}(\lambda, n) := \mathcal{U}\Big((\mathcal{Z}_i(\delta, n))_{i\in[\mathsf{k}]}\Big) \tag{4}$$

where $(\mathcal{Z}_i(\delta, n))_{i\in[\mathsf{k}]}$ are the function spaces in Definition 3.10. We present the canonical Fiat–Shamir transformation for public-coin interactive proofs. The construction here is a simplification of [CY24, Construction 14.1.1] that relies on a single salt rather than k salts. This simplification does not affect the soundness (and knowledge soundness) results given in that book.

Construction 3.16. *Let* $\delta \in \mathbb{N}$*. We define* $\mathsf{NARG} := \mathsf{FS}[\mathsf{IP}, \delta]$ *to be the non-interactive argument* $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ *in the* $\mathcal{D}_{\mathsf{IP}}$*-oracle model constructed as follows. The argument prover* $\mathcal{P}$ *receives as input an instance* $\mathbb{x}$ *and witness* $\mathbb{w}$*, and the argument verifier* $\mathcal{V}$ *receives as input the instance* $\mathbb{x}$ *and an argument string* π*. Both receive query access to oracles* $\boldsymbol{f}$ *sampled from* $\mathcal{D}_{\mathsf{IP}}(\lambda, n)$ *defined in Equation* 4*.*

- $\mathcal{P}^{\boldsymbol{f}}(\mathbb{x}, \mathbb{w})$*:*
  1. *Sample a random salt* $\tau \in \{0,1\}^{\delta}$*.*
  2. *For* $i = 1, \ldots, \mathsf{k}$*:*
     (a) *Compute the* i*-th message (and auxiliary state) of the IP prover:*
     $$(\alpha_i, \mathsf{aux}_i) := \begin{cases} \mathbf{P}(\mathbb{x}, \mathbb{w}) & \text{if } i = 1 \\ \mathbf{P}(\mathsf{aux}_{i-1}, \rho_{i-1}) & \text{if } i > 1 \end{cases} \,.$$
     (b) *If* $i < \mathsf{k}$*, derive the* i*-th random message of the IP verifier:*
     $$\rho_i := f_i\big(\mathbb{x}, \tau, \alpha_1, \ldots, \alpha_i\big) \,.$$
  3. *Output the argument string* $\pi := \big(\tau, (\alpha_i)_{i\in[\mathsf{k}]}\big)$*.*
- $\mathcal{V}^{f}(\mathbb{x}, \pi)$*:*
  1. *Parse the argument string* π *as a tuple* $\big(\tau, (\alpha_i)_{i\in[\mathsf{k}]}\big)$*.*

2. *For $i = 1, \dots, \mathsf{k}$, derive the i-th IP verifier message ρ_i (as $\mathcal{P}$ does):*

$$\rho_i \coloneqq f_i\big(\mathbb{x}, \tau, \alpha_1, \dots, \alpha_i\big) .$$

3. *Check that the IP verifier accepts: $\mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i \in [\mathsf{k}]}, (\rho_i)_{i \in [\mathsf{k}]}\big) = 1$.*

The canonical Fiat–Shamir transformation for public-coin IPs described above satisfies soundness and knowledge soundness, as we now explain. Theorem 3.17 below (adapted from [CY24]) links the soundness error of $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ to the state-restoration soundness error of $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$; similarly, Theorem 3.18 (also adapted from [CY24]) does the same for knowledge soundness.

Theorem 3.17. *If $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ has state-restoration soundness error $\varepsilon_{\mathsf{IP}}^{\mathrm{sr}}$ (see Definition 3.13) then $\mathsf{NARG} \coloneqq \mathsf{FS}[\mathsf{IP}, \delta]$ in Construction 3.16 has soundness error $\varepsilon_{\mathsf{NARG}}$ (see Definition 3.4) such that, for every security parameter $\lambda \in \mathbb{N}$, query bound $t \in \mathbb{N}$, and instance size bound $n \in \mathbb{N}$,*

$$\varepsilon_{\mathsf{NARG}}(\lambda, t, n) \leq \varepsilon_{\mathsf{IP}}^{\mathrm{sr}}(\delta, t, n) .$$

Theorem 3.18. *If IP has rewinding state-restoration knowledge soundness error $\kappa_{\mathsf{IP}}^{\mathrm{sr}}$ with extraction time $\mathbf{et}_{\mathsf{IP}}$ (see Definition 3.15) then $\mathsf{NARG} \coloneqq \mathsf{FS}[\mathsf{IP}, \delta]$ in Construction 3.16 has rewinding knowledge soundness error $\kappa_{\mathsf{NARG}}\big(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n)\big)$ with extraction time $\mathbf{et}_{\mathsf{NARG}}$ (see Definition 3.7) such that, for every security parameter $\lambda \in \mathbb{N}$, query bound $t \in \mathbb{N}$, and instance size bound $n \in \mathbb{N}$,*

- $\kappa_{\mathsf{NARG}}\big(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n)\big) \leq \kappa_{\mathsf{IP}}^{\mathrm{sr}}\left(\delta, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n)\right)$, *and*
- $\mathbf{et}_{\mathsf{NARG}}(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n), \tau_{\tilde{\mathcal{P}}}(\lambda, n)) \leq \mathbf{et}_{\mathsf{IP}}^{\mathrm{sr}}\left(\delta, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n), \tau_{\tilde{\mathcal{P}}}(\lambda, n) + O(\mathsf{r}_{\mathsf{max}} \cdot t)\right) + O(\mathsf{r}_{\mathsf{max}} \cdot t)$.

Moreover, if the IP state-restoration extractor is straightline (see Definition 3.13) then the NARG extractor is also straightline (see Definition 3.5). In this case:

- *the (straightline) knowledge soundness error is $\kappa_{\mathsf{NARG}}(\lambda, t, n) \leq \kappa_{\mathsf{IP}}^{\mathrm{sr}}(\delta, t, n)$; and*
- *the (straightline) extraction time is $\mathbf{et}_{\mathsf{NARG}}(\lambda, t, n) \leq \mathbf{et}_{\mathsf{IP}}^{\mathrm{sr}}(\delta, t, n) + O(\mathsf{r}_{\mathsf{max}} \cdot t)$.*

Above $\mathsf{r}_{\mathsf{max}} \coloneqq \max_{i \in [\mathsf{k}]} \log_2 \left|\mathcal{M}_{\mathbf{V},i}\right|$ denotes the maximum verifier randomness length.

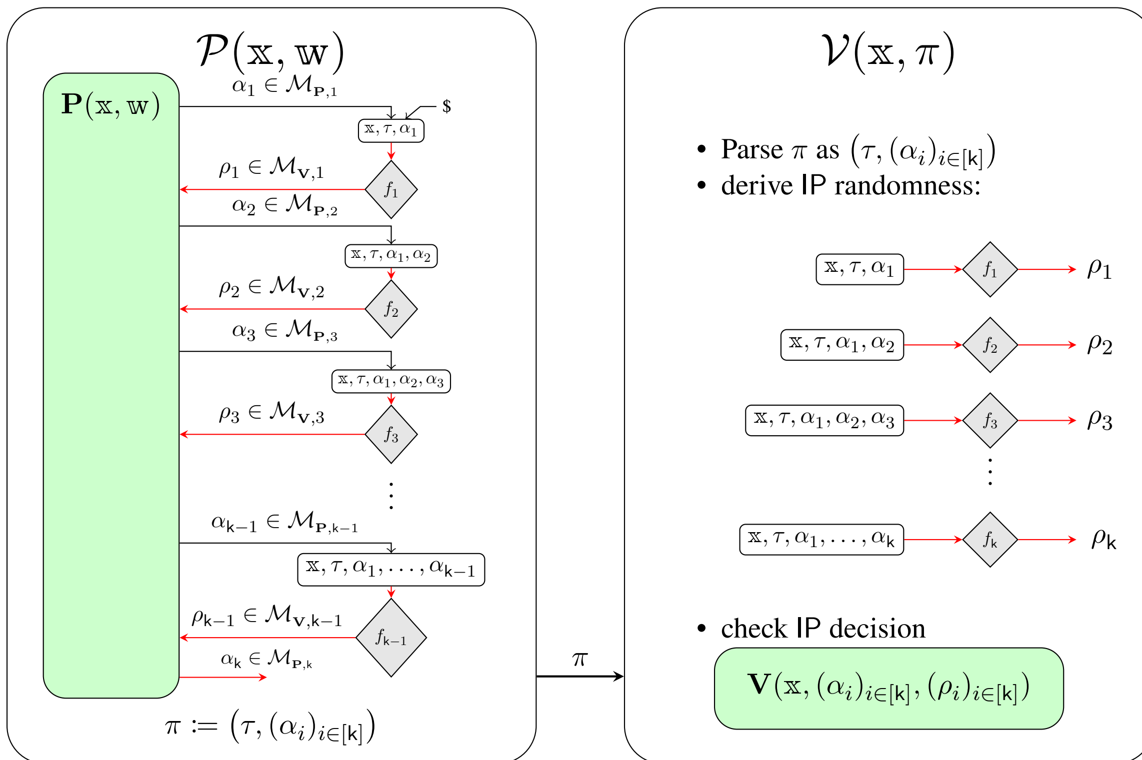

Figure 2: Diagram of $\mathsf{FS}[\mathsf{IP}, \delta]$ as defined in Construction 3.16.

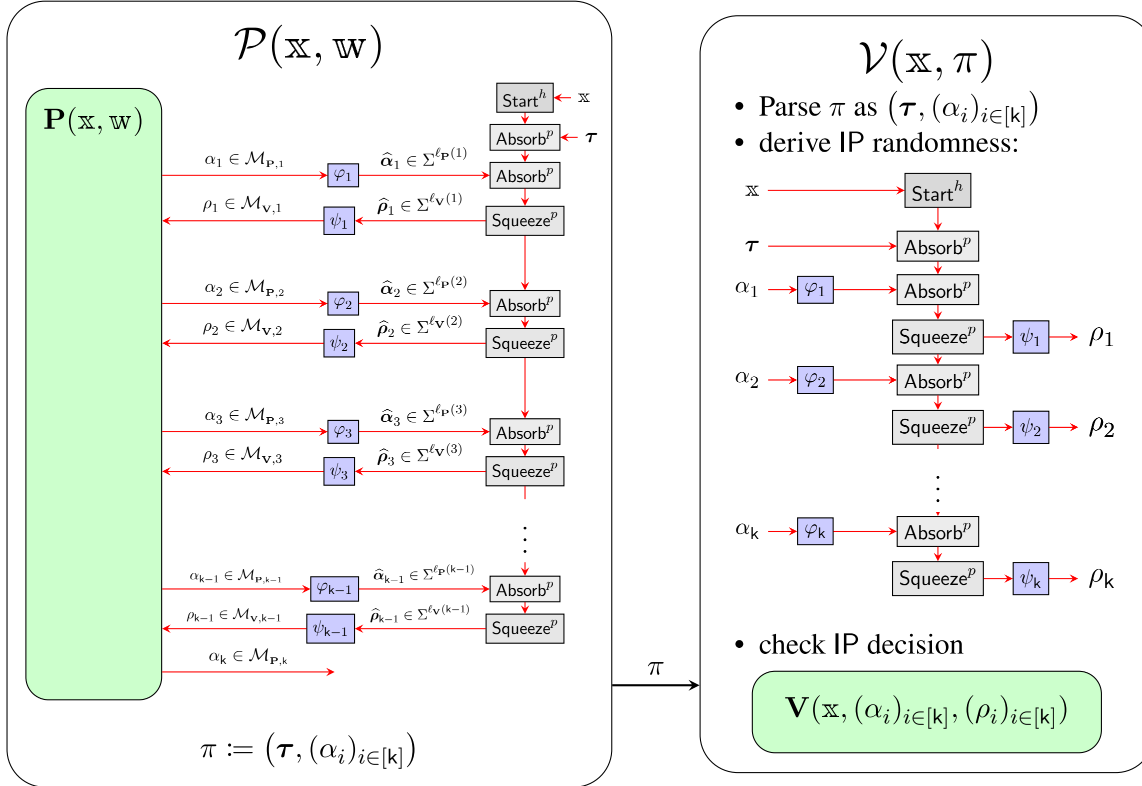

Figure 3: Diagram of $\mathsf{DSFS}[\mathsf{IP}, \delta]$ as defined in Construction 4.3.

4 Duplex-sponge Fiat–Shamir transformation

We describe the Fiat–Shamir transformation that we propose in this paper; the transformation is based on the (ideal) duplex sponge described in Section 3.3. The main security reduction for the transformation is in Section 5, from which in Section 6 we deduce soundness and knowledge soundness. Separately, in Section 7, we establish the zero knowledge property for the transformation.

First we give the definition of a *codec* which is a way to encode prover messages and decode verifier messages relative to a given alphabet Σ.

Definition 4.1. *Let Σ be a finite alphabet and $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ be an IP with round complexity k and message spaces $((\mathcal{M}_{\mathbf{P},i}, \mathcal{M}_{\mathbf{V},i}))_{i\in[\mathsf{k}]}$. A* **codec** *for IP over Σ with bias $\varepsilon_{\mathsf{cdc}}$ is a function cdc that maps every security parameter $\lambda \in \mathbb{N}$ and instance size bound $n \in \mathbb{N}$ to a tuple*

$$\mathsf{cdc}(\lambda, n) = (\ell_{\mathbf{P}}, \ell_{\mathbf{V}}, \boldsymbol{\varphi}, \boldsymbol{\psi})$$

where

- *$\ell_{\mathbf{P}}, \ell_{\mathbf{V}} : \mathbb{N} \to \mathbb{N}$ are length functions for each round,*
- *$\boldsymbol{\varphi}$ is a list of injective maps $(\varphi_i : \mathcal{M}_{\mathbf{P},i} \to \Sigma^{\ell_{\mathbf{P}}(i)})_{i\in[\mathsf{k}]}$, and*
- *$\boldsymbol{\psi}$ is a list of maps $(\psi_i : \Sigma^{\ell_{\mathbf{V}}(i)} \to \mathcal{M}_{\mathbf{V},i})_{i\in[\mathsf{k}]}$ where, for each $i \in [\mathsf{k}]$, ψ_i is $\varepsilon_{\mathsf{cdc},i}(\lambda, n)$-biased (the distributions $\psi_i(\mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)}))$ and $\mathcal{U}(\mathcal{M}_{\mathbf{V},i})$ are $\varepsilon_{\mathsf{cdc},i}(\lambda, n)$-close in statistical distance).*[11]

Next we give the definition of the oracle distribution $\mathcal{D}_{\mathfrak{S}}(\lambda, n)$ that we consider.

Definition 4.2. *The* **ideal permutation oracle distribution** *$\mathcal{D}_{\mathfrak{S}}$ over alphabet Σ with capacity $c \in \mathbb{N}$, and rate $r \in \mathbb{N}$ is defined as follows: $\mathcal{D}_{\mathfrak{S}}(\lambda, n)$ outputs (h, p, p^{-1}) where $h : \{0,1\}^{\leq n} \to \Sigma^c$ is a random function, $p : \Sigma^{r+c} \to \Sigma^{r+c}$ is a random permutation, and p^{-1} is the inverse of p.*

Finally we describe our Fiat–Shamir transformation. (Note: p^{-1} will only be used by the adversary and is not present.)

Construction 4.3. *Let $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ be a public-coin IP with round complexity k. Let Σ be a finite alphabet and cdc a codec for IP over Σ (the codec's bias does not matter for describing the construction). Let $\mathcal{D}_{\mathfrak{S}}$ be the ideal permutation oracle distribution over Σ with capacity $c \in \mathbb{N}$ and rate $r \in \mathbb{N}$. For a salt size $\delta \in \mathbb{N}$, the non-interactive argument $\mathsf{NARG} = \mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ in the $\mathcal{D}_{\mathfrak{S}}$-oracle model is the non-interactive argument $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ constructed as follows. The argument prover $\mathcal{P}$ receives as input an instance $\mathbb{x}$ and witness $\mathbb{w}$, and the argument verifier $\mathcal{V}$ receives as input the instance $\mathbb{x}$ and an argument string π. Both receive query access to oracles (h, p) sampled from $\mathcal{D}_{\mathfrak{S}}(\lambda, n)$ defined in Definition 4.2. Let $(\ell_{\mathbf{P}}, \ell_{\mathbf{V}}, \boldsymbol{\varphi}, \boldsymbol{\psi}) := \mathsf{cdc}(\lambda, n)$.*

- *$\mathcal{P}^{h,p}(\mathbb{x}, \mathbb{w})$:*
  1. *Initialize the sponge state with the instance: $\mathsf{st}'_0 := \mathsf{DS.Start}^h(\mathbb{x}) \in \Sigma^{r+c} \times [0, r] \times [0, r]$.*
  2. *Sample a random salt $\boldsymbol{\tau} \in \Sigma^{\delta}$.*
  3. *Absorb the salt: $\mathsf{st}_0 := \mathsf{Absorb}^p(\mathsf{st}'_0, \boldsymbol{\tau}) \in \Sigma^{r+c} \times [0, r] \times [0, r]$.*
  4. *For $i = 1, \ldots, \mathsf{k}$:*
     (a) *Compute the i-th message (and auxiliary state) of the IP prover:*

$$(\alpha_i, \mathsf{aux}_i) := \begin{cases} \mathbf{P}(\mathbb{x}, \mathbb{w}) \in \mathcal{M}_{\mathbf{P},1} & \text{if } i = 1 \\ \mathbf{P}(\mathsf{aux}_{i-1}, \rho_{i-1}) \in \mathcal{M}_{\mathbf{P},i} & \text{if } i > 1 \end{cases} .$$

[11] I.e., ψ_i maps the uniform distribution on $\Sigma^{\ell_{\mathbf{V}}(i)}$ to a distribution that is $\varepsilon_{\mathsf{cdc},i}(\lambda, n)$-close to the uniform distribution on $\mathcal{M}_{\mathbf{V},i}$.

(b) If $i < \mathsf{k}$, encode the prover message: $\widehat{\boldsymbol{\alpha}}_i \coloneqq \varphi_i(\alpha_i) \in \Sigma^{\ell_{\mathbf{P}}(i)}$.

(c) If $i < \mathsf{k}$, absorb the encoded prover message:

$$\mathsf{st}'_i \coloneqq \mathsf{DS.Absorb}^p(\mathsf{st}_{i-1}, \widehat{\boldsymbol{\alpha}}_i) \in \Sigma^{r+c} \times [0,r] \times [0,r] \,.$$

(d) If $i < \mathsf{k}$, squeeze the encoded verifier message:

$$(\widehat{\boldsymbol{\rho}}_i, \mathsf{st}_i) \coloneqq \mathsf{DS.Squeeze}^p(\mathsf{st}'_i, \ell_{\mathbf{V}}(i)) \in \Sigma^{\ell_{\mathbf{V}}(i)} \times \Sigma^{r+c} \times [0,r] \times [0,r] \,.$$

(e) If $i < \mathsf{k}$, decode the verifier message: $\rho_i \coloneqq \psi_i(\widehat{\boldsymbol{\rho}}_i) \in \mathcal{M}_{\mathbf{V},i}$.

5. Output the argument string $\pi \coloneqq (\boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]})$.

- $\mathcal{V}^{h,p}(\mathbb{x}, \pi)$:

1. Parse the argument string π as a salt $\boldsymbol{\tau} \in \Sigma^\delta$ and prover messages $(\alpha_i)_{i\in[\mathsf{k}]} \in \mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},\mathsf{k}}$.

2. Initialize the sponge state with the instance (as $\mathcal{P}$ does): $\mathsf{st}'_0 \coloneqq \mathsf{DS.Start}^h(\mathbb{x}) \in \Sigma^{r+c} \times [0,r] \times [0,r]$.

3. Absorb the salt (as $\mathcal{P}$ does): $\mathsf{st}_0 \coloneqq \mathsf{Absorb}^p(\mathsf{st}'_0, \boldsymbol{\tau}) \in \Sigma^{r+c} \times [0,r] \times [0,r]$.

4. For $i = 1, \ldots, \mathsf{k}$, derive the i-th IP verifier message ρ_i (as $\mathcal{P}$ does):

(a) Encode the prover message: $\widehat{\boldsymbol{\alpha}}_i \coloneqq \varphi_i(\alpha_i) \in \Sigma^{\ell_{\mathbf{P}}(i)}$.

(b) Absorb the encoded prover message:

$$\mathsf{st}'_i \coloneqq \mathsf{DS.Absorb}^p(\mathsf{st}_{i-1}, \widehat{\boldsymbol{\alpha}}_i) \in \Sigma^{r+c} \times [0,r] \times [0,r] \,.$$

(c) Squeeze the encoded verifier message:

$$(\widehat{\boldsymbol{\rho}}_i, \mathsf{st}_i) \coloneqq \mathsf{DS.Squeeze}^p(\mathsf{st}'_i, \ell_{\mathbf{V}}(i)) \in \Sigma^{\ell_{\mathbf{V}}(i)} \times \Sigma^{r+c} \times [0,r] \times [0,r] \,.$$

(d) Decode the verifier message: $\rho_i \coloneqq \psi_i(\widehat{\boldsymbol{\rho}}_i) \in \mathcal{M}_{\mathbf{V},i}$.

5. Check that the IP verifier accepts: $\mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}\big) = 1$.

Efficiency. We discuss efficiency properties of the above constructions.

- *Argument size.* The argument string π contains a salt $\boldsymbol{\tau} \in \Sigma^\delta$ and all IP prover messages (and none of the IP verifier messages). Hence the number of bits in π is

$$\operatorname{len}(\boldsymbol{\tau}) + \sum_{i\in[\mathsf{k}]} \operatorname{len}(\alpha_i) = \delta \cdot \log_2 |\Sigma| + \sum_{i\in[\mathsf{k}]} \log_2 \left|\mathcal{M}_{\mathbf{P},i}\right| \,.$$

- *Prover complexity.* The cost of the argument prover $\mathcal{P}$ is essentially the same as that of the underlying IP prover $\mathbf{P}$. The difference is that the argument prover $\mathcal{P}$ additionally:
  - makes 1 query of length n to the oracle h, in the call to DS.Start;
  - makes $\left\lceil \frac{\delta+\ell_{\mathbf{P}}(1)}{r} \right\rceil + \sum_{i=2}^{\mathsf{k}-1} \left\lceil \frac{\ell_{\mathbf{P}}(i)}{r} \right\rceil$ queries of length $r+c$ to the oracle p, across calls to DS.Absorb;
  - makes $\sum_{i=1}^{\mathsf{k}-1} \left\lceil \frac{\ell_{\mathbf{V}}(i)}{r} \right\rceil$ queries of length $r+c$ to the oracle p, across calls to DS.Squeeze;
  - invokes once each of $(\varphi_i)_{i\in[\mathsf{k}-1]}$ and $(\psi_i)_{i\in[\mathsf{k}-1]}$.

- *Verifier complexity.* The cost of the argument verifier $\mathcal{V}$ is essentially the same as that of the underlying IP verifier $\mathbf{V}$. The difference is that the argument verifier $\mathcal{V}$ additionally:
  - makes the same queries to h and p as $\mathcal{P}$ does;
  - makes $\left\lceil \frac{\ell_{\mathbf{P}}(\mathsf{k})}{r} \right\rceil$ queries of length $r+c$ to the oracle p, due to another call to DS.Absorb;
  - makes $\left\lceil \frac{\ell_{\mathbf{V}}(\mathsf{k})}{r} \right\rceil$ queries of length $r+c$ to the oracle p, due to another call to DS.Squeeze;
  - invokes once each of $(\varphi_i)_{i\in[\mathsf{k}]}$ and $(\psi_i)_{i\in[\mathsf{k}]}$.

5 Security analysis

We provide the main security reduction in this paper: we reduce the security of the duplex-sponge Fiat–Shamir transformation to the security of the basic Fiat–Shamir transformation. In Section 6 we show that this directly implies the soundness and knowledge soundness of the duplex-sponge Fiat–Shamir transformation.

Let $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ be a public-coin IP for a relation $\mathcal{R}$ with round complexity k. Let Σ be a finite alphabet and cdc a codec for IP over Σ with bias $\varepsilon_{\mathsf{cdc}}$ (see Definition 4.1). Let $\mathcal{D}_{\mathfrak{S}}$ be an ideal permutation oracle distribution over Σ with capacity $c \in \mathbb{N}$ and rate $r \in \mathbb{N}$ (see Definition 4.2). Let $\delta \in \mathbb{N}$ be a salt size (in Σ-elements) and $\delta_\star := \delta \log_2 |\Sigma|$ be its corresponding bit-size.

Consider the following two non-interactive arguments:

- $(\mathcal{P}, \mathcal{V}) := \mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ is the non-interactive argument in the $\mathcal{D}_{\mathfrak{S}}$-oracle model in Construction 4.3;
- $(\mathcal{P}_{\mathsf{std}}, \mathcal{V}_{\mathsf{std}}) := \mathsf{FS}[\mathsf{IP}, \delta_\star]$ is the non-interactive in the $\mathcal{D}_{\mathsf{IP}}$-oracle model in Construction 3.16.

We reduce the security of $(\mathcal{P}, \mathcal{V})$ to the security of $(\mathcal{P}_{\mathsf{std}}, \mathcal{V}_{\mathsf{std}})$ via Lemma 5.1 below. Informally, Lemma 5.1 shows that any $(t_h, t_p, t_{p^{-1}})$-query malicious argument prover $\tilde{\mathcal{P}}$ for $\mathcal{V}$ can be transformed, in a black-box way via an auxiliary procedure D2SAlgo defined in Section 5.3, into a $\theta_\star(t_h, t_p, t_{p^{-1}})$-query malicious argument prover $\tilde{\mathcal{P}}_{\mathsf{std}}$ for $\mathcal{V}_{\mathsf{std}}$ (where $\theta_\star(t_h, t_p, t_{p^{-1}})$ is defined below) that "behaves the same" as $\mathcal{P}$ up to a certain additive error $\eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))$. Specifically, the output instance, argument string, and convincing probability are preserved, as are the query-answer traces (after an appropriate transformation by an auxiliary procedure D2STrace to account for the differing constructions).

Lemma 5.1. *There exist algorithms* D2SAlgo *and* D2STrace *such that the following holds: for every security parameter $\lambda \in \mathbb{N}$, instance size bound $n \in \mathbb{N}$, and $(t_h, t_p, t_{p^{-1}})$-query argument prover $\tilde{\mathcal{P}}$, the two distributions below*

$(h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n)$	$\boldsymbol{f} \leftarrow \mathcal{D}_{\mathsf{IP}}(\lambda, n)$
$(\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}}$	$(\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}_\star}} \mathsf{D2SAlgo}^{\boldsymbol{f}}(\tilde{\mathcal{P}})$
$b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi)$	$b \xleftarrow{\mathsf{tr}_{\mathcal{V}_{\mathsf{std}}}} \mathcal{V}_{\mathsf{std}}^{\boldsymbol{f}}(\mathbb{x}, \pi)$
$\mathsf{tr} := \mathsf{D2STrace}(\mathsf{tr}_{\tilde{\mathcal{P}}} \Vert \mathsf{tr}_{\mathcal{V}})$	$\mathsf{tr} := \mathsf{tr}_{\tilde{\mathcal{P}}_\star} \Vert \mathsf{tr}_{\mathcal{V}_{\mathsf{std}}}$
return $(b, \mathbb{x}, \pi, \mathsf{tr})$	**return** $(b, \mathbb{x}, \pi, \mathsf{tr})$

have statistical distance at most:

$$\eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) := \frac{t_h(t_h + 3t_p + 2L_{\mathbf{P}} + 1) + (t_p + L_{\mathbf{P}})(2t_p + 3L_{\mathbf{P}} - 1) + t_{p^{-1}}(2t_h + 2t_p + t_{p^{-1}} - 1)}{2\,|\Sigma|^c} + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n)\,. \tag{5}$$

Moreover, $\mathsf{D2SAlgo}(\tilde{\mathcal{P}})$ *is a $\theta_\star(t_h, t_p, t_{p^{-1}})$-query algorithm where*

$$\theta_\star(t_h, t_p, t_{p^{-1}}) := \left\lceil \frac{t_p + t_{p^{-1}}}{\min_{i \in [\mathsf{k}]} \left\lceil \frac{\ell_{\mathbf{P}}(i)}{r} \right\rceil} \right\rceil. \tag{6}$$

We prove the lemma in Section 5.6. Before proving it, we define the two algorithms D2SAlgo and D2STrace in Sections 5.3 and 5.4. In turn, these rely on two other algorithms, BackTrack and LookAhead, defined in Sections 5.1 and 5.2. After defining them, we study the probability that these procedures abort in Section 5.5 to set the stage for the main lemma.

Throughout, we use the following notation to count number of blocks:

$$L_\delta := \lceil \delta/r \rceil \quad , \quad L_{\mathbf{P}}(i) := \lceil \ell_{\mathbf{P}}(i)/r \rceil \quad , \quad L_{\mathbf{V}}(i) := \lceil \ell_{\mathbf{V}}(i)/r \rceil \, . \tag{7}$$

5.1 Backtracking procedure

The backtracking procedure $\mathsf{BackTrack}(\mathsf{tr}_h, \mathsf{tr}_p, \boldsymbol{s})$ takes as input query-answer traces tr_h and tr_p and a permutation state $\boldsymbol{s} = (\boldsymbol{s}_{\mathrm{R}}, \boldsymbol{s}_{\mathrm{C}}) \in \Sigma^{r+c}$, and returns a special symbol (err or none) or a tuple $(i, \mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}})$ with a round index $i \in [\mathsf{k}]$, an instance $\mathbb{x} \in \{0,1\}^n$, a salt string $\boldsymbol{\tau} \in \Sigma^\delta$, and a tuple $(\widehat{\boldsymbol{\alpha}}_j, \ldots, \widehat{\boldsymbol{\alpha}}_j) \in \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)}$ of encoded prover messages. Informally, BackTrack uses the query-answer traces to recover from the permutation state $\boldsymbol{s}$ the list of elements absorbed so far, returning the current round index i and absorbed instance $\mathbb{x}$, salt string $\boldsymbol{\tau}$, and encoded prover messages $\widehat{\boldsymbol{\alpha}}$.

$\mathsf{BackTrack}(\mathsf{tr}_h, \mathsf{tr}_p, \boldsymbol{s})$:

1. Initialize an empty list Outs := [].
2. Parse the query-answer traces $\mathsf{tr}_h, \mathsf{tr}_p$ into a list

   $$\mathsf{Chains} := \left(\mathbb{x}^{(j)}, (\boldsymbol{s}^{(j)}_{\text{in},0}, \boldsymbol{s}^{(j)}_{\text{out},0}, \boldsymbol{s}^{(j)}_{\text{in},1}, \boldsymbol{s}^{(j)}_{\text{out},1}, \ldots, \boldsymbol{s}^{(j)}_{\text{in},m_j}, \boldsymbol{s}^{(j)}_{\text{out},m_j}, \boldsymbol{s}^{(j)}_{\text{in},m_j})\right)_j \tag{8}$$

   where, for every $j \in [|\mathsf{Chains}|]$:
   (a) $(\mathbb{x}^{(j)}, \boldsymbol{s}^{(j)}_{\mathrm{C},\text{in},0}) \in \mathsf{tr}_h$ (i.e., $\boldsymbol{s}^{(j)}_{\mathrm{C},\text{in},0} = h(\mathbb{x}^{(j)})$).
   (b) $\boldsymbol{s}^{(j)}_{\text{in},m_j} = \boldsymbol{s}$ (i.e., the last capacity segment is the input query).
   (c) $\forall\, \iota \in [0, m_j]$, $(\boldsymbol{s}^{(j)}_{\text{in},\iota}, \boldsymbol{s}^{(j)}_{\text{out},\iota}) \in \mathsf{tr}_p$ (i.e., $\boldsymbol{s}^{(j)}_{\text{out},\iota} = p(\boldsymbol{s}^{(j)}_{\text{in},\iota})$).
   (d) $\forall\, \iota \in [0, m_j{-}1]$, $\boldsymbol{s}^{(j)}_{\mathrm{C},\text{out},\iota} = \boldsymbol{s}^{(j)}_{\mathrm{C},\text{in},\iota+1}$ (i.e., the capacity segments of $\boldsymbol{s}^{(j)}_{\text{out},\iota}$ and $\boldsymbol{s}^{(j)}_{\text{in},\iota+1}$ equal).

   Each element in Chains is a candidate sequence of absorb and squeeze operations.
3. For every $j \in [|\mathsf{Chains}|]$, assemble a candidate salt $\boldsymbol{\tau}^{(j)}$:

   $$\boldsymbol{\tau}^{(j)} := (\boldsymbol{s}^{(j)}_{\mathrm{R},\text{in},0} \| \cdots \| \boldsymbol{s}^{(j)}_{\mathrm{R},\text{in},L_\delta})[0\colon \delta] \in \Sigma^\delta \, .$$

4. For every $j \in [|\mathsf{Chains}|]$ and every $i \in [\mathsf{k}]$, assemble a candidate list of encoded prover messages:
   - Let $w^{(i)} := L_\delta + \sum_{\iota < i} L_{\mathbf{P}}(\iota) + L_{\mathbf{V}}(\iota)$ be the current "state offset".
   - If $w^{(i)} + L_{\mathbf{P}}(i) \neq m_j$, then the j-th candidate has not yet fully absorbed the i-th input in full and the candidate is not valid.
     Discard the j-th element of Chains from the list and continue to the next element in the list (if any).
   - If $w^{(i)} + L_{\mathbf{P}}(i) \leq m_j$ then we read $L_{\mathbf{P}}(i)$ rate segments and interpret them as:

     $$\widehat{\boldsymbol{\alpha}}^{(j)}_i := (\boldsymbol{s}^{(j)}_{\mathrm{R},\text{in},w^{(i)}} \| \boldsymbol{s}^{(j)}_{\mathrm{R},\text{in},w^{(i)}+1} \| \cdots \| \boldsymbol{s}^{(j)}_{\mathrm{R},\text{in},w^{(i)}+L_{\mathbf{P}}(i)})[0\colon \ell_{\mathbf{P}}(i)] \in \Sigma^{\ell_{\mathbf{P}}(j)} \, . \tag{9}$$

   - Let $\mathbf{z}^{(j)}_i := \boldsymbol{s}^{(j)}_{\mathrm{R},\text{in},w^{(i)}+L_{\mathbf{P}}(i)}[\ell_{\mathbf{P}}(i) \bmod r\colon r]$.

- Check that the remainder of the message is equal to the previous output:

$$\mathbf{z}_i^{(j)} = (\boldsymbol{s}^{(j)}_{\mathrm{R,out},w^{(i)}+L_{\mathbf{P}}(i)})[\ell_{\mathbf{P}}(i) \bmod r : r]\,.$$

If the above does not hold, discard the j-th element from Chains from the list and continue to the next element in the list (if any).
- If $i > 1$, check that the previous verifier message was squeezed correctly by checking that the rate part is preserved across invocations of the permutation function (to match the overwrite mode of the sponge):

$$\boldsymbol{s}^{(j)}_{\mathrm{R,out},i-1} = \boldsymbol{s}^{(j)}_{\mathrm{R,in},i-1}\,.$$

If the above does not hold, discard the j-th element from Chains from the list and continue to the next element in the list (if any).
- If $w^{(i)} + L_{\mathbf{P}}(i) = m_j$, store the message read so far as $(i, \mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}})$ into Outs, where $\widehat{\boldsymbol{\alpha}}$ is a collection of i vectors, indexed in ι, such that the ι-element is a vector of length $\ell_{\mathbf{P}}(\iota)$ from Eq. 9. Otherwise, discard the j-th element from Chains from the list and continue to the next element in the list (if any).

5. Final output:
   - If Outs contains more than one element, return err.
   - If Outs contains no element, return none.
   - Else, return the only element in Outs, $(i, \mathbb{x}^{(1)}, \boldsymbol{\tau}^{(1)}, \widehat{\boldsymbol{\alpha}}^{(1)})$.

Time complexity. We assume that the caller of BackTrack provides sorted query-answer traces:
- tr_h is assumed to be sorted lexicographically by output capacity segment, and
- tr_p is assumed to be sorted by output capacity segment, then output rate segment, then input capacity segment, then input rate segment.

In particular, this requires the alphabet Σ to possess an order.

The time complexity of BackTrack is dominated by the time to construct Chains in Eq. 8. Constructing an element of Chains involves at worst $|\mathsf{tr}_p| \leq t_p + t_{p^{-1}}$ lookups. Each lookup takes time $(\log(t_p + t_{p^{-1}}) + \log t_h) \cdot c \cdot \log|\Sigma|$ to find the element via dichotomic search. The last lookup is done over the trace tr_h and takes time $\log t_h \cdot c \cdot \log|\Sigma|$. Once a chain is constructed, each block (of length $r + c$) and the initial instance (of length n) must be checked according to step 4 to be consistent with the construction $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$. The search can stop when it encounters two "conflicting" chains. Overall, the total running time is $O((t_p + t_{p^{-1}}) \cdot (\log(t_p + t_{p^{-1}}) + \log t_h) \cdot c \cdot \log|\Sigma| \cdot (r + c + n))$.

5.2 Lookahead procedure

The procedure $\mathsf{LookAhead}(\mathsf{tr}_p, \boldsymbol{s}, i)$ takes as input a query-answer trace tr_p, a permutation state $\boldsymbol{s} = (\boldsymbol{s}_{\mathrm{R}}, \boldsymbol{s}_{\mathrm{C}}) \in \Sigma^{r+c}$, a round index $i \in [\mathsf{k}]$, and returns a special symbol (err or none) or a vector $\widehat{\boldsymbol{\rho}}_i \in \Sigma^{\ell_{\mathbf{V}}(i)}$. Informally, LookAhead recovers an encoded verifier message $\widehat{\boldsymbol{\rho}}_i$ obtained from a hash chain starting at $\boldsymbol{s}$.

$\mathsf{LookAhead}(\mathsf{tr}_p, \boldsymbol{s}, i)$:
1. Parse the query-answer list of tr_p into a list

$$\mathsf{Chains} := \Big((\boldsymbol{s}^{(j)}_{\mathrm{in},0}, \boldsymbol{s}^{(j)}_{\mathrm{out},0}), (\boldsymbol{s}^{(j)}_{\mathrm{in},1}, \boldsymbol{s}^{(j)}_{\mathrm{out},1}), \ldots, (\boldsymbol{s}^{(j)}_{\mathrm{in},L_{\mathbf{V}}(i)}, \boldsymbol{s}^{(j)}_{\mathrm{out},L_{\mathbf{V}}(i)})\Big)_{j=1}^{m}$$

where, for every $j \in [m]$:

- $\boldsymbol{s}^{(j)}_{\text{in},0} = \boldsymbol{s}$;
- $\forall\, \iota \in [0, L_{\mathbf{V}}(i){-}1]$, $\boldsymbol{s}^{(j)}_{\text{out},\iota} = \boldsymbol{s}^{(j)}_{\text{in},\iota+1}$ (i.e., the ι-th output state equals the $(\iota+1)$-th input state).

2. Final output:
   - If Chains is empty then return none.
   - If Chains contains more than one element ($m > 1$) then return err.
   - If Chains contains exactly one element ($m = 1$) then return

$$\widehat{\boldsymbol{\rho}}_i := (\boldsymbol{s}_{\text{R},\text{out},0} \| \boldsymbol{s}_{\text{R},\text{out},1} \| \cdots \| \boldsymbol{s}_{\text{R},\text{out},L_{\mathbf{V}}(i)})[0\colon \ell_{\mathbf{V}}(i)] \in \Sigma^{\ell_{\mathbf{V}}(i)} .$$

Time complexity. The time of $\mathsf{LookAhead}$ is dominated by Item 1. Producing an element in Chains involves, at worst, $L_{\mathbf{V}}(i)$ lookups in tr_p over elements Σ^{r+c}. Each lookup can be performed via a dichotomic search. The caller of $\mathsf{LookAhead}$ will provide tr_p sorted: tr_p is a list of query-answer pairs of the form $((\boldsymbol{s}_{\text{R},\text{in},i}, \boldsymbol{s}_{\text{C},\text{in},i}), (\boldsymbol{s}_{\text{R},\text{out},i}, \boldsymbol{s}_{\text{C},\text{out},i}))$ sorted lexicographically by input (capacity segment, then rate segment) and then by output (capacity segment, then rate segment). The search stops when it encounters two "conflicting" chains. Therefore, the overall time complexity is $O(L_{\mathbf{V}}(i) \cdot (r + c) \cdot \log|\Sigma| \cdot \log(t_p + t_{p^{-1}}))$.

5.3 Prover transformation

The auxiliary procedure $\mathsf{D2SAlgo}$ translates queries of a malicious prover $\tilde{\mathcal{P}}$ for $\mathcal{V}$ into oracle queries of a malicious prover $\tilde{\mathcal{P}}_{\mathsf{std}}$ for $\mathcal{V}_{\mathsf{std}}$.

The procedure $\mathsf{D2SAlgo}$ not only has to deal with converting the queries for oracles sampled from $\mathcal{D}_{\mathfrak{S}}(\lambda, n)$ to oracles sampled from $\mathcal{D}_{\mathsf{IP}}(\lambda, n)$, but also has to deal with the fact that the prover messages are encoded differently. To reduce the complexity of this step, we first define $\mathsf{D2SQuery}$ (re-mapping oracle queries for $\mathsf{D2SAlgo}$) to deal with most of the complexity of translating the oracle queries, and then define $\mathsf{D2SAlgo}$ as a thin wrapper around it, dealing with the encoding of the prover messages.

Oracle wrapper $\mathsf{D2SQuery}$. The oracle $\mathsf{D2SQuery}$ responds to oracle queries for (h, p, p^{-1}) by translating and forwarding them to oracles of the form

$$\boldsymbol{g} := (g_i)_{i=1}^{\mathsf{k}} \leftarrow \mathcal{U}\left((\{0,1\}^{\leq n} \times \Sigma^{\delta} \times \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)} \to \Sigma^{\ell_{\mathbf{V}}(i)})_{i \in [\mathsf{k}]} \right) .$$

Looking ahead, such oracles $(g_i)_{i=1}^{\mathsf{k}}$ will be implemented using oracles sampled from $\mathcal{D}_{\mathsf{IP}}(\lambda, n)$ and the codec maps $\boldsymbol{\varphi}$, $\boldsymbol{\psi}$. The oracle $\mathsf{D2SQuery}$ works as follows:

1. Initialize list Cache_p, sorted lists $\mathsf{tr}, \mathsf{tr}_h^{\mathsf{BT}}, \mathsf{tr}_p^{\mathsf{BT}}$.
   The lists $\mathsf{tr}_h^{\mathsf{BT}}, \mathsf{tr}_p^{\mathsf{BT}}$ are sorted lexicographically by output capacity segment, then output rate capacity segment, then input segment.
   The list tr will host queries to h and p, stored as triplet $(`h', \mathbb{x}, \boldsymbol{s}_{\text{C}})$ for queries to $h$ and as pairs $(`p', \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ for queries to p, ordered by query time of the adversary.
2. For every query $\mathbb{x} \in \{0,1\}^{\leq n}$ to the oracle h:
   (a) Let $\boldsymbol{s}_{\text{C},\text{out}} \in \Sigma^c$ be the first to match $(\mathbb{x}, \boldsymbol{s}_{\text{C},\text{out}}) \in \mathsf{tr}_h^{\mathsf{BT}}$, if any.
   (b) Else, sample $\boldsymbol{s}_{\text{C},\text{out}} \leftarrow \mathcal{U}(\Sigma^c)$ and add $(\mathbb{x}, \boldsymbol{s}_{\text{C},\text{out}})$ to $\mathsf{tr}_h^{\mathsf{BT}}$.
   (c) Add $(\mathbb{x}, \boldsymbol{s}_{\text{C},\text{out}})$ to tr_h.
   (d) Add $(`h', \mathbb{x}, \boldsymbol{s}_{\text{C},\text{out}})$ to tr and return $\boldsymbol{s}_{\text{C},\text{out}}$.
3. For every query $\boldsymbol{s}_{\text{out}} \in \Sigma^{r+c}$ to the oracle p^{-1}:
   (a) If $\exists\, \boldsymbol{s}_{\text{in}}$ such that $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \mathsf{tr}_p^{\mathsf{BT}}$, then return $\boldsymbol{s}_{\text{in}}$.

(b) Else, sample $\boldsymbol{s}_{\text{in}} \leftarrow \mathcal{U}(\Sigma^{r+c})$.
(c) Add $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ to $\mathsf{tr}_p^{\mathsf{BT}}$.
(d) Add $(`p', \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ to tr.
(e) Return $\boldsymbol{s}_{\text{in}}$.

4. For every query $\boldsymbol{s}_{\text{in}} = (\boldsymbol{s}_{\text{R,in}}, \boldsymbol{s}_{\text{C,in}}) \in \Sigma^{r+c}$ to the oracle p:
   (a) If $\exists\, \boldsymbol{s}_{\text{out}} \in \Sigma^{r+c}$ such that $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \mathsf{Cache}_p$
      i. Pop the first such element from Cache_p,
      ii. Add $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ to $\mathsf{tr}_p^{\mathsf{BT}}$,
   (b) Else, if $\exists\, \boldsymbol{s}_{\text{out}} \in \Sigma^{r+c}$ such that $((\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \mathsf{tr}_p^{\mathsf{BT}}$, let the first such match be denoted $\boldsymbol{s}_{\text{out}}$.
   (c) Else, run $\mathsf{BackTrack}(\mathsf{tr}_h^{\mathsf{BT}}, \mathsf{tr}_p^{\mathsf{BT}}, \boldsymbol{s}_{\text{in}})$, and proceed as below according to its output.
      i. If $\mathsf{BackTrack}$'s output is err then abort.
      ii. If $\mathsf{BackTrack}$'s output is none:
         A. Sample a sponge state $\boldsymbol{s}_{\text{out}} \leftarrow \mathcal{U}(\Sigma^{r+c})$.
         B. Add $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ to $\mathsf{tr}_p^{\mathsf{BT}}$.
      iii. If $\mathsf{BackTrack}$'s output is a tuple $(i, \mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}})$, then:
      iv. Let $\widehat{\boldsymbol{\rho}}_i := g_i(\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}})$ and $\ell_z := r - (\ell_{\mathbf{V}}(i) \bmod r)$.
      v. Sample random capacities $\boldsymbol{s}_{\mathrm{C}}^{(0)}, \ldots, \boldsymbol{s}_{\mathrm{C}}^{(L_{\mathbf{V}}(i))} \leftarrow \mathcal{U}(\Sigma^c)$, and a random remainder $\mathbf{z} \leftarrow \mathcal{U}(\Sigma^{\ell_z})$. Let $(\boldsymbol{s}_{\mathrm{R}}^{(0)}, \ldots, \boldsymbol{s}_{\mathrm{R}}^{(L_{\mathbf{V}}(i))}) = (\widehat{\boldsymbol{\rho}}_i \| \mathbf{z})$.
      vi. Append to Cache_p the following pairs in $\Sigma^{r+c} \times \Sigma^{r+c}$:

$$\left(\boldsymbol{s}_{\text{in}}, (\boldsymbol{s}_{\mathrm{C}}^{(0)}, \boldsymbol{s}_{\mathrm{R}}^{(0)})\right), \ \ldots, \ \left((\boldsymbol{s}_{\mathrm{C}}^{(L_{\mathbf{V}}(i)-1)}, \boldsymbol{s}_{\mathrm{R}}^{(L_{\mathbf{V}}(i)-1)}), (\boldsymbol{s}_{\mathrm{C}}^{(L_{\mathbf{V}}(i))}, \boldsymbol{s}_{\mathrm{R}}^{(L_{\mathbf{V}}(i))})\right) .$$

      vii. Let $\boldsymbol{s}_{\text{out}} := (\boldsymbol{s}_{\mathrm{R}}^{(0)}, \boldsymbol{s}_{\mathrm{C}}^{(0)})$.
   (d) Add $(`p', \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ to tr.
   (e) Return $\boldsymbol{s}_{\text{out}}$.

Auxiliary procedure D2SAlgo. The algorithm $\mathsf{D2SAlgo}^{\boldsymbol{f}}(\mathcal{A})$ is defined as follows:

$\mathsf{D2SAlgo}^{\boldsymbol{f}}(\mathcal{A})$:

1. For every $i \in [\mathsf{k}]$, initialize an empty list tr_i.
2. Run $\mathcal{A}^{\mathsf{D2SQuery}}$, answering its queries as follows.
3. For every query $(\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i)$ to oracle g_i (with $i \in [\mathsf{k}]$, $\mathbb{x} \in \{0,1\}^{\leq n}$, $\boldsymbol{\tau} \in \Sigma^{\delta}$, and $\widehat{\boldsymbol{\alpha}}_\iota \in \Sigma^{\ell_{\mathbf{P}}(\iota)}$):
   (a) If the query is already present in tr_i, respond with the corresponding answer. Otherwise, proceed as follows.
   (b) Set $\alpha_\iota := \varphi_\iota^{-1}(\widehat{\boldsymbol{\alpha}}_\iota)$ for $\iota \in [i]$ (compute the unique preimage of α_ι under φ_ι).
   (c) Let $\check{\boldsymbol{\tau}} := \mathsf{bin}(\boldsymbol{\tau}) \in \{0,1\}^{\delta_\star}$ be the binary representation of $\boldsymbol{\tau} \in \Sigma^{\delta}$, where $\delta_\star := \log|\Sigma| \cdot \delta$.
   (d) Set $\rho_i := f_i(\mathbb{x}, \check{\boldsymbol{\tau}}, \alpha_1, \ldots, \alpha_i)$.
   (e) Sample $\widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i)$ (sample a preimage of ρ_i under ψ_i).
   (f) Add the query-answer pair $((\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i), \widehat{\boldsymbol{\rho}}_i)$ to tr_i.
   (g) Respond to $\mathcal{A}$ with $\widehat{\boldsymbol{\rho}}_i$.
4. Output $\mathcal{A}$'s output.

To simplify notation in the analysis, we use the following shorthand when referring to D2SAlgo:

$$\mathsf{D2SAlgo}^{\boldsymbol{f}}(\mathcal{A}) := \mathcal{A}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}} . \tag{10}$$

Query complexity. Intuitively, $\mathsf{D2SAlgo}$ groups random oracle queries to p of length r into a single query then forwarded to $\boldsymbol{f}$. Therefore, assuming no collisions are found during the adversary's execution, to trigger a random oracle query to f_i, $\mathcal{A}$ must perform at least $L_{\mathbf{P}}(i)$ queries in order to trigger a query to f_i. Given a $(t_h, t_p, t_{p^{-1}})$-query adversary $\mathcal{A}$ making at least one query to f_i ($i \in [\mathsf{k}]$), the procedure $\mathsf{D2SAlgo}$ makes at most:

$$\theta_\star(t_h, t_p, t_{p^{-1}}) := \left\lceil \frac{t_p + t_{p^{-1}}}{\min_{i\in[\mathsf{k}]} L_{\mathbf{P}}(i)} \right\rceil = \left\lceil \frac{t_p + t_{p^{-1}}}{\min_{i\in[\mathsf{k}]} \left\lceil \frac{\ell_{\mathbf{P}}(i)}{r} \right\rceil} \right\rceil$$

queries to the oracles $\boldsymbol{f}$.

Time complexity. $\mathsf{D2SAlgo}$ has an additive cost on the runtime of $\mathcal{A}$.

- For each query to h, a lookup over $\mathsf{tr}_h^{\mathsf{BT}}$ is made and $n + \log|\Sigma|$ bits are copied.
- For each query to p, $\mathsf{BackTrack}$ is called, and its output is re-mapped via φ_i^{-1}, ψ_i^{-1}, for each $i \in [\mathsf{k}]$.

Assuming $t_p \geq \max_{i\in[\mathsf{k}]} L_{\mathbf{P}}(i)$ (for notational simplicity), $\mathsf{D2SAlgo}$ runs in time

$$O(t_h \cdot \log t_h \cdot (n + c\cdot \log|\Sigma|) + (t_p + t_{p^{-1}})^2 \cdot (\log(t_p + t_{p^{-1}}) + \log t_h)\cdot c \cdot \log|\Sigma| \cdot (r + c + n) + t_{\boldsymbol{\psi}^{-1}} + t_{\boldsymbol{\varphi}^{-1}}),$$

where

- $t_{\boldsymbol{\varphi}^{-1}}$ is a bound on the time to compute each φ_i^{-1}, and
- $t_{\boldsymbol{\psi}^{-1}}$ is a bound on the time to compute each ψ_i^{-1}.

5.4 Trace transformation

Similarly to Section 5.3, before defining $\mathsf{D2STrace}$ we introduce $\mathsf{StdTrace}$, re-mapping query-answer traces into input-output traces from $\mathcal{Z}_i(\delta, n)$ for $i \in [\mathsf{k}]$ (cf. Definition 3.10), and then define $\mathsf{D2STrace}$ as a thin wrapper around it dealing with encoding and decoding of prover messages.

Auxiliary procedure StdTrace. The procedure $\mathsf{StdTrace}$ works as follows:

$\mathsf{StdTrace}(\mathsf{tr})$:

1. Initialize an empty list $\mathsf{tr}_{\mathsf{std}}$ and lists $\mathsf{tr}_h, \mathsf{tr}_p^{\mathsf{BT}}, \mathsf{tr}_p^{\mathsf{LA}}$.
2. Parse the query-answer trace tr and partition its elements in h-traces tr_h and p-traces $\mathsf{tr}_p^{\mathsf{BT}}, \mathsf{tr}_p^{\mathsf{LA}}$.
   The lists $\mathsf{tr}_h, \mathsf{tr}_p^{\mathsf{BT}}$ are sorted lexicographically by output capacity segment, then output rate capacity segment, then input segment. The list $\mathsf{tr}_p^{\mathsf{LA}}$ are sorted lexicographically by input capacity segment, then input rate capacity segment, then output segment. The two lists will contain the same elements and are kept in sync for element addition and deletion. For clarity, we will talk about tr_p whenever we refer to $\mathsf{tr}_p^{\mathsf{BT}}, \mathsf{tr}_p^{\mathsf{LA}}$ for membership, addition, or deletion.
3. For each query-answer in tr indicating a query to p of the form $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \Sigma^{r+c}$:
   (a) Run $\mathsf{BackTrack}(\mathsf{tr}_h, \mathsf{tr}_p^{\mathsf{BT}}, \boldsymbol{s}_{\text{in}})$ and $\mathsf{LookAhead}(\mathsf{tr}_p^{\mathsf{LA}}, \boldsymbol{s}_{\text{in}}, i)$:
      i. If any of them outputs err, then abort;
      ii. If any of them outputs none, then continue to the next element in tr_p;
      iii. Else, denote the output of $\mathsf{BackTrack}$ as $(i, \mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}})$ and the output of $\mathsf{LookAhead}$ as $\widehat{\boldsymbol{\rho}}_i$.
   (b) Add $(i, (\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}), \widehat{\boldsymbol{\rho}}_i)$ to $\mathsf{tr}_{\mathsf{std}}$.
4. Return $\mathsf{tr}_{\mathsf{std}}$.

Auxiliary procedure D2STrace. The algorithm $\mathsf{D2STrace}$ is defined as follows:

$\mathsf{D2STrace}(\mathsf{tr})$:
1. Initialize an empty list $\mathsf{tr}_{\mathsf{std}}$.
2. Set $\widehat{\mathsf{tr}_{\mathsf{std}}} := \mathsf{StdTrace}(\mathsf{tr})$.
3. For each $(i, (\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}), \widehat{\boldsymbol{\rho}}_i)$ in $\widehat{\mathsf{tr}_{\mathsf{std}}}$:
   - Set $\alpha_1 := {\varphi_1}^{-1}(\widehat{\boldsymbol{\alpha}}_1), \ldots, \alpha_i := {\varphi_i}^{-1}(\widehat{\boldsymbol{\alpha}}_i)$, that is, compute the (unique) preimage of $\widehat{\boldsymbol{\alpha}}$ under $\boldsymbol{\varphi}$.
   - Set $\rho_i := \psi_i(\widehat{\boldsymbol{\rho}}_i)$, that is, map the output to the verifier message space via ψ_i.
   - Let $\check{\boldsymbol{\tau}} := \mathsf{bin}(\boldsymbol{\tau})$.
   - Append $(i, (\mathbb{x}, \check{\boldsymbol{\tau}}, \boldsymbol{\alpha}), \rho_i)$ to $\mathsf{tr}_{\mathsf{std}}$.
4. Return $\mathsf{tr}_{\mathsf{std}}$.

With a slight abuse of notation, we indicate the above procedure with

$$\mathsf{D2STrace} := (\boldsymbol{\varphi}^{-1}, \boldsymbol{\psi}) \circ \mathsf{StdTrace}\,.$$

Time complexity. The time of $\mathsf{D2STrace}$ is dominated by the time of $\mathsf{StdTrace}$, which internally runs $\mathsf{BackTrack}$ and $\mathsf{LookAhead}$ for every query-answer pair made by $\mathcal{A}$ to p (Item 3). Queries to h are also sorted and copied into $\mathsf{tr}_h^{\mathsf{BT}}, \mathsf{tr}_h^{\mathsf{LA}}$ (see Item 2). All prover messages recovered are mapped to the relative prover and verifier message. For notational simplicity, we assume that $L_{\mathbf{V}}(i) \le t_p$ for every $i \in [\mathsf{k}]$. Then, the time of $\mathsf{D2STrace}$ is:

$$O(t_h \cdot n \cdot \log t_h + (t_p + t_{p^{-1}})^2 \cdot (\log(t_p + t_{p^{-1}}) + \log t_h) \cdot c \cdot ((r+c) \cdot \log|\Sigma| + n) + t_p \cdot (t_{\boldsymbol{\varphi}^{-1}} + t_{\boldsymbol{\psi}}))\,,$$

where $t_{\boldsymbol{\varphi}^{-1}}$ is a bound on the time to compute φ_i^{-1} and $t_{\boldsymbol{\psi}}$ is a bound on the time to compute ψ_i, for $i \in [\mathsf{k}]$.

5.5 Analysis of aborts

To help bound the abort probability of the auxiliary procedures $\mathsf{D2SAlgo}$ and $\mathsf{D2STrace}$, we consider a sequence of "bad events". Let tr be the query-answer trace of an algorithm $\mathcal{A}$ with oracle access to (h, p, p^{-1}). An entry in tr is denoted as $(`h', \mathbb{x}, \boldsymbol{s}_{\mathrm{C}})$ if the query was made to $h$ with input $\mathbb{x} \in \{0,1\}^{\le n}$ and output $\boldsymbol{s}_{\mathrm{C}} \in \Sigma^c$, or as $(`p', \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ if the query was made to p with input $\boldsymbol{s}_{\text{in}} \in \Sigma^{r+c}$ and output $\boldsymbol{s}_{\text{out}} \in \Sigma^{r+c}$, or as $(`p^{-1}', \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ if the query was made to p^{-1} with input $\boldsymbol{s}_{\text{out}} \in \Sigma^{r+c}$ and output $\boldsymbol{s}_{\text{in}} \in \Sigma^{r+c}$.

Tree of permutation calls. It is central for the analysis (and the sub-procedure $\mathsf{BackTrack}$) to consider, for a given permutation state $(\boldsymbol{s}_{\mathrm{R}}, \boldsymbol{s}_{\mathrm{C}}) \in \Sigma^{r+c}$, the *sequence of permutation calls* that led to $\boldsymbol{s}_{\mathrm{C}}$. Let $S = \left\{\sigma^{(k)}\right\}_k$ is the set of sequences indexed in k

$$\sigma^{(k)} := (\mathbb{x}^{(k)}, \boldsymbol{s}_{\text{in},1}^{(k)}, \boldsymbol{s}_{\text{out},1}^{(k)}, \boldsymbol{s}_{\text{in},2}^{(k)}, \boldsymbol{s}_{\text{out},2}^{(k)}, \ldots, \boldsymbol{s}_{\text{in},m_k}^{(k)}, \boldsymbol{s}_{\text{out},m_k}^{(k)})\,, \tag{11}$$

such that, for every k:

- $(`h', \mathbb{x}^{(k)}, \boldsymbol{s}_{\mathrm{C},\text{in},1}^{(k)}) \in \mathsf{tr}$, that is, the first query is to h with input $\mathbb{x}$ outputs the capacity segment of $\boldsymbol{s}_{\text{in},1}^{(k)}$.
- $(`p', \boldsymbol{s}_{\text{in},j}^{(k)}, \boldsymbol{s}_{\text{out},j}^{(k)}) \in \mathsf{tr}$ or $(`p^{-1}', \boldsymbol{s}_{\text{out},j}^{(k)}, \boldsymbol{s}_{\text{in},j}^{(k)}) \in \mathsf{tr}$ for $j \in [k]$, that is, the input-output states are consistent with the permutation.
- $\boldsymbol{s}_{\text{out},m_k}^{(k)} = \boldsymbol{s}$, that is, the last permutation state is the one given as input.

Let $T_S := \left\{\Delta^{(k)}\right\}$ be the set of indices in the query-answer trace associated to the set S above, that is

$$\Delta^{(k)} := (j_0^{(k)}, j_2^{(k)}, \ldots, j_{m_k}^{(k)}) \tag{12}$$

where:

- $j_0^{(k)}$ is the index of the first query to h with input $\mathbb{x}^{(k)}$, that is:

$$\mathsf{tr}_{j_0} = (`h', \mathbb{x}^{(k)}, \boldsymbol{s}_{\mathrm{C,in},1}^{(k)}) \quad \text{and } \forall j' < j_0 \colon \mathsf{tr}_{j'} \neq (`h', \mathbb{x}^{(k)}, \boldsymbol{s}_{\mathrm{C,in},1}^{(k)}) \,.$$

- $j_\iota^{(k)}$ is the index of the first query to p with input $\boldsymbol{s}_{\mathrm{in},\iota}^{(k)}$ and output $\boldsymbol{s}_{\mathrm{out},\iota}^{(k)}$ (or to p^{-1} with input $\boldsymbol{s}_{\mathrm{out},\iota}^{(k)}$ and output $\boldsymbol{s}_{\mathrm{in},\iota}^{(k)}$), for $\iota \in [m]$. That is,

$$\begin{aligned} &\mathsf{tr}_{j_\iota^{(k)}} \in \left\{(`p', \boldsymbol{s}_{\mathrm{in},\iota}^{(k)}, \boldsymbol{s}_{\mathrm{out},\iota}^{(k)}), (`p^{-1}', \boldsymbol{s}_{\mathrm{out},\iota}^{(k)}, \boldsymbol{s}_{\mathrm{in},\iota}^{(k)})\right\} \quad \text{and} \\ &\forall j' < j_\iota^{(k)} \colon \mathsf{tr}_{j'} \notin \left\{(`p', \boldsymbol{s}_{\mathrm{in},\iota}^{(k)}, \boldsymbol{s}_{\mathrm{out},\iota}^{(k)}), (`p^{-1}', \boldsymbol{s}_{\mathrm{out},\iota}^{(k)}, \boldsymbol{s}_{\mathrm{in},\iota}^{(k)})\right\}. \end{aligned}$$

Bad events. Given query-answer trace tr and permutation state $\boldsymbol{s}$, two events make it hard to invoke D2SAlgo and D2STrace consistently.

- **Forking events**, where backtracking from $\boldsymbol{s}$ using tr leads to two different sequences of query-answers chains to h and p (cf. Eq. 8). If S has more than one element, we have a fork.
- **Inconsistent answers**, where backtracking will provide inconsistent answers across two different queries. Queries are inconsistent if $\Delta^{(k)}$ is not monotonically increasing, or if there are queries $p^{-1} \circ p$ or $p \circ p^{-1}$ is not the identity.

We capture the above with the following predicates, whose conjunction is denoted $E(\mathsf{tr}, \boldsymbol{s})$:

1. $E_{\mathrm{col}}(\mathsf{tr})$. Checks if a collision happened, for any of the random oracles:

$$\exists\, (`h', \mathbb{x}, \boldsymbol{s}_{\mathrm{C}}), (`h', \mathbb{x}', \boldsymbol{s}'_{\mathrm{C}}) \in \mathsf{tr} \colon \boldsymbol{s}_{\mathrm{C}} = \boldsymbol{s}'_{\mathrm{C}} \wedge \mathbb{x} \neq \mathbb{x}', \tag{13}$$
$$\text{or } \exists\, (`p', \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}), (`p', \boldsymbol{s}'_{\mathrm{in}}, \boldsymbol{s}'_{\mathrm{out}}) \in \mathsf{tr} \colon \boldsymbol{s}_{\mathrm{C,out}} = \boldsymbol{s}'_{\mathrm{C,out}} \wedge \boldsymbol{s}_{\mathrm{in}} \neq \boldsymbol{s}'_{\mathrm{in}}, \tag{14}$$
$$\text{or } \exists\, (`p^{-1}', \boldsymbol{s}_{\mathrm{out}}, \boldsymbol{s}_{\mathrm{in}}), (`p^{-1}', \boldsymbol{s}'_{\mathrm{out}}, \boldsymbol{s}'_{\mathrm{in}}) \in \mathsf{tr} \colon \boldsymbol{s}_{\mathrm{C,out}} = \boldsymbol{s}'_{\mathrm{C,out}} \wedge \boldsymbol{s}_{\mathrm{in}} \neq \boldsymbol{s}'_{\mathrm{in}} \,. \tag{15}$$

   (While collisions over h and p lead to a fork event, collisions over p^{-1} do not. Instead, collisions over p^{-1} will inconsistencies since p will not behave like a permutation function anymore.)

2. $E_{\mathrm{inv}}(\mathsf{tr}, \boldsymbol{s})$ Checks if there exists a sequence of permutation states $\sigma^{(k)} \in S$ that was built using p^{-1}. That is, $\exists\, \Delta^{(k)} \in T_S$ such that

$$\exists\, \Delta^{(k)} = (j_0^{(k)}, j_1^{(k)}, \ldots, j_{m_k}^{(k)}) \in T_S \colon \mathsf{tr}_{j_\iota} = (`p^{-1}', \boldsymbol{s}_{\mathrm{out}}, \boldsymbol{s}_{\mathrm{in}}) \,. \tag{16}$$

3. $E_{\mathrm{time}}(\mathsf{tr}, \boldsymbol{s})$ if the indices associated to $\sigma^{(k)} \in S$ are not ordered. Checks if $\exists\, \Delta^{(k)} \in T_S$ such that $j_\iota^{(k)} > j_{\iota+1}^{(k)}$.

4. $E_{\text{start}}(\mathsf{tr})$. Checks if there is a collision between the output of h and an output of p, that is:

$$\exists\, j, j' \colon\ \ \mathsf{tr}_j = (\text{`}h\text{'}, \mathbb{x}, \boldsymbol{s}_{\text{C,out}}) \ \wedge\ \mathsf{tr}_{j'} = (\text{`}p\text{'}, \boldsymbol{s}'_{\text{in}}, \boldsymbol{s}'_{\text{out}}) \ \wedge\ \boldsymbol{s}'_{\text{C,in}} = \boldsymbol{s}_{\text{C,out}} \tag{17}$$

We now present a sequence of claims that bound the probability of the above events. Let $h\colon \{0,1\}^{\leq n} \to \Sigma^c$ be a random oracle and $p\colon \Sigma^{r+c} \to \Sigma^{r+c}$ be an ideal permutation with p^{-1} its inverse oracle. As defined in Section 3, for $(t_h, t_p, t_{p^{-1}})$-query algorithm $\tilde{\mathcal{P}}$, the expression $y \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}}$ denote the execution of $\tilde{\mathcal{P}}$ with random oracles h, p, whose output is denoted y. In sums, our argument is as follows: if we exclude the presence of p^{-1} queries in S (that is, Eq. 16 does not hold) then:

- If a **forking event** happened, it means that two capacity segments collided. If the lengths of the two reconstructed sequences mismatch, then E_{start} must have happened (that is, Eq. 17 holds). Otherwise, a collision can be found in the query-answer trace (that is, one of Eqs. 13 to 15 happened).
- If **inconsistent answers** are found, then the adversary must have guessed an output capacity segment before it was sampled by the oracle. This is covered by E_{time}.

We upper bound the probability that $E(\mathsf{tr}) = 1$ via a counting argument. We index the queries to h and p by $j \in [t_h + t_p + t_{p^{-1}}]$. Let $\mathsf{tr}_p^{<j}$ be the first $j-1$ entries of the form $(\text{`}p\text{'}, \cdot, \cdot)$ in $\mathsf{tr}$, $\mathsf{tr}_{p^{-1}}^{<j}$ be the first $j-1$ entries of the form $(\text{`}p^{-1}\text{'}, \cdot, \cdot)$ in tr, and, similarly, $\mathsf{tr}_h^{<j}$ be the list of $j-1$ entries of the form $(\text{`}h\text{'}, \cdot, \cdot)$ in tr.

Lemma 5.2. *For every $(t_h, t_p, t_{p^{-1}})$-query algorithm $\tilde{\mathcal{P}}$,*

$$\Pr\left[\exists\, \boldsymbol{s}\colon E(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s}) \;\middle|\; \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right]$$
$$\leq \frac{t_h(t_h + 3t_p + 2L_{\mathbf{P}} + 1) + (t_p + L_{\mathbf{P}})(2t_p + 2L_{\mathbf{P}} - 1) + t_{p^{-1}}(2t_h + 2t_p + t_{p^{-1}} - 1)}{2\,|\Sigma|^c}\,.$$

Proof. The proof follows from Claim 5.3, Claim 5.5, Claim 5.4, Claim 5.6 (used in the penultimate inequality):

$$\Pr\left[\exists\, \boldsymbol{s}\colon E(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s}) \;\middle|\; \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right]$$
$$\leq \Pr\left[E_{\text{col}}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \;\middle|\; \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right]$$
$$+ \Pr\left[\exists\, \boldsymbol{s}\colon\ \ E_{\text{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s}) \;\middle|\; \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right]$$
$$+ \Pr\left[\exists\, \boldsymbol{s}\colon\ \ \begin{array}{c} E_{\text{start}}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s}) \\ \wedge\, \overline{E_{\text{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s})} \end{array} \;\middle|\; \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right]$$

$$+\Pr\left[\exists \boldsymbol{s}: \begin{array}{c} E_{\mathrm{time}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s}) \\ \wedge\, \overline{E_{\mathrm{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})} \\ \wedge\, \overline{E_{\mathrm{start}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})} \end{array} \middle| \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ (\mathbb{x},\pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x},\pi) \end{array}\right]$$

$$\leq \frac{t_h\cdot(t_h+1)+(t_p+L_{\mathbf{P}})\cdot(t_p+L_{\mathbf{P}}-1)+t_{p^{-1}}\cdot(t_{p^{-1}}-1)}{2\cdot|\Sigma^c|}+\frac{t_{p^{-1}}\cdot(t_h+t_p)}{|\Sigma^c|}+\frac{(t_p+L_{\mathbf{P}})\cdot t_h}{|\Sigma^c|}$$

$$+\frac{(t_p+L_{\mathbf{P}})\cdot(t_p+L_{\mathbf{P}}-1)}{2\cdot|\Sigma^c|}+\frac{t_p t_h}{|\Sigma^c|}$$

$$\leq \frac{t_h(t_h+3t_p+2L_{\mathbf{P}}+1)+(t_p+L_{\mathbf{P}})(2t_p+2L_{\mathbf{P}}-1)+t_{p^{-1}}(2t_h+2t_p+t_{p^{-1}}-1)}{2\,|\Sigma|^c}\,.$$

□

Claim 5.3. *For every $(t_h,t_p,t_{p^{-1}})$-query algorithm $\tilde{\mathcal{P}}$,*

$$\Pr\left[E_{\mathrm{col}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}}) \middle| \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ (\mathbb{x},\pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x},\pi) \end{array}\right] \leq \frac{t_h\cdot(t_h+1)+(t_p+L_{\mathbf{P}})\cdot(t_p+L_{\mathbf{P}}-1)+t_{p^{-1}}\cdot(t_{p^{-1}}-1)}{2\cdot|\Sigma^c|}\,.$$

Proof. We partition E_{col} into three events $E_{\mathrm{col},h}, E_{\mathrm{col},p}, E_{\mathrm{col},p^{-1}}$ that correspond to collisions in the output of h (Eq. 13), p (Eq. 14), and p^{-1} (Eq. 15). We write:

$$\Pr\left[E_{\mathrm{col}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}}) \middle| \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ (\mathbb{x},\pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x},\pi) \end{array}\right]$$

$$=\Pr\left[E_{\mathrm{col},h}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}}) \middle| \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ (\mathbb{x},\pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x},\pi) \end{array}\right]$$

$$+\Pr\left[E_{\mathrm{col},p}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}}) \middle| \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ (\mathbb{x},\pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x},\pi) \end{array}\right]$$

$$+\Pr\left[E_{\mathrm{col},p^{-1}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}}) \middle| \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ (\mathbb{x},\pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x},\pi) \end{array}\right]$$

$$=\sum_{j=1}^{t_h+1}\frac{|\mathsf{tr}_h^{<j}|}{|\Sigma^c|}+\sum_{j=1}^{t_p+L_{\mathbf{P}}}\frac{|\mathsf{tr}_p^{<j}|}{|\Sigma^c|}+\sum_{j=1}^{t_{p^{-1}}}\frac{|\mathsf{tr}_{p^{-1}}^{<j}|}{|\Sigma|^{r+c}}$$

$$\leq\sum_{j=1}^{t_h+1}\frac{j-1}{|\Sigma^c|}+\sum_{j=1}^{t_p+L_{\mathbf{P}}}\frac{j-1}{|\Sigma^c|}+\sum_{j=1}^{t_{p^{-1}}}\frac{j-1}{|\Sigma|^{r+c}}$$

$$=\frac{t_h\cdot(t_h+1)+(t_p+L_{\mathbf{P}})\cdot(t_p+L_{\mathbf{P}}-1)+t_{p^{-1}}\cdot(t_{p^{-1}}-1)}{2\cdot|\Sigma^c|}\,.$$

□

Claim 5.4. *For every $(t_h, t_p, t_{p^{-1}})$-query algorithm $\tilde{\mathcal{P}}$,*

$$\Pr\left[\exists\, \boldsymbol{s}:\quad E_{\mathrm{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s}) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right] \leq \frac{t_{p^{-1}} \cdot (t_h + t_p)}{|\Sigma^c|} .$$

Proof. Let ι^* be the first index (if it exists) for which a sequence of permutation states $\sigma^{(k)} \in S$ has associated timestamps T_S contains $\Delta^{(k)} = (j_0, j_1, \ldots, j_{m_k})$ such that $\mathsf{tr}_{j_{\iota^*}} = (\text{`}p^{-1}\text{'}, \boldsymbol{s}_{\mathrm{out}}, \boldsymbol{s}_{\mathrm{in}})$. We partition the event in two cases: the inversion happens after a query to h (i.e., $\iota^* = 1$ in Eq. 16), and the inversion happens after a query to p (i.e., $\iota^* > 1$ in Eq. 16). Therefore:

$$\Pr\left[\exists\, \boldsymbol{s}:\quad E_{\mathrm{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s}) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right]$$
$$\leq \Pr\left[\begin{array}{l} \exists\, \Delta^{(k)} \in T_S \\ \exists\, j, j' \in \Delta^{(k)}: \end{array} : \begin{array}{l} \mathsf{tr}_j = (\text{`}h\text{'}, \mathbb{x}, \boldsymbol{s}_{\mathrm{C,out}}) \\ \wedge\, \mathsf{tr}_{j'} = (\text{`}p^{-1}\text{'}, \boldsymbol{s}'_{\mathrm{out}}, \boldsymbol{s}'_{\mathrm{in}}) \\ \wedge\, \boldsymbol{s}_{\mathrm{C,out}} = \boldsymbol{s}'_{\mathrm{C,in}} \\ \wedge\, \forall\, j'' < j' : \mathsf{tr}_{j''} \neq (\text{`}p\text{'}, \boldsymbol{s}'_{\mathrm{in}}, \boldsymbol{s}'_{\mathrm{out}}) \end{array} \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right]$$
$$+ \Pr\left[\begin{array}{l} \exists\, \Delta^{(k)} \in T_S \\ \exists\, j, j' \in \Delta^{(k)} \end{array} : \begin{array}{l} \mathsf{tr}_j = (\text{`}p\text{'}, \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}) \\ \wedge\, \mathsf{tr}_{j'} = (\text{`}p^{-1}\text{'}, \boldsymbol{s}'_{\mathrm{out}}, \boldsymbol{s}'_{\mathrm{in}}) \\ \wedge\, \boldsymbol{s}'_{\mathrm{C,out}} = \boldsymbol{s}_{\mathrm{C,out}} \\ \wedge\, \forall\, j'' < j' : \mathsf{tr}_{j''} \neq (\text{`}p\text{'}, \boldsymbol{s}'_{\mathrm{in}}, \boldsymbol{s}'_{\mathrm{out}}) \end{array} \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right]$$
$$\leq \sum_{j=1}^{t_{p^{-1}}} \frac{|\mathsf{tr}_p^{<j}|}{|\Sigma^c|} + \sum_{j=1}^{t_{p^{-1}}} \frac{|\mathsf{tr}_h^{<j}|}{|\Sigma^c|} \leq \sum_{j=1}^{t_{p^{-1}}} \frac{t_p}{|\Sigma^c|} + \sum_{j=1}^{t_{p^{-1}}} \frac{t_h}{|\Sigma^c|} = \frac{t_{p^{-1}} \cdot (t_h + t_p)}{|\Sigma^c|} .$$

□

Claim 5.5. *For every $(t_h, t_p, t_{p^{-1}})$-query algorithm $\tilde{\mathcal{P}}$,*

$$\Pr\left[\exists\, \boldsymbol{s}: \begin{array}{c} E_{\mathrm{start}}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s}) \\ \wedge\, \overline{E_{\mathrm{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s})} \end{array} \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right] \leq \frac{(t_p + L_{\mathbf{P}}) \cdot t_h}{|\Sigma^c|} .$$

Proof. We have:

$$\Pr\left[\exists\, \boldsymbol{s}: \begin{array}{c} E_{\mathrm{start}}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s}) \\ \wedge\, \overline{E_{\mathrm{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}, \boldsymbol{s})} \end{array} \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right]$$
$$= \Pr\left[\exists\, j, j': \begin{array}{l} \mathsf{tr}_j = (\text{`}h\text{'}, \mathbb{x}, \boldsymbol{s}_{\mathrm{C,out}}) \\ \wedge\, \mathsf{tr}_{j'} = (\text{`}p\text{'}, \boldsymbol{s}'_{\mathrm{in}}, \boldsymbol{s}'_{\mathrm{out}}) \\ \wedge\, \boldsymbol{s}'_{\mathrm{C,in}} = \boldsymbol{s}_{\mathrm{C,out}} \end{array} \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array}\right]$$

$$= \sum_{j=1}^{t_p+L_{\mathbf{P}}} \frac{t_h}{|\Sigma^c|} = \frac{(t_p+L_{\mathbf{P}})\cdot t_h}{|\Sigma^c|}\,.$$

□

Claim 5.6. *For every $(t_h, t_p, t_{p^{-1}})$-query algorithm $\tilde{\mathcal{P}}$,*

$$\Pr\left[\exists\, \boldsymbol{s}:\ \begin{array}{r}E_{\mathrm{time}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})\\ \wedge\ \overline{E_{\mathrm{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})}\\ \wedge\ \overline{E_{\mathrm{start}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})}\end{array} \ \middle|\ \begin{array}{l}(h,p,p^{-1})\leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\pi)\xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}}\tilde{\mathcal{P}}^{h,p,p^{-1}}\\ b\xleftarrow{\mathsf{tr}_{\mathcal{V}}}\mathcal{V}^{h,p}(\mathbb{x},\pi)\end{array}\right] \le \frac{(t_p+L_{\mathbf{P}})\cdot(t_p+L_{\mathbf{P}}-1)}{2\cdot|\Sigma^c|} + \frac{t_p t_h}{|\Sigma^c|}\,.$$

Proof. Since $\overline{E_{\mathrm{start}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})}$ and $\overline{E_{\mathrm{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}},\boldsymbol{s})}$, we have two cases:

- The case where $\exists\,\Delta^{(k)}\in T_S$ with indices $j_\iota > j_{\iota+1}$ and $\mathsf{tr}_{j_\iota}, \mathsf{tr}_{j_{\iota+1}}$ are queries to h and p, respectively.
- The case where $\exists\,\Delta^{(k)}\in T_S$ with query indices $j_\iota > j_{\iota+1}$ and $\mathsf{tr}_{j_\iota}, \mathsf{tr}_{j_{\iota+1}}$ are both queries to p.

Therefore:

$$\Pr\left[\exists\, \boldsymbol{s}:\ \begin{array}{r}E_{\mathrm{time},p}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})\\ \wedge\ \overline{E_{\mathrm{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})}\\ \wedge\ \overline{E_{\mathrm{start}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})}\end{array} \ \middle|\ \begin{array}{l}(h,p,p^{-1})\leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\pi)\xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}}\tilde{\mathcal{P}}^{h,p,p^{-1}}\\ b\xleftarrow{\mathsf{tr}_{\mathcal{V}}}\mathcal{V}^{h,p}(\mathbb{x},\pi)\end{array}\right]$$

$$\le \Pr\left[\exists\, \boldsymbol{s}:\ \begin{array}{r}E_{\mathrm{time},p}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})\\ \wedge\ \overline{E_{\mathrm{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})}\\ \wedge\ \overline{E_{\mathrm{start}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})}\end{array} \ \middle|\ \begin{array}{l}(h,p,p^{-1})\leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\pi)\xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}}\tilde{\mathcal{P}}^{h,p,p^{-1}}\\ b\xleftarrow{\mathsf{tr}_{\mathcal{V}}}\mathcal{V}^{h,p}(\mathbb{x},\pi)\end{array}\right]$$

$$+ \Pr\left[\exists\, \boldsymbol{s}:\ \begin{array}{r}E_{\mathrm{time},h}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})\\ \wedge\ \overline{E_{\mathrm{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})}\\ \wedge\ \overline{E_{\mathrm{start}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}},\boldsymbol{s})}\end{array} \ \middle|\ \begin{array}{l}(h,p,p^{-1})\leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\pi)\xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}}\tilde{\mathcal{P}}^{h,p,p^{-1}}\\ b\xleftarrow{\mathsf{tr}_{\mathcal{V}}}\mathcal{V}^{h,p}(\mathbb{x},\pi)\end{array}\right]$$

$$\le \Pr\left[\exists\, j' < j:\ \begin{array}{l}\mathsf{tr}_j = (\text{`}h\text{'},\mathbb{x},\boldsymbol{s}_{\mathrm{C,out}})\\ \wedge\ \mathsf{tr}_{j'} = (\text{`}p\text{'},\boldsymbol{s}'_{\mathrm{in}},\boldsymbol{s}'_{\mathrm{out}})\\ \wedge\ \boldsymbol{s}^{j'}_{\mathrm{C,in}} = \boldsymbol{s}_{\mathrm{C,out}}\\ \wedge\ \forall\, j'' < j:\ \mathsf{tr}_{j''} \neq (\text{`}h\text{'},\mathbb{x},\boldsymbol{s}'_{\mathrm{C,out}})\end{array} \ \middle|\ \begin{array}{l}(h,p,p^{-1})\leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\pi)\xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}}\tilde{\mathcal{P}}^{h,p,p^{-1}}\\ b\xleftarrow{\mathsf{tr}_{\mathcal{V}}}\mathcal{V}^{h,p}(\mathbb{x},\pi)\end{array}\right]$$

$$+ \Pr\left[\exists\, j' < j:\ \begin{array}{l}\mathsf{tr}_j = (\text{`}p\text{'},\boldsymbol{s}_{\mathrm{in}},\boldsymbol{s}_{\mathrm{out}})\\ \wedge\ \mathsf{tr}_{j'} = (\text{`}p\text{'},\boldsymbol{s}'_{\mathrm{in}},\boldsymbol{s}'_{\mathrm{out}})\\ \wedge\ \boldsymbol{s}^{j'}_{\mathrm{C,in}} = \boldsymbol{s}_{\mathrm{C,out}}\\ \wedge\ \forall\, j'' < j:\ \mathsf{tr}_{j''} \neq (\text{`}p\text{'},\boldsymbol{s}_{\mathrm{in}},\boldsymbol{s}'_{\mathrm{in}})\end{array} \ \middle|\ \begin{array}{l}(h,p,p^{-1})\leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\pi)\xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}}\tilde{\mathcal{P}}^{h,p,p^{-1}}\\ b\xleftarrow{\mathsf{tr}_{\mathcal{V}}}\mathcal{V}^{h,p}(\mathbb{x},\pi)\end{array}\right]$$

$$\le \sum_{j=1}^{t_p+L_{\mathbf{P}}} \frac{|\mathsf{tr}_p^{<j}|}{|\Sigma^c|} + \sum_{j=1}^{t_h} \frac{|\mathsf{tr}_p^{<j}|}{|\Sigma^c|} \le \sum_{j=1}^{t_p+L_{\mathbf{P}}} \frac{j-1}{|\Sigma^c|} + \sum_{j=1}^{t_h} \frac{t_p}{|\Sigma^c|} = \frac{(t_p+L_{\mathbf{P}})\cdot(t_p+L_{\mathbf{P}}-1)}{2\cdot|\Sigma^c|} + \frac{t_p t_h}{|\Sigma^c|}\,.$$

□

5.6 Proof of Lemma 5.1

We finally prove the lemma via a hybrid argument; hybrids are summarized in Figure 4.

$\mathsf{Hyb_0}$. The first hybrid is the left-side experiment in the lemma statement, where the malicious argument prover $\tilde{\mathcal{P}}$ queries the oracles $h\colon \{0,1\}^{\leq n} \to \Sigma^c$ and $p, p^{-1}\colon \Sigma^{r+c} \to \Sigma^{r+c}$ sampled from $\mathcal{D}_{\mathfrak{S}}$ (cf. Definition 4.2).

$\mathsf{Hyb_1}$. In this hybrid we make the following changes:

- in line 1 we sample oracles as

$$\boldsymbol{g} \coloneqq (g_i)_{i\in[\mathsf{k}]} \leftarrow \mathcal{U}\left(\left(\{0,1\}^{\leq n} \times \Sigma^{\delta} \times \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)} \to \Sigma^{\ell_{\mathbf{V}}(i)}\right)_{i\in[\mathsf{k}]}\right) \tag{18}$$

  instead of the oracles $(h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}$ from Hyb_0;
- in line 2 the argument prover is $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}$ instead of $\tilde{\mathcal{P}}^{h,p,p^{-1}}$ from Hyb_0;
- in line 3 the argument verifier is $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}$ instead of $\mathcal{V}^{h,p}(\mathbb{x}, \pi)$ from Hyb_0;
- in line 4 the query-answer trace is set to $(\boldsymbol{\varphi}^{-1}, \boldsymbol{\psi})(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$ instead of $\mathsf{D2STrace}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$ from Hyb_0.

Claim 5.7. *The statistical distance between Hyb_0 and Hyb_1 is at most*

$$\frac{t_h(t_h + 3t_p + 2L_{\mathbf{P}} + 1) + (t_p + L_{\mathbf{P}})(2t_p + 3L_{\mathbf{P}} - 1) + t_{p^{-1}}(2t_h + 2t_p + t_{p^{-1}} - 1)}{2\,|\Sigma|^c} .$$

To prove the above, we proceed with a sequence of smaller intermediate steps.

$\mathsf{Hyb}_{0.1}$ In this hybrid, we modify line 2 adding two conditions:

- We introduce Item 4(c)i of $\mathsf{D2SQuery}$ (for every query of $\tilde{\mathcal{P}}$, we run $\mathsf{BackTrack}$ and abort if it returns err).
- At the end of the execution of $\tilde{\mathcal{P}}$ and the verifier $\mathcal{V}$, we run $\mathsf{StdTrace}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$ and throw away the result.

The hybrid $\mathsf{Hyb}_{0.1}$ is a strengthening of Hyb_0 since we abort the experiment if $\mathsf{BackTrack}$ or $\mathsf{D2STrace}$ returns err. In particular, the game aborts every time $\mathsf{D2STrace}$ aborts (since $\mathsf{D2STrace}$ runs $\mathsf{BackTrack}$ for every permutation state). Therefore

$$\Delta\left(\mathsf{Hyb}_0, \mathsf{Hyb}_{0.1}\right) = \Pr\left[\ \mathsf{StdTrace}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \ \ \textbf{aborts} \ \middle|\ \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array} \right] .$$

The procedure $\mathsf{StdTrace}(\mathsf{tr})$ (Item 2) aborts in one of the following two cases (cf. Item 3(a)i).

1. $\mathsf{BackTrack}$ returns err. This happens only if there exists two different sequences in Chains, which in turn means that two different input segments led to the same output capacity segment. More precisely, $\mathsf{BackTrack}$ returns err if there exists two different sequences in Chains:

$$(\mathbb{x}^{(1)}, \boldsymbol{\tau}^{(1)}, (\boldsymbol{s}^{(1)}_{\mathrm{R,in},0}, \boldsymbol{s}^{(1)}_{\mathrm{R,out},0}, \boldsymbol{s}^{(1)}_{\mathrm{R,in},1}, \boldsymbol{s}^{(1)}_{\mathrm{R,out},1}, \dots))$$
$$(\mathbb{x}^{(2)}, \boldsymbol{\tau}^{(2)}, (\boldsymbol{s}^{(2)}_{\mathrm{R,in},0}, \boldsymbol{s}^{(2)}_{\mathrm{R,out},0}, \boldsymbol{s}^{(2)}_{\mathrm{R,in},1}, \boldsymbol{s}^{(2)}_{\mathrm{R,out},1}, \dots))$$

   from which it is possible to read a valid sequence of prover messages $\widehat{\boldsymbol{\alpha}}^{(1)}, \widehat{\boldsymbol{\alpha}}^{(2)}$. Both chains are tied to the same final capacity state $\boldsymbol{s}_{\mathrm{C}}$ queried to $\mathsf{BackTrack}$. We have the following cases:

- If the sequences have different length, then without loss of generality $h(\mathbb{x}^{(1)})$ is the capacity state of a segment of a permutation state $\boldsymbol{s}^{(2)}_{\text{out},\iota}$ in the second chain. In this case, $E_{\text{inv}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathcal{V}, \boldsymbol{s}^{(2)}_{\text{out},\iota})$ or $E_{\text{start}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathcal{V})$ hold.
- $\mathbb{x}^{(1)} \neq \mathbb{x}^{(2)}$, and yet $h(\mathbb{x}^{(1)}) = h(\mathbb{x}^{(2)})$. This event happens when E_{col} holds.
- There exist $k_1, k_2 > 0$ such that $\boldsymbol{s}^{(1)}_{\text{R,in},k_1} \neq \boldsymbol{s}^{(2)}_{\text{R,in},k_2}$ and yet $\boldsymbol{s}^{(1)}_{\text{C,out},k_1} = \boldsymbol{s}^{(2)}_{\text{C,out},k_2}$, and all previous elements are equal. In other words, there are two different inputs in tr_p for which the output capacity is the same. This event is covered by the collision predicate $E_{\text{col}}(\mathsf{tr})$.
- There exist k_1, k_2 such that $\boldsymbol{s}^{(1)}_{\text{R,out},k_1} \neq \boldsymbol{s}^{(2)}_{\text{R,out},k_2}$ and $\boldsymbol{s}^{(1)}_{\text{C,out},k_1} = \boldsymbol{s}^{(2)}_{\text{C,out},k_2}$, and all previous elements are equal. In other words, two outputs share the same inputs, the same output capacity segment, but have different output rate segments. This means that p answers queries inconsistently, and in this case $E_{\text{col}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}})$ holds.

2. LookAhead returns err. This can happen only if the input-output queries are inconsistent. As above, in this case $E_{\text{col}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}})$ holds.

We conclude that

$$\begin{aligned}
&\Delta\left(\mathsf{Hyb}_0, \mathsf{Hyb}_{0.1}\right) \\
&\leq \Pr\left[\ \exists\, \boldsymbol{s} \colon E(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}}, \boldsymbol{s}) \ \middle|\ \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array} \right] \\
&\leq \frac{t_h(t_h + 3t_p + 2L_{\mathbf{P}} + 1) + (t_p + L_{\mathbf{P}})(2t_p + 2L_{\mathbf{P}} - 1) + t_{p^{-1}}(2t_h + 2t_p + t_{p^{-1}} - 1)}{2\,|\Sigma|^c}\,.
\end{aligned}$$

(by Lemma 5.2)

$\mathsf{Hyb}_{0.2}$ We introduce Items 4a and 4c of D2SQuery to the experiment. Before starting the experiment, we initialize an empty list Cache_p. Throughout the execution of $\tilde{\mathcal{P}}$, for each query $p(\boldsymbol{s}_{\text{in}})$ for some $\boldsymbol{s}_{\text{in}} \in \Sigma^{r+c}$, we first check if $\exists\, \boldsymbol{s}_{\text{out}} \in \Sigma^{r+c}$: $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \mathsf{Cache}_p$, and if so we pop the first match and return it (as in Item 4a). Else, we run BackTrack.

- If, as in the previous hybrid, BackTrack returns err, we abort the experiment.
- If BackTrack returns none, as in Item 4(c)ii, we sample a fresh answer $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$, append it to $\mathsf{tr}_p^{\mathsf{BT}}$, and return it.
- If BackTrack returns a tuple $(i, \mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}})$, for $\iota \in [L_{\mathbf{V}}(i)]$, sample fresh $\boldsymbol{s}^{(\iota)} \leftarrow\!\$\ \Sigma^{r+c}$. Then, add the pairs $((\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}), (\boldsymbol{s}^{(2)}, \boldsymbol{s}^{(3)}), \cdots, (\boldsymbol{s}^{(L_{\mathbf{V}}(i)-1)}, \boldsymbol{s}^{(L_{\mathbf{V}}(i))}))$ to a list Cache_p.

Claim 5.8. *The statistical distance between $\mathsf{Hyb}_{0.1}$ and $\mathsf{Hyb}_{0.2}$ is bounded by $\sum_{i \in [\mathsf{k}]} \left\lceil \frac{\ell_{\mathbf{V}}(i)}{r} \right\rceil \cdot \frac{t_p + t_{p^{-1}}}{|\Sigma^{r+c}|}$.*

Proof. Upon receiving a query $p(\boldsymbol{s}_{\text{in}})$ or $p^{-1}(\boldsymbol{s}_{\text{out}})$, we have two cases.

- If $\boldsymbol{s}_{\text{in}}$ is not a query stored in Cache_p, then either the query was previously made (cf. Item 4b), in which case the query is responded to consistently if $E(\mathsf{tr}) = 0$. If BackTrack is called, the output is either none or a tuple, in which case a uniformly distributed element in Σ^{r+c} is returned.

- Else, let $\boldsymbol{s}_{\text{out}} \in \Sigma^{r+c}$ be such that $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \mathsf{Cache}_p$. If $\boldsymbol{s}_{\text{in}}$ was previously queried to p or $\boldsymbol{s}_{\text{out}}$ was queried to p^{-1}, denote with j the first time that $\boldsymbol{s}_{\text{in}}$ is queried to p receiving $\boldsymbol{s}_{\text{out}}$, and with j' the index of the query that received a different output $\boldsymbol{s}'_{\text{out}} \neq \boldsymbol{s}_{\text{out}}$ as a result of Item 4a. That is, during the j'-th query, $\tilde{\mathcal{P}}$ made a query $p(\boldsymbol{s}_{\text{in}})$ that matched an element in Cache_p, whereas during the j-th query, a match for $\boldsymbol{s}_{\text{in}}$ was not found in Cache_p (cf. Item 4a) nor in the oracle trace (cf. Item 4b). (Same goes for p^{-1}.) In this case, since $E(\mathsf{tr}_{\tilde{\mathcal{P}}}) = 0$, the response is sampled uniformly random after BackTrack's execution. This means that, for some query index $j'' \in [t_h + t_p]$, $j \leq j'' < j'$ exists, which led to adding elements to Cache_p (cf. Item 4(c)vi) and one of those sampled elements was already appearing in tr. Since the entries of Cache_p are pairs sampled uniformly at random from $\Sigma^{r+c} \times \Sigma^{r+c}$, and since Cache_p receives at most $\sum_{i \in [\mathsf{k}]} \left\lceil \frac{\ell_{\mathbf{V}}(i)}{r} \right\rceil$, the probability of this bad event happening is bounded by

$$\sum_{i \in [\mathsf{k}]} \left\lceil \frac{\ell_{\mathbf{V}}(i)}{r} \right\rceil \cdot \frac{t_p + t_{p^{-1}}}{\left|\Sigma^{r+c}\right|} . \qquad \square$$

Hyb$_{0.3}$ We now change the way elements are added to Cache_p: instead of sampling them uniformly at random, if BackTrack responds with a tuple then we forward the queries to the oracles $\boldsymbol{g}$ (cf. Item 4(c)iv). The capacity states and the "remainder" elements (cf. Item 4(c)v) are sampled uniformly at random. This last hybrid is identical to Hyb_1.

Claim 5.9. *$\mathsf{Hyb}_{0.2}$ and $\mathsf{Hyb}_{0.3}$ are perfectly indistinguishable.*

Proof. We first prove that all invocations of $\boldsymbol{g}$ are on different elements, and then argue their distributions.

If a query $p(\boldsymbol{s}_{\text{in}})$, for some $\boldsymbol{s}_{\text{in}} \in \Sigma^{r+c}$ led to Item 4c, then it must have not been present in tr_p and therefore not previously queried. In particular, this means that one of the following holds: the rate segment $\boldsymbol{s}_{\text{R,in}}$ is different from all previous queries, which implies that the output triple of BackTrack is different in the last r elements of $\widehat{\boldsymbol{\alpha}}$ (cf. Item 2b), or the capacity segment $\boldsymbol{s}_{\text{C,in}}$ is different from all previous queries, which, since $E(\mathsf{tr}) = 0$, implies that BackTrack's output must be different.

Since $\boldsymbol{g}$'s queries are always different throughout the execution of $\tilde{\mathcal{P}}$, then g_i's output is independently and uniformly distributed in $\Sigma^{\ell_{\mathbf{V}}(i)}$, for $i \in [\mathsf{k}]$, just like $(\boldsymbol{s}_{\text{R,out},0} \| \boldsymbol{s}_{\text{R,out},1} \| \cdots \| \boldsymbol{s}_{\text{R,out},L_{\mathbf{V}}(i)})[0\colon \ell_{\mathbf{V}}(i)]$. $\square$

Putting the above intermediate changes together:

$$\begin{aligned}
&\Delta\left(\mathsf{Hyb}_0, \mathsf{Hyb}_1\right) \\
&\leq \frac{t_h(t_h + 3t_p + 2L_{\mathbf{P}} + 1) + (t_p + L_{\mathbf{P}})(2t_p + 2L_{\mathbf{P}} - 1) + t_{p^{-1}}(2t_h + 2t_p + t_{p^{-1}} - 1)}{2\left|\Sigma\right|^c} + \frac{t_p \sum_{i \in [\mathsf{k}]} \lceil \ell_{\mathbf{V}}(i)/r \rceil}{\left|\Sigma\right|^{r+c}} \\
&\leq \frac{t_h(t_h + 3t_p + 2L_{\mathbf{P}} + 1) + (t_p + L_{\mathbf{P}})(2t_p + 2L_{\mathbf{P}} - 1) + t_{p^{-1}}(2t_h + 2t_p + t_{p^{-1}} - 1)}{2\left|\Sigma\right|^c} + \frac{t_p \cdot \sum_{i \in [\mathsf{k}]} \lceil \ell_{\mathbf{V}}(i)/r \rceil}{2\left|\Sigma\right|^{c}} \\
&\leq \frac{t_h(t_h + 3t_p + 2L_{\mathbf{P}} + 1) + (t_p + L_{\mathbf{P}})(2t_p + 3L_{\mathbf{P}} - 1) + t_{p^{-1}}(2t_h + 2t_p + t_{p^{-1}} - 1)}{2\left|\Sigma\right|^c} .
\end{aligned}$$

Hyb$_2$. In this hybrid we make the following changes:

- in line 1 we sample oracles as

$$\boldsymbol{e} := (e_i)_{i \in [\mathsf{k}]} \leftarrow \mathcal{U}\left((\{0,1\}^{\leq n} \times \Sigma^{\delta} \times \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)} \to \mathcal{M}_{\mathbf{V},i})_{i \in [\mathsf{k}]}\right) \tag{19}$$

instead of the oracles $\boldsymbol{g}$ from Hyb_1 (see Equation 18);
- in line 2 the argument prover is $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{e}}}$ instead of $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}$ from Hyb_1;
- in line 3 the argument verifier is $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{e}}}$ instead of $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}(\mathbb{x}, \pi)$ from Hyb_1;
- in line 4 the query-answer trace is set to $\boldsymbol{\varphi}^{-1}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$ instead of $(\boldsymbol{\varphi}^{-1}, \boldsymbol{\psi})(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$ from Hyb_1.

Claim 5.10. *The statistical distance between Hyb_1 and Hyb_2 is at most*

$$\sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) .$$

Proof. This is immediate from the definition of $\varepsilon_{\mathsf{cdc}}(\lambda, n)$-biased maps, since the statistical distance between Hyb_1 and Hyb_2 is bounded by the distance between $(\mathcal{U}(\mathcal{M}_{\mathbf{V},i}))_{i \in [\mathsf{k}]}$ and $(\psi_i(\mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)})))_{i \in [\mathsf{k}]}$. □

Hyb_3. In this hybrid we make the following changes:
- in line 1 we sample oracles as

$$\boldsymbol{f} := (f_i)_{i \in [\mathsf{k}]} \leftarrow \mathcal{U}\left((\{0,1\}^{\leq n} \times \{0,1\}^{\delta_\star} \times \mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i} \to \mathcal{M}_{\mathbf{V},i})_{i \in [\mathsf{k}]}\right) \tag{20}$$

instead of the oracles $\boldsymbol{e}$ from Hyb_2 (see Equation 19);
- in line 2 the argument prover is $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}$ instead of $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{e}}}$ from Hyb_2;
- in line 3 the argument verifier is $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}(\mathbb{x}, \pi)$ instead of $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{e}}}(\mathbb{x}, \pi)$ from Hyb_2;
- in line 4 the query-answer trace is set to $\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}$ instead of $\boldsymbol{\varphi}^{-1}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$ from Hyb_2.

Claim 5.11. *Hyb_2 and Hyb_3 are identically distributed.*

Proof. The difference in the two hybrids is the query-answer trace: while the oracle query-answer trace in Hyb_2 is of the form but stores mappings in $\{0,1\}^{\leq n} \times \Sigma^{\delta} \times \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)} \to \mathcal{M}_{\mathbf{V},i}$, in hybrid Hyb_3 it is of the form $\{0,1\}^{\leq n} \times \{0,1\}^{\delta_\star} \times \mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i}$, then composed with φ_i^{-1} to obtain the final output. The map φ_i is into and hence $\varphi_i^{-1} \circ \varphi_i$ is the identity. A similar argument applies to $\mathsf{bin}(\cdot)$, which is used to encode the salt. Therefore, at most one element of $\mathcal{M}_{\mathbf{P},i}$ is associated with the malicious prover query and the two are perfectly indistinguishable. □

Hyb_4. This hybrid corresponds to the right-side experiment in Lemma 5.1:
- in line 1 we sample oracles $\boldsymbol{f} \leftarrow \mathcal{D}_{\mathsf{IP}}(\lambda, n)$, the same as the oracles $\boldsymbol{f}$ in Hyb_3 (see Equation 20);
- in line 2 the argument prover is run as $\mathsf{D2SAlgo}^{\boldsymbol{f}}(\tilde{\mathcal{P}})$, which by definition (see Equation 10), equals the argument prover $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}$ in Hyb_3;
- in line 3 the argument verifier is run as $\mathcal{V}^{\boldsymbol{f}}_{\mathsf{std}}(\mathbb{x}, \pi)$ instead of as $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}(\mathbb{x}, \pi)$ in Hyb_3; and
- in line 4 the query-answer trace is not translated as in Hyb_3.

Claim 5.12. *Hyb_3 and Hyb_4 are identically distributed.*

Proof. We discuss the argument verifier (line 3) as that is the only potential difference: we argue that $\mathcal{V}_{\mathsf{std}}^{\boldsymbol{f}}(\mathbb{x}, \pi)$ in Hyb_4 behaves the same as $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}(\mathbb{x}, \pi)$ in Hyb_3.

We are left to prove that, if D2SAlgo does not abort, then the decision bit output by $\mathcal{V}_{\mathsf{std}}^{\boldsymbol{f}}(\mathbb{x}, \pi)$ on the left, and by $\mathsf{D2SAlgo}^{\mathsf{D2SQuery}^{\boldsymbol{\psi} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}(\mathcal{V})(\mathbb{x}, \pi)$ on the right, are the same. Both verifiers return 1 if and only if $\mathbf{V}(\mathbb{x}, \boldsymbol{\alpha}, \boldsymbol{\rho}) = 1$, so it is sufficient to prove that the vector $\boldsymbol{\rho}$ is the same in both cases.

- In $\mathcal{V}_{\mathsf{std}}^{\boldsymbol{f}}(\mathbb{x}, \pi)$, for each $i \in [\mathsf{k}]$ has $\rho_i := f_i(\mathbb{x}, \mathsf{bin}(\boldsymbol{\tau}), \alpha_1, \dots, \alpha_i)$.
- In $\mathsf{D2SAlgo}^{\mathsf{D2SQuery}^{\boldsymbol{\psi} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}(\mathcal{V})(\mathbb{x}, \pi)$, the verifier algorithm will initialize the state $\mathsf{st}_0 := \mathsf{DS.Start}(\mathbb{x})$. Since the oracle is queried only once, a value $\boldsymbol{s}_{\mathrm{C}}$ is always returned by D2SQuery when queried on $\mathbb{x}$.

  Then, the verifier proceeds running $\mathsf{DS.Absorb}(\mathsf{st}_0, \boldsymbol{\tau} \| \widehat{\boldsymbol{\alpha}_1})$ after encoding the prover message $\widehat{\boldsymbol{\alpha}}_i = \varphi_i(\alpha_i)$. Internally, DS.Absorb procedure will chain calls to the random oracle wrapper D2SQuery which, since it never aborts, will always produce different capacity states. When querying D2SQuery on the last permutation state $\boldsymbol{s}_1$, the procedure BackTrack invoked on $\boldsymbol{s}_1$ will return $(1, \mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1)$. This is shown by inspection, noting that a single chain exists since each capacity state is different and used only once. Then, a query to $f_1(\mathbb{x}, \mathsf{bin}(\boldsymbol{\tau}), \varphi_1^{-1}(\widehat{\boldsymbol{\alpha}}_1))$ is made. Since $\varphi_1^{-1} \circ \varphi_1$ is the identity (φ_1 being injective), $\varphi_1^{-1}(\widehat{\boldsymbol{\alpha}}_1) = \alpha_1$ and the output is exactly ρ_1. The output is then mapped into the permutation's alphabet sampling a preimage via $\psi_1^{-1}(\rho_1)$. D2SQuery is then programmed to return blocks of it when permutation queries are chained starting from $\boldsymbol{s}_1$, which are recovered by the verifier via DS.Squeeze and mapped into the verifier message space via $\psi_1(\widehat{\boldsymbol{\rho}}_1)$. Since $\psi_1 \circ \psi_1^{-1}$ is the identity, the output is exactly ρ_1 and the first verifier message is equal to the one produced by $\mathcal{V}_{\mathsf{std}}^{\boldsymbol{f}}(\mathbb{x}, \pi)$.

  The cases for $i > 1$ ($i \in [\mathsf{k}]$) proceed analogously and thus, as claimed, the two distributions are identical.

□

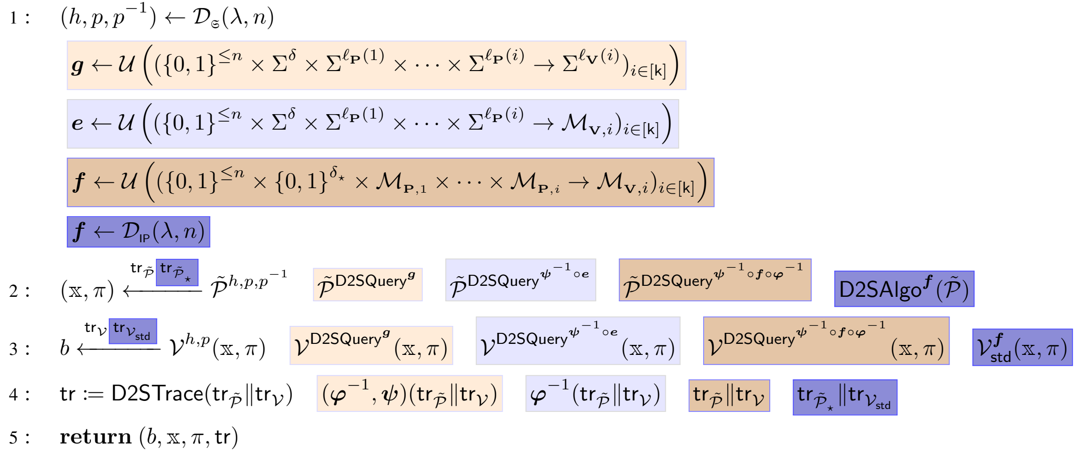

Figure 4: Hybrid experiments Hyb_0, Hyb_1, Hyb_2, Hyb_3, Hyb_4 for Lemma 5.1.

6 Soundness and knowledge soundness

We formally state and prove the theorems about soundness and knowledge soundness for our Fiat–Shamir transformation (in particular, the two theorems below formally restate the two bounds in Theorem 1). Let $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ be a public-coin IP for a relation $\mathcal{R}$. Let Σ be a finite alphabet and cdc be a codec for IP over Σ with bias $\varepsilon_{\mathsf{cdc}}$ (see Definition 4.1). Let $\mathcal{D}_{\mathfrak{S}}$ be an ideal permutation distribution over Σ with capacity $c \in \mathbb{N}$ and rate $r \in \mathbb{N}$ (see Definition 4.2). For a salt size $\delta \in \mathbb{N}$, let $\mathsf{NARG} := \mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ be the non-interactive argument for $\mathcal{R}$ in the $\mathcal{D}_{\mathfrak{S}}$-oracle model constructed in Construction 4.3.

The theorems below rely on the following expressions for a query bound $\theta_\star(t)$, an additive error $\eta_\star(\lambda, t)$, and a privacy parameter $\delta_\star$:

$$
\begin{aligned}
\theta_\star(t) &:= \left\lceil \frac{t}{\min_{i \in [\mathsf{k}]} \left\lceil \frac{\ell_{\mathbf{P}}(i)}{r} \right\rceil} \right\rceil \leq t \,, \\
\eta_\star(\lambda, t) &:= \frac{4t^2 + \max_{i \in [\mathsf{k}]} \left\lceil \frac{\ell_{\mathbf{P}}(i)}{r} \right\rceil \left(t + \max_{i \in [\mathsf{k}]} \frac{\ell_{\mathbf{P}}(i)}{r} \right)}{|\Sigma|^c} + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc}, i}(\lambda, n) \text{ , and} \\
\delta_\star &:= \delta \log |\Sigma| \;.
\end{aligned}
\tag{21}
$$

Theorem 6.1 (soundness)**.** *If* IP *has state-restoration soundness error* $\varepsilon^{\mathrm{sr}}_{\mathsf{IP}}$ *(see Definition 3.13) then* NARG *has soundness error* $\varepsilon_{\mathsf{NARG}}$ *(see Definition 3.4) such that, for every security parameter* $\lambda \in \mathbb{N}$*, query bound* $t \in \mathbb{N}$*, and instance size bound* $n \in \mathbb{N}$*,*

$$
\varepsilon_{\mathsf{NARG}}(\lambda, t, n) \leq \varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t, n)) + \eta_\star(\lambda, t) \,.
$$

Theorem 6.2 (knowledge soundness)**.** *If* IP *has rewinding state-restoration knowledge soundness error* $\kappa^{\mathrm{sr}}_{\mathsf{IP}}$ *with extraction time* $\mathbf{et}_{\mathsf{IP}}$ *(see Definition 3.15) then* NARG *has rewinding knowledge soundness error* $\kappa_{\mathsf{NARG}}\big(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n)\big)$ *with extraction time* $\mathbf{et}_{\mathsf{NARG}}$ *(see Definition 3.7) such that, for every security parameter* $\lambda \in \mathbb{N}$*, query bound* $t \in \mathbb{N}$*, instance size bound* $n \in \mathbb{N}$*,*

- $\kappa_{\mathsf{NARG}}\big(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n)\big) \leq \kappa^{\mathrm{sr}}_{\mathsf{IP}}\left(\delta_\star, \theta_\star(t), n, \delta'_{\tilde{\mathcal{P}}}(\lambda, n)\right) + \eta_\star(\lambda, t)$*, and*
- $\mathbf{et}_{\mathsf{NARG}}(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n), \tau_{\tilde{\mathcal{P}}}(\lambda, n)) \leq \mathbf{et}^{\mathrm{sr}}_{\mathsf{IP}}\left(\delta_\star, \theta_\star(t), n, \delta'_{\tilde{\mathcal{P}}}(\lambda, n), \tau'_{\tilde{\mathcal{P}}}(\lambda, n)\right) + t^2 \cdot \log t \cdot ((r + c^2) \log |\Sigma| + n)$*.*

Above,

$$
\begin{aligned}
\delta'_{\tilde{\mathcal{P}}}(\lambda, n) &:= \delta_{\tilde{\mathcal{P}}}(\lambda, n) + \eta_\star(\lambda, t) \\
\tau'_{\tilde{\mathcal{P}}}(\lambda, n) \,, &:= \tau_{\tilde{\mathcal{P}}}(\lambda, n) + t^2 \cdot \log t \cdot ((r + c^2) \log |\Sigma| + n) \,.
\end{aligned}
$$

Moreover, if the IP state-restoration extractor is straightline (see Definition 3.13) then the NARG extractor is also straightline (see Definition 3.5). In this case:

- *the (straightline) knowledge soundness error is* $\kappa_{\mathsf{NARG}}(\lambda, t, n) \leq \kappa^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t), n) + \eta_\star(\lambda, t)$*; and*
- *the (straightline) extraction time is* $\mathbf{et}_{\mathsf{NARG}}(\lambda, t, n) \leq \mathbf{et}^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t), n) + t^2 \cdot \log t \cdot ((r + c^2) \log |\Sigma| + n)$*.*

The proof of Theorem 6.1 is in Section 6.1, and the proof of Theorem 6.2 is in Section 6.2. Both theorems are proved by using Lemma 5.1 to reduce the security property (soundness or knowledge soundness) to the corresponding property for the basic variant of the Fiat–Shamir transformation (Section 3.6).

6.1 Proof of Theorem 6.1

We reduce the soundness of $(\mathcal{P}, \mathcal{V}) \coloneqq \mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ (Construction 4.3) to the soundness of $(\mathcal{P}_{\mathsf{std}}, \mathcal{V}_{\mathsf{std}}) \coloneqq \mathsf{FS}[\mathsf{IP}, \delta_\star]$ (Construction 3.16). Specifically, by Lemma 5.1, every $(t_h, t_p, t_{p^{-1}})$-query malicious argument prover $\tilde{\mathcal{P}}$ for $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ can be translated to a $\theta_\star(t_h, t_p, t_{p^{-1}})$-query malicious argument prover $\tilde{\mathcal{P}}_{\mathsf{std}} \coloneqq \mathsf{D2SAlgo}(\tilde{\mathcal{P}})$ for $\mathsf{FS}[\mathsf{IP}, \delta_\star]$ that "behaves the same" up to an additive error $\eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))$. The result then follows from Theorem 3.17, which states that the soundness of $\mathsf{FS}[\mathsf{IP}, \delta_\star]$ is determined by the state-restoration soundness of IP. Specifically:

$$
\begin{aligned}
&\varepsilon_{\mathsf{NARG}}(\lambda, (t_h, t_p, t_{p^{-1}}), n) \\
&= \Pr\left[\begin{array}{l} |\mathbb{x}| \le n \\ \wedge\, \mathbb{x} \notin \mathcal{L}(\mathcal{R}) \\ \wedge\, \mathcal{V}^{h,p}(\mathbb{x}, \pi) = 1 \end{array} \,\middle|\, \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}^{h,p,p^{-1}} \end{array}\right] && \text{(by Definition 3.4)} \\
&\le \Pr\left[\begin{array}{l} |\mathbb{x}| \le n \\ \wedge\, \mathbb{x} \notin \mathcal{L}(\mathcal{R}) \\ \wedge\, \mathcal{V}_{\mathsf{std}}^{\boldsymbol{f}}(\mathbb{x}, \pi) = 1 \end{array} \,\middle|\, \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}_{\mathsf{IP}}(\lambda, n) \\ (\mathbb{x}, \pi) \coloneqq \mathsf{D2SAlgo}^{\boldsymbol{f}}(\tilde{\mathcal{P}}) \end{array}\right] + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) && \text{(by Lemma 5.1)} \\
&\le \varepsilon_{\mathsf{FS}[\mathsf{IP}, \delta_\star]}(\lambda, \theta_\star(t_h, t_p, t_{p^{-1}}), n) + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) && \text{(by Definition 3.4)} \\
&\le \varepsilon_{\mathsf{IP}}^{\mathrm{sr}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n) + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))\,, && \text{(by Theorem 3.17)}
\end{aligned}
$$

where $\delta_\star$ is defined in Eq. 21. Finally, by combining the above derivation with the definition of $\eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))$ in Equation 5 in Lemma 5.1, we obtain the upper bound claimed in the theorem:

$$
\begin{aligned}
&\varepsilon_{\mathsf{NARG}}(\lambda, (t_h, t_p, t_{p^{-1}}), n) \\
&\le \varepsilon_{\mathsf{IP}}^{\mathrm{sr}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n) + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) && \text{(by the above derivation)} \\
&= \varepsilon_{\mathsf{IP}}^{\mathrm{sr}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n) \\
&\quad + \tfrac{t_h(t_h + 3t_p + 2L_{\mathbf{P}} + 1) + (t_p + L_{\mathbf{P}})(2t_p + 3L_{\mathbf{P}} - 1) + t_{p^{-1}}(2t_h + 2t_p + t_{p^{-1}} - 1)}{2|\Sigma|^c} && \text{(by Equation 5)} \\
&\quad + \textstyle\sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) \\
&\le \varepsilon_{\mathsf{IP}}^{\mathrm{sr}}(\delta_\star, \theta_\star(t), n) + \frac{4t^2 + \max_{i \in [\mathsf{k}]} \left\lceil \frac{\ell_{\mathbf{P}}(i)}{r} \right\rceil \left(t + \max_{i \in [\mathsf{k}]} \frac{\ell_{\mathbf{P}}(i)}{r}\right)}{|\Sigma|^c} + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) \\
&= \varepsilon_{\mathsf{IP}}^{\mathrm{sr}}(\delta_\star, \theta_\star(t), n) + \eta_\star(\lambda, t)\,. && \text{(by definition of } \eta_\star(\lambda, t))
\end{aligned}
$$

6.2 Proof of Theorem 6.2

First we describe how to obtain a knowledge extractor $\mathcal{E}$ for $\mathcal{V}$ from the knowledge extractor $\mathcal{E}_{\mathsf{std}}$ for $\mathcal{V}_{\mathsf{std}}$; this additionally relies on the two algorithms $\mathsf{D2SAlgo}$ and $\mathsf{D2STrace}$ from Lemma 5.1.

Construction 6.3. *Let $\mathcal{E}_{\mathsf{std}}$ be the knowledge extractor for $\mathcal{V}_{\mathsf{std}}$. The knowledge extractor $\mathcal{E}$ for $\mathcal{V}$ receives as input an instance $\mathbb{x}$, argument string π, query-answer trace tr of the argument prover $\tilde{\mathcal{P}}$, query-answer trace $\mathsf{tr}_{\mathcal{V}}$ of the argument verifier, and (black-box access to) the IP prover $\tilde{\mathcal{P}}$, and works as follows.*

$\mathcal{E}(\mathbb{x}, \pi, \mathsf{tr}, \mathsf{tr}_{\mathcal{V}}, \tilde{\mathcal{P}})$:

1. *Compute the query-answer trace* $\mathsf{tr}_{\mathsf{std}} \coloneqq \mathsf{D2STrace}(\mathsf{tr} \| \mathsf{tr}_{\mathcal{V}})$.

2. *Set* $\mathsf{tr}_{\mathsf{std},\mathcal{V}}$ *to be the suffix of* $\mathsf{tr}_{\mathsf{std}}$ *of length* k.
3. *Let* $\tilde{\mathcal{P}}_{\mathsf{std}} := \mathsf{D2SAlgo}(\tilde{\mathcal{P}})$.
4. *Compute* $\mathbb{w} := \mathcal{E}_{\mathsf{std}}(\mathbb{x}, \pi, \mathsf{tr}_{\mathsf{std}}, \mathsf{tr}_{\mathsf{std},\mathcal{V}}, \tilde{\mathcal{P}}_{\mathsf{std}})$.
5. *Output* $\mathbb{w}$.

We reduce the knowledge soundness of $(\mathcal{P}_{\mathsf{std}}, \mathcal{V}_{\mathsf{std}}) := \mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ (Construction 4.3) to the knowledge soundness of $(\mathcal{P}, \mathcal{V}) := \mathsf{FS}[\mathsf{IP}, \delta_\star]$ (Construction 3.16). Specifically, by Lemma 5.1, every $(t_h, t_p, t_{p^{-1}})$-query malicious argument prover $\tilde{\mathcal{P}}$ for $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ can be translated to a $\theta_\star(t_h, t_p, t_{p^{-1}})$-query malicious argument prover $\tilde{\mathcal{P}}_{\mathsf{std}} := \mathsf{D2SAlgo}(\tilde{\mathcal{P}})$ for $\mathsf{FS}[\mathsf{IP}, \delta_\star]$ that "behaves the same" up to an additive error $\eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))$. The result then follows from Theorem 3.18, which states that the knowledge soundness of $\mathsf{FS}[\mathsf{IP}, \delta_\star]$ is determined by the state-restoration knowledge soundness of IP. Specifically:

$$
\begin{aligned}
&\kappa_{\mathsf{NARG}}(\lambda, (t_h, t_p, t_{p^{-1}}), n, \delta_{\tilde{\mathcal{P}}}) \\
&= \Pr\left[\begin{array}{l|l} |\mathbb{x}| \le n & (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ \wedge\, (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} & (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ \wedge\, b = 1 & b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \\ & \mathbb{w} \leftarrow \mathcal{E}(\mathbb{x}, \pi, \mathsf{tr}, \mathsf{tr}_{\mathcal{V}}, \tilde{\mathcal{P}}) \end{array}\right] && \text{(by Definition 3.7)} \\
&= \Pr\left[\begin{array}{l|l} |\mathbb{x}| \le n & (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ \wedge\, (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} & (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ \wedge\, b = 1 & b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \\ & \mathsf{tr}_{\mathsf{std}} := \mathsf{D2STrace}(\mathsf{tr} \| \mathsf{tr}_{\mathcal{V}}) \\ & \mathsf{tr}_{\mathsf{std},\mathcal{V}} \text{ is the suffix of } \mathsf{tr}_{\mathsf{std}} \\ & \mathbb{w} \leftarrow \mathcal{E}_{\mathsf{std}}(\mathbb{x}, \pi, \mathsf{tr}_{\mathsf{std}}, \mathsf{tr}_{\mathsf{std},\mathcal{V}}, \tilde{\mathcal{P}}_{\mathsf{std}}) \end{array}\right] && \text{(by Construction 6.3)} \\
&\le \Pr\left[\begin{array}{l|l} |\mathbb{x}| \le n & \boldsymbol{f} \leftarrow \mathcal{D}_{\mathsf{IP}}(\lambda, n) \\ \wedge\, (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} & (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}} \tilde{\mathcal{P}}_{\mathsf{std}}^{\boldsymbol{f}} \\ \wedge\, b = 1 & b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}_{\mathsf{std}}^{\boldsymbol{f}}(\mathbb{x}, \pi) \\ & \mathbb{w} \leftarrow \mathcal{E}_{\mathsf{std}}(\mathbb{x}, \pi, \mathsf{tr}, \mathsf{tr}_{\mathcal{V}}, \tilde{\mathcal{P}}_{\mathsf{std}}) \end{array}\right] + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) && \text{(by Lemma 5.1)} \\
&\le \kappa_{\mathsf{NARG}}(\lambda, (t_h, t_p, t_{p^{-1}}), n, \delta_{\tilde{\mathcal{P}}_{\mathsf{std}}}) + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) && \text{(by Definition 3.7)} \\
&\le \kappa_{\mathsf{NARG}}\Big(\lambda, \theta_\star(t_h, t_p, t_{p^{-1}}), n, \delta_{\tilde{\mathcal{P}}} + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))\Big) + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) && \text{(by Lemma 5.1)} \\
&\le \kappa_{\mathsf{IP}}^{\mathrm{sr}}\Big(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n, \delta_{\tilde{\mathcal{P}}} + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))\Big) + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))\,. && \text{(by Theorem 3.18)}
\end{aligned}
$$

Finally, note that $\eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) \le \eta_\star(\lambda, t)$ (as argued in Section 6.1).

Next, we discuss the running time of $\mathcal{E}$ on $\tilde{\mathcal{P}}$. The extractor $\mathcal{E}$ runs $\mathsf{D2SAlgo}$ on both traces in time $t_h \cdot n \cdot \log t_h + (t_p + t_{p^{-1}})^2 \cdot (\log(t_p + t_{p^{-1}}) + \log t_h) \cdot c \cdot ((r + c) \cdot \log|\Sigma| + n) + t_p \cdot (t_{\boldsymbol{\varphi}^{-1}} + t_{\boldsymbol{\psi}})$, and then emulates the execution of $\mathcal{E}_{\mathsf{std}}$ on $\tilde{\mathcal{P}}_{\mathsf{std}} := \tilde{\mathcal{P}}_{\mathsf{std}}(\tilde{\mathcal{P}})$. We deduce that

$$
\begin{aligned}
&\mathbf{et}_{\mathsf{NARG}}(\lambda, (t_h, t_p, t_{p^{-1}}), n, \delta_{\tilde{\mathcal{P}}}, \tau_{\tilde{\mathcal{P}}}) \\
&\le \mathbf{et}_{\mathsf{IP}}^{\mathrm{sr}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n, \delta_{\tilde{\mathcal{P}}_{\mathsf{std}}}, \tau_{\tilde{\mathcal{P}}_{\mathsf{std}}}) + \\
&\qquad t_h \cdot n \cdot \log t_h + (t_p + t_{p^{-1}})^2 \cdot (\log(t_p + t_{p^{-1}}) + \log t_h) \cdot c \cdot ((r + c) \cdot \log|\Sigma| + n) + t_p \cdot (t_{\boldsymbol{\varphi}^{-1}} + t_{\boldsymbol{\psi}})\,.
\end{aligned}
$$

Since $\delta_{\tilde{\mathcal{P}}_{\mathsf{std}}} \leq \delta_{\tilde{\mathcal{P}}} + \eta_\star(\lambda, t)$, $\tau_{\tilde{\mathcal{P}}_{\mathsf{std}}} \leq \tau_{\tilde{\mathcal{P}}} + t_h \cdot n \cdot \log t_h + (t_p + t_{p^{-1}})^2 \cdot (\log(t_p + t_{p^{-1}}) + \log t_h) \cdot c \cdot ((r + c) \cdot \log|\Sigma| + n) + t_p \cdot (t_{\varphi^{-1}} + t_{\psi})$, and using Theorem 3.18 we get that

$$
\begin{aligned}
&\mathbf{et}_{\mathsf{NARG}}(\lambda, (t_h, t_p, t_{p^{-1}}), n, \delta_{\tilde{\mathcal{P}}}, \tau_{\tilde{\mathcal{P}}}) \\
&\leq \mathbf{et}^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n, \delta_{\tilde{\mathcal{P}}_{\mathsf{std}}}, \tau_{\tilde{\mathcal{P}}_{\mathsf{std}}}) \\
&\quad + t_h \cdot n \cdot \log t_h + (t_p + t_{p^{-1}})^2 \cdot (\log(t_p + t_{p^{-1}}) + \log t_h) \cdot c \cdot ((r + c) \cdot \log|\Sigma| + n) + t_p \cdot (t_{\varphi^{-1}} + t_{\psi}) .
\end{aligned}
$$

We can simplify the expression by using the fact that $t_h + t_p + t_{p^{-1}} \leq t$:

$$
\begin{aligned}
\mathbf{et}_{\mathsf{NARG}}(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}, \tau_{\tilde{\mathcal{P}}}) \leq &\mathbf{et}^{\mathrm{sr}}_{\mathsf{IP}}\Big(\delta_\star, \theta_\star(t), n, \delta_{\tilde{\mathcal{P}}} + \eta_\star(\lambda, t), \tau_{\tilde{\mathcal{P}}} + t^2 \cdot \log t \cdot ((r + c^2) \log|\Sigma| + n)\Big) \\
&+ t^2 \cdot \log t \cdot ((r + c^2) \log|\Sigma| + n) .
\end{aligned}
$$

The straightline case. The above analysis directly specializes to the straightline case. Suppose that $\mathbf{E}^{\mathrm{sr}}$ is a straightline extractor (Definition 3.13): it is a deterministic algorithm that does not need access to the IP state-restoration prover $\tilde{\mathbf{P}}^{\mathrm{sr}}$. Then, by Theorem 3.18, $\mathcal{E}_{\mathsf{std}}$ is a straightline extractor (Definition 3.5): it is a deterministic algorithm that does not need access to the argument prover $\tilde{\mathcal{P}}_{\mathsf{std}}$. In turn, $\mathcal{E}$ in Construction 6.3 is a straightline extractor (Definition 3.5): it is a deterministic algorithm that does not need access to the argument prover $\tilde{\mathcal{P}}$. In this case, the knowledge soundness error bound does not depend on the failure probability of the prover, and simplifies to

$$
\begin{aligned}
\kappa_{\mathsf{NARG}}(\lambda, (t_h, t_p, t_{p^{-1}}), n) &\leq \kappa^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n) + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) \\
&\leq \kappa^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n) + \eta_\star(\lambda, t) .
\end{aligned}
$$

Similarly, the extraction time bound does not depend on the failure probability or running time of the prover, and simplifies to

$$
\begin{aligned}
&\mathbf{et}_{\mathsf{NARG}}(\lambda, (t_h, t_p, t_{p^{-1}}), n) \\
&\leq \quad \mathbf{et}^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n) \\
&\quad + t_h \cdot n \cdot \log t_h + (t_p + t_{p^{-1}})^2 \cdot (\log(t_p + t_{p^{-1}}) + \log t_h) \cdot c \cdot ((r + c) \cdot \log|\Sigma| + n) + t_p \cdot (t_{\varphi^{-1}} + t_{\psi}) .
\end{aligned}
$$

We simplify the expression using $t_h + t_p + t_{p^{-1}} \leq t$:

$$
\mathbf{et}_{\mathsf{NARG}}(\lambda, t, n) \leq \mathbf{et}^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t), n) + t^2 \cdot \log t \cdot ((r + c^2) \log|\Sigma| + n) .
$$

7 Zero knowledge

We formally state and prove the theorem about zero knowledge for our Fiat–Shamir transformation. Let $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ be a public-coin IP for a relation $\mathcal{R}$. Let Σ be a finite alphabet and cdc be a codec for IP over Σ with bias $\varepsilon_{\mathsf{cdc}}$ (see Definition 4.1). Let $\mathcal{D}_{\mathfrak{S}}$ be an ideal permutation distribution over Σ with capacity $c \in \mathbb{N}$ and rate $r \in \mathbb{N}$ (see Definition 4.2). For a salt size $\delta \in \mathbb{N}$, let $\mathsf{NARG} := \mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ be the non-interactive argument for $\mathcal{R}$ in the $\mathcal{D}_{\mathfrak{S}}$-oracle model constructed in Construction 4.3.

Theorem 7.1. *If* IP *has honest-verifier zero-knowledge error* z_{IP} *(see Definition 7.3) then* NARG *has zero-knowledge error* z_{NARG} *(see Definition 7.4) such that, for every security parameter* $\lambda \in \mathbb{N}$*, query bound* $t \in \mathbb{N}$*, and instance size bound* $n \in \mathbb{N}$*,*

$$z_{\mathsf{NARG}}(\lambda, t, n) \leq z_{\mathsf{IP}} + \frac{t}{|\Sigma|^{\min(\delta,c)}} + \frac{t \cdot \sum_{i\in[\mathsf{k}]} \lceil \ell_{\mathbf{V}}(i)/r \rceil}{|\Sigma|^{r+c}} .$$

The definitions of honest-verifier zero-knowledge for IPs and zero-knowledge for non-interactive arguments are in Section 7.1. The simulator is in Section 7.2. The analysis of the simulator, establishing the theorem, is in Section 7.3.

7.1 Definitions for zero knowledge

Definition 7.2. *The IP verifier's* **view** *in* $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *on the instance-witness pair* $(\mathbb{x}, \mathbb{w})$*, denoted* $\mathsf{View}_{\mathsf{IP}}(\mathbf{P}, \mathbf{V}, \mathbb{x}, \mathbb{w})$*, is the random variable* $\big(\mathbb{x}, \rho, (\alpha_i)_{i\in[\mathsf{k}]}\big)$ *where:*

- ρ *is a random choice of randomness for the IP verifier* $\mathbf{V}$*; and*
- $(\alpha_i)_{i\in[\mathsf{k}]}$ *are the prover messages received in an interaction between* $\mathbf{P}(\mathbb{x}, \mathbb{w})$ *and* $\mathbf{V}(\mathbb{x}, \rho)$*.*

Note that the honest IP prover $\mathbf{P}(\mathbb{x}, \mathbb{w})$ *may use its own private randomness (and is not part of the IP verifier's view). If the IP is public-coin then the view shows each round's randomness:* $\big(\mathbb{x}, (\rho_i)_{i\in[\mathsf{k}]}, (\alpha_i)_{i\in[\mathsf{k}]}\big)$*.*

Definition 7.3. $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *for a relation* $\mathcal{R}$ *has* **honest-verifier zero-knowledge error** z_{IP} *if there exists a polynomial-time probabilistic algorithm* $\mathbf{S}$ *such that for every instance-witness pair* $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ *the following random variables are* $z_{\mathsf{IP}}(\mathbb{x})$*-close in statistical distance:*

$$\mathsf{View}_{\mathsf{IP}}(\mathbf{P}, \mathbf{V}, \mathbb{x}, \mathbb{w}) \;\; \text{and} \;\; \mathbf{S}(\mathbb{x}) .$$

We additionally define $z_{\mathsf{IP}}(n) := \max_{\substack{(\mathbb{x},\mathbb{w})\in\mathcal{R} \\ |\mathbb{x}|\leq n}} z_{\mathsf{IP}}(\mathbb{x})$*.*

Definition 7.4. *A non-interactive argument* $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ *for a relation* $\mathcal{R}$ *has* **(adaptive) zero-knowledge error** z_{NARG} *(in the EPROM) if there exists a probabilistic polynomial-time simulator* $\mathcal{S}$ *such that, for every security parameter* $\lambda \in \mathbb{N}$*, query bound* $t \in \mathbb{N}$*,* t*-query admissible adversary* $\mathcal{A}$*, and instance bound* $n \in \mathbb{N}$*, the following two distributions are* $z_{\mathsf{NARG}}(\lambda, t, n)$*-close in statistical distance:*

$$\mathcal{D}_{\text{real}} := \left\{ \mathsf{out} \;\middle|\; \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \leftarrow \mathcal{A}^{\boldsymbol{f}} \\ \pi \leftarrow \mathcal{P}^{\boldsymbol{f}}(\mathbb{x}, \mathbb{w}) \\ \mathsf{out} \leftarrow \mathcal{A}^{\boldsymbol{f}}(\mathsf{aux}, \pi) \end{array} \right\} \;\; \text{and} \;\; \mathcal{D}_{\text{sim}} := \left\{ \mathsf{out} \;\middle|\; \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \leftarrow \mathcal{A}^{\boldsymbol{f}} \\ (\pi, \mu) \leftarrow \mathcal{S}^{\boldsymbol{f}}(\mathbb{x}) \\ \mathsf{out} \leftarrow \mathcal{A}^{\boldsymbol{f}[\mu]}(\mathsf{aux}, \pi) \end{array} \right\} .$$

Above, $\mathcal{A}$ *is admissible if it always outputs* $\mathbb{x}, \mathbb{w}$ *such that* $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ *and* $|\mathbb{x}| \leq n$*. With* $f[\mu]$*, we denote the function* f *modified to be consistent with the query-answer list* μ*.*

In the simulated-world experiment (the one on the right), the simulator has access to the random oracle, and "programs" the random oracle via a list μ of query-answer pairs.

7.2 The simulator

We describe the simulator $\mathcal{S}$ that we use to establish zero knowledge for NARG.

Construction 7.5. *The simulator is an algorithm $\mathcal{S}^{h,p}(\mathbb{x})$ that works as follows. Below we denote by $\mathbf{S}$ the honest-verifier zero-knowledge simulator of* IP *(see Definition 7.3).*

$\mathcal{S}^{h,p}(\mathbb{x})$:
1. *Sample a simulated view of the IP verifier:* $\big(\mathbb{x}, (\rho_i)_{i\in[\mathsf{k}]}, (\alpha_i)_{i\in[\mathsf{k}]}\big) \leftarrow \mathbf{S}(\mathbb{x})$.
2. *Initialize the sponge state with the instance:* $\mathsf{st}'_0 := \mathsf{DS.Start}^h(\mathbb{x})$.
3. *Sample a random salt* $\boldsymbol{\tau} \in \Sigma^\delta$.
4. *Absorb the salt:* $\mathsf{st}_0 := \mathsf{DS.Absorb}^p(\mathsf{st}'_0, \boldsymbol{\tau})$.
5. *For* $i = 1, \ldots, \mathsf{k}$*:*
   (a) *Encode the prover message:* $\widehat{\boldsymbol{\alpha}}_i := \varphi_i(\alpha_i)$.
   (b) *Absorb the encoded prover message:* $\mathsf{st}'_i := \mathsf{DS.Absorb}^p(\mathsf{st}_{i-1}, \widehat{\boldsymbol{\alpha}}_i)$.
   (c) *Sample an encoded verifier message:* $\widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i)$.
   (d) *Sample* $(\mu_{p,i}, \mu_{p^{-1},i}, \mathsf{st}_i) \leftarrow \mathsf{ProgramBlocks}(i, \mathsf{st}'_i, \widehat{\boldsymbol{\rho}}_i)$ *(see below).*
6. *Set the argument string* $\pi := (\boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]})$.
7. *Let* μ_h *be the empty list. (The oracle* h *is not programmed.)*
8. *Set* μ_p *to be the concatenation of the query-answer lists* $(\mu_{p,i})_{i\in[\mathsf{k}]}$.
9. *Set* $\mu_{p^{-1}}$ *to be the concatenation of the query-answer lists* $(\mu_{p^{-1},i})_{i\in[\mathsf{k}]}$.
10. *Set* $\mu := (\mu_h, \mu_p, \mu_{p^{-1}})$ *to be the list of all programmed locations for the oracles.*
11. *Output* (π, μ).

Note that $\mathcal{S}$ does not query the inverse permutation p^{-1} and does not program h.

Programming blocks. The auxiliary procedure $\mathsf{ProgramBlocks}$ used in Construction 7.5 outputs the permutation blocks to be programmed: it receives as input the round index $i \in [\mathsf{k}]$, a sponge state $\mathsf{st} \in \Sigma^{r+c} \times [0,r] \times [0,r]$, and a message $\widehat{\boldsymbol{\rho}}_i \in \Sigma^{\ell_{\mathbf{V}}(i)}$, and outputs a query-answer list μ_p for the permutation oracle p, a corresponding query-answer list $\mu_{p^{-1}}$ for the inverse permutation oracle p^{-1}, and a new sponge state st_i corresponding to the sponge state for the next round.

$\mathsf{ProgramBlocks}(i, \mathsf{st}, \widehat{\boldsymbol{\rho}}_i)$:
1. Set $L_{\mathbf{V}}(i) := \left\lceil \frac{\ell_{\mathbf{V}}(i)}{r} \right\rceil$.
2. Parse the sponge state st as a tuple $((\boldsymbol{s}_{\mathrm{R},0}, \boldsymbol{s}_{\mathrm{C},0}), i_{\mathrm{A}}, i_{\mathrm{S}})$.
3. For every $j \in [L_{\mathbf{V}}(i)]$, sample random $\boldsymbol{s}_{\mathrm{C},j} \leftarrow \mathcal{U}(\Sigma^c)$.
4. Sample random $\boldsymbol{z}_i \leftarrow \mathcal{U}(\Sigma^{r-(\ell_{\mathbf{V}}(i) \bmod r)})$.
5. Parse $\widehat{\boldsymbol{\rho}}_i \| \boldsymbol{z}_i$ into segments $(\boldsymbol{s}_{\mathrm{R},1}, \boldsymbol{s}_{\mathrm{R},2}, \ldots, \boldsymbol{s}_{\mathrm{R},L_{\mathbf{V}}(i)})$, each of length r.
6. Set $\mu_{p,i} := ((\boldsymbol{s}_{\mathrm{R},i-1}, \boldsymbol{s}_{\mathrm{C},i-1}), (\boldsymbol{s}_{\mathrm{R},i}, \boldsymbol{s}_{\mathrm{C},i}))_{i\in[L_{\mathbf{V}}(i)]}$ and $\mu_{p^{-1},i} := ((\boldsymbol{s}_{\mathrm{R},i}, \boldsymbol{s}_{\mathrm{C},i}), (\boldsymbol{s}_{\mathrm{R},i-1}, \boldsymbol{s}_{\mathrm{C},i-1}))_{i\in[L_{\mathbf{V}}(i)]}$.
7. Set $\mathsf{st}_i := ((\boldsymbol{s}_{\mathrm{R},L_{\mathbf{V}}(i)}, \boldsymbol{s}_{\mathrm{C},L_{\mathbf{V}}(i)}), i_{\mathrm{A}}, i_{\mathrm{S}})$ where $i_{\mathrm{A}} := r$ and $i_{\mathrm{S}} := \ell_{\mathbf{V}}(i) \bmod r$.
8. Return $(\mu_{p,i}, \mu_{p^{-1},i}, \mathsf{st}_i)$.

7.3 Proof of Theorem 7.1

We prove the theorem via a hybrid argument; hybrids are summarized in Figure 5. Throughout we consider an (admissible) adversary $\mathcal{A}$ that makes at most t_h queries to h, at most t_p queries to p, and at most $t_{p^{-1}}$ queries

to p^{-1}. We show that the statistical distance between $\mathcal{D}_{\text{real}}$ and $\mathcal{D}_{\text{sim}}$, by adding the errors across all hybrids, is at most

$$z_{\mathsf{IP}}(n) + \frac{t_p + t_{p^{-1}}}{|\Sigma|^{\min(\delta,c)}} + \frac{(t_p + t_{p^{-1}}) \cdot \sum_{i \in [\mathsf{k}]} \lceil \ell_{\mathbf{V}}(i)/r \rceil}{|\Sigma|^{r+c}} + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) \,.$$

The bound in the theorem statement follows from the fact that $t_h + t_p + t_{p^{-1}} \leq t$.

Hyb_0. This is the distribution $\mathcal{D}_{\text{real}}$. The argument prover is displayed in Figure 5 in the non-boxed lines. To simplify hybrid exposition, the argument prover also computes the last verifier message ρ_{k} (even though it does not use it).

Hyb_1. In this hybrid we program the permutation p. We alter line 11 and, instead of using DS to produce encoded verifier messages, we sample fresh $\widehat{\boldsymbol{\rho}}_1 \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(1)}), \dots, \widehat{\boldsymbol{\rho}}_{\mathsf{k}} \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(\mathsf{k})})$ uniformly at random and use the procedure $\mathsf{ProgramBlocks}$ (see line 6) to obtain query-answer lists used to program the oracles p and p^{-1}. Overall, $L_{\mathbf{V}} := \sum_{i \in [\mathsf{k}]} L_{\mathbf{V}}(i) = \sum_{i \in [\mathsf{k}]} \lceil \ell_{\mathbf{V}}(i)/r \rceil$ locations are programmed.

Claim 7.6. *The statistical distance between Hyb_0 and Hyb_1 is at most*

$$\frac{t_p + t_{p^{-1}}}{|\Sigma|^{\min(\delta,c)}} + \frac{(t_p + t_{p^{-1}}) \cdot \sum_{i \in [\mathsf{k}]} \lceil \ell_{\mathbf{V}}(i)/r \rceil}{|\Sigma|^{r+c}} \,.$$

We prove the claim by showing that Hyb_0 and Hyb_1 are perfectly indistinguishable conditioned on a certain bad event not happening, and and upper bounding the probability of this bad event.

Consider the following two predicates.

- E_{start}. For a query-answer trace tr and permutation state $\boldsymbol{s}_0 \in \Sigma^{r+c}$, we define the predicate

$$E_{\text{start}}(\mathsf{tr}, \boldsymbol{s}_0) := \text{“} \exists\, \boldsymbol{s}_1' \in \Sigma^{r+c} \colon (\text{‘}p\text{’}, \boldsymbol{s}_0, \boldsymbol{s}_1') \in \mathsf{tr} \;\vee\; (\text{‘}p^{-1}\text{’}, \boldsymbol{s}_1', \boldsymbol{s}_0) \in \mathsf{tr} \text{”} \,.$$

- E_{squeeze}. For a query-answer trace tr and permutation states $\boldsymbol{s}_1, \dots, \boldsymbol{s}_{L_{\mathbf{V}}} \in \Sigma^{r+c}$, we define the predicate

$$E_{\text{squeeze}}(\mathsf{tr}, (\boldsymbol{s}_1, \dots, \boldsymbol{s}_{L_{\mathbf{V}}})) := \text{“} \exists\, \iota \in [L_{\mathbf{V}} - 1]\,,\; \boldsymbol{s}_{\text{out}} \in \Sigma^{r+c} \colon \boldsymbol{s}_{\text{out}} \neq \boldsymbol{s}_{\iota+1} \wedge (\text{‘}p\text{’}, \boldsymbol{s}_\iota, \boldsymbol{s}_{\text{out}}) \in \mathsf{tr} \text{”} \,.$$

Next, consider the following two events.

- In Hyb_0, E_0 is the (bad) event that $E_{\text{start}}(\mathsf{tr}, \boldsymbol{s}_0) \vee E_{\text{squeeze}}(\mathsf{tr}, (\boldsymbol{s}_1, \dots, \boldsymbol{s}_{L_{\mathbf{V}}}))$ holds, where:
  - tr is the query-answer trace of $\mathcal{A}^{h,p,p^{-1}}$ at the end of line 2;
  - $\boldsymbol{s}_0$ is the permutation state in st_0 after absorbing the salt $\boldsymbol{\tau}$ at the end of line 5; and
  - $\boldsymbol{s}_1, \dots, \boldsymbol{s}_{L_{\mathbf{V}}}$ are the permutations states in the encoded verifier messages $\widehat{\boldsymbol{\rho}}_1, \dots, \widehat{\boldsymbol{\rho}}_{\mathsf{k}}$ produced by $\mathsf{DS.Squeeze}^p$ in line 11 (across all k rounds).
- In Hyb_1, E_1 is the (bad) event that $E_{\text{start}}(\mathsf{tr}, \boldsymbol{s}_0) \vee E_{\text{squeeze}}(\mathsf{tr}, (\boldsymbol{s}_1, \dots, \boldsymbol{s}_{L_{\mathbf{V}}}))$ holds, where:
  - tr is as above (tr is the same in Hyb_1 and Hyb_0);
  - $\boldsymbol{s}_0$ is as above ($\boldsymbol{s}_0$ is the same in Hyb_1 and Hyb_0);
  - $\boldsymbol{s}_1, \dots, \boldsymbol{s}_{L_{\mathbf{V}}}$ are the permutations states contained in the query-answer lists $(\mu_{p,i})_{i \in [\mathsf{k}]}$ and $(\mu_{p^{-1},i})_{i \in [\mathsf{k}]}$ produced by $\mathsf{ProgramBlocks}$ in line 11 (across all k rounds).

We argue that $\Pr[E_0] \leq \Pr[E_1]$. This follows from observing that E_{start} is identical across Hyb_0 and Hyb_1, and that E_{squeeze} never holds in Hyb_0 (because there is no programming) In fact, p's outputs are consistent and therefore

$$\forall\, (`p', \boldsymbol{s}_\iota, \boldsymbol{s}_{\text{out}}), (`p', \boldsymbol{s}'_\iota, \boldsymbol{s}'_{\text{out}}) \in \mathsf{tr} \colon \boldsymbol{s}_\iota = \boldsymbol{s}'_\iota \implies \boldsymbol{s}_{\text{out}} = \boldsymbol{s}'_{\text{out}}\,,$$

which implies that E_{squeeze} never holds. Therefore:

$$\begin{aligned}
&\Pr[E_0] \\
&\leq \Pr\left[E_{\text{start}}(\mathsf{tr}, \boldsymbol{s}_0) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta) \\ \mathsf{st}'_0 \coloneqq \mathsf{DS.Start}^h(\mathbb{x}) \\ \mathsf{st}_0 \coloneqq \mathsf{DS.Absorb}^p(\mathsf{st}'_0, \boldsymbol{\tau}) \end{array} \right] \\
&\quad + \Pr\left[E_{\text{squeeze}}(\mathsf{tr}, (\boldsymbol{s}_1, \dots, \boldsymbol{s}_{L_{\mathbf{V}}})) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta) \\ \mathsf{st}'_0 \coloneqq \mathsf{DS.Start}^h(\mathbb{x}) \\ \mathsf{st}_0 \coloneqq \mathsf{DS.Absorb}^p(\mathsf{st}'_0, \boldsymbol{\tau}) \\ \textbf{for } i = 1, \dots, \mathsf{k} \\ \quad (\alpha_i, \mathsf{aux}_i) \coloneqq \mathbf{P}(\mathbb{x}, \mathbb{w}) \textbf{ if } i = 1 \textbf{ else } \mathbf{P}(\mathsf{aux}_{i-1}, \rho_{i-1}) \\ \quad \widehat{\boldsymbol{\alpha}}_i \coloneqq \varphi_i(\alpha_i) \\ \quad \mathsf{st}'_i \coloneqq \mathsf{DS.Absorb}^p(\mathsf{st}_{i-1}, \widehat{\boldsymbol{\alpha}}_i) \\ \quad (\widehat{\boldsymbol{\rho}}_i, \mathsf{st}'_i) \coloneqq \mathsf{DS.Squeeze}^p(\mathsf{st}'_i, \ell_{\mathbf{V}}(i)); \quad \rho_i \coloneqq \psi_i(\widehat{\boldsymbol{\rho}}_i) \end{array} \right] \\
&= \Pr\left[E_{\text{start}}(\mathsf{tr}, \boldsymbol{s}_0) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta) \\ \mathsf{st}'_0 \coloneqq \mathsf{DS.Start}^h(\mathbb{x}) \\ \mathsf{st}_0 \coloneqq \mathsf{DS.Absorb}^p(\mathsf{st}'_0, \boldsymbol{\tau}) \end{array} \right] + 0 \\
&\leq \Pr[E_1]\,.
\end{aligned}$$

We argue that $\mathsf{Hyb}_0 \mid \overline{E_0}$ identically distributed to $\mathsf{Hyb}_1 \mid \overline{E_1}$:

- In Hyb_0, p's answers are uniformly and independently sampled in Σ^{r+c}, and there is no programming.
- In Hyb_1, the permutation states $\boldsymbol{s}_1, \dots, \boldsymbol{s}_{L_{\mathbf{V}}}$ are uniformly and independently sampled in $\mathsf{ProgramBlocks}$. Since E_1 does not hold,

  $$\forall\, \iota \in [0, L_{\mathbf{V}}{-}1] \colon (`p', \boldsymbol{s}_\iota, \boldsymbol{s}_{\text{out}}) \in \mathsf{tr} \implies \boldsymbol{s}_{\text{out}} = \boldsymbol{s}_{\iota+1}$$

  which implies that p's answers are consistent.

Together with Claims 7.7 and 7.8, this implies:

$$\begin{aligned}
\Delta\left(\mathsf{Hyb}_0, \mathsf{Hyb}_1\right) &\leq \max(\Pr[E_0], \Pr[E_1]) + \Delta\left(\mathsf{Hyb}_0 \;\middle|\; \overline{E_0}, \mathsf{Hyb}_1 \;\middle|\; \overline{E_1}\right) && \text{(by [CY24, Claim 1.2.10])} \\
&= \Pr[E_1] + 0 && \text{(argued above)} \\
&\leq \frac{t_p + t_{p^{-1}}}{|\Sigma|^{\min(\delta, c)}} + \frac{(t_p + t_{p^{-1}}) \cdot \sum_{i \in [\mathsf{k}]} \lceil \ell_{\mathbf{V}}(i)/r \rceil}{|\Sigma|^{r+c}}\,. && \text{(by Claims 7.7 and 7.8)}
\end{aligned}$$

Claim 7.7. *The following holds:*

$$\Pr\left[E_{\mathrm{start}}(\mathsf{tr}, \boldsymbol{s}_0) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^{\delta}) \\ \mathsf{st}_0' := \mathsf{DS.Start}^{h}(\mathbb{x}) \\ \mathsf{st}_0 := \mathsf{DS.Absorb}^{p}(\mathsf{st}_0', \boldsymbol{\tau}) \end{array} \right] \leq \frac{t_p + t_{p^{-1}}}{|\Sigma|^{\min(\delta, c)}} .$$

Proof. Let $L_\delta := \lceil \delta / r \rceil$ be the number of blocks in the salt string $\boldsymbol{\tau}$. Without loss of generality, we assume that $\mathcal{A}$ queries $\mathbb{x}$ to h and let $\boldsymbol{s}_{\mathrm{C},-L_\delta+1} := h(\mathbb{x}) \in \Sigma^c$ be the query answer. (Construct the $(t_h + 1, t_p, t_{p^{-1}})$-query adversary $\mathcal{A}'$ that internally runs $\mathcal{A}$ and, after $\mathcal{A}$ outputs $(\mathbb{x}, \mathbb{w}, \mathsf{aux})$, queries $\mathbb{x}$ to h and outputs $(\mathbb{x}, \mathbb{w}, \mathsf{aux})$.) We distinguish two cases.

- *Case 1: $L_\delta = 1$ (i.e. $\delta \leq r$).* The permutation state $\boldsymbol{s}_0$ is $(\boldsymbol{\tau} \| 0^{r-\delta}, h(\mathbb{x})) \in \Sigma^{r+c}$. Therefore:

$$\begin{aligned} &\Pr\left[E_{\mathrm{start}}(\mathsf{tr}, \boldsymbol{s}_0) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^{\delta}) \\ \mathsf{st}_0' := \mathsf{DS.Start}^{h}(\mathbb{x}) \\ \mathsf{st}_0 := \mathsf{DS.Absorb}^{p}(\mathsf{st}_0', \boldsymbol{\tau}) \end{array} \right] \\ &\leq \Pr\left[(\text{‘}p\text{’}, \boldsymbol{s}_0, \cdot) \in \mathsf{tr} \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^{\delta}) \\ \boldsymbol{s}_0 := (\boldsymbol{\tau} \| 0^{r-\delta}, h(\mathbb{x})) \end{array} \right] + \Pr\left[(\text{‘}p^{-1}\text{’}, \cdot, \boldsymbol{s}_0) \in \mathsf{tr} \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^{\delta}) \\ \boldsymbol{s}_0 := (\boldsymbol{\tau} \| 0^{r-\delta}, h(\mathbb{x})) \end{array} \right] \\ &\leq \frac{t_p}{|\Sigma|^{\delta}} + \frac{t_{p^{-1}}}{|\Sigma|^{\delta}} . \end{aligned}$$

- *Case 2: $L_\delta > 1$ (i.e. $\delta > r$).* Parse $\boldsymbol{\tau} \| 0^{r - (\delta \bmod r)}$ into the sequence of L_δ blocks $(\boldsymbol{s}_{\mathrm{R},-L_\delta+1} \| \cdots \| \boldsymbol{s}_{\mathrm{R},0})$ that are to be absorbed. Either all these blocks appropriately appear in the trace, or they do not.

  (a) Suppose that there exists a sequence of queries to p that match the salt, i.e., $\exists\, \boldsymbol{s}_{\mathrm{C},-L_\delta+2}, \ldots, \boldsymbol{s}_{\mathrm{C},-1} \in \Sigma^c$ such that

$$\forall\, j \in \{-L_\delta + 1, \ldots, 0\} \colon \quad \begin{array}{ll} & (\text{‘}p\text{’}, (\boldsymbol{s}_{\mathrm{R},j-1}, \boldsymbol{s}_{\mathrm{C},i-1}), (\boldsymbol{s}_{\mathrm{R},j}, \boldsymbol{s}_{\mathrm{C},j})) \in \mathsf{tr} \\ \vee & (\text{‘}p^{-1}\text{’}, (\boldsymbol{s}_{\mathrm{R},j}, \boldsymbol{s}_{\mathrm{C},t}), (\boldsymbol{s}_{\mathrm{R},j-1}, \boldsymbol{s}_{\mathrm{C},j-1})) \in \mathsf{tr} . \end{array}$$

  Then:

$$\Pr\left[E_{\mathrm{start}}(\mathsf{tr}, \boldsymbol{s}_0) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^{\delta}) \\ \mathsf{st}_0' := \mathsf{DS.Start}^{h}(\mathbb{x}) \\ \mathsf{st}_0 := \mathsf{DS.Absorb}^{p}(\mathsf{st}_0', \boldsymbol{\tau}) \end{array} \right]$$

$$\leq \Pr\left[\begin{array}{l}\forall j \in \{-L_\delta+1,\dots,0\}\colon\\ (\text{`}p\text{'}, (\boldsymbol{s}_{\mathrm{R},j-1},\cdot),(\boldsymbol{s}_{\mathrm{R},j},\cdot)) \in \mathsf{tr}\end{array}\;\middle|\;\begin{array}{l}(h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\mathbb{w},\mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}}\\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta)\\ (\boldsymbol{s}_{\mathrm{R},-L_\delta+1}\|\cdots\|\boldsymbol{s}_{\mathrm{R},0}) \coloneqq (\boldsymbol{\tau}\|0^{r-(\delta \bmod r)})\end{array}\right]$$
$$+ \Pr\left[\begin{array}{l}\forall j \in \{-L_\delta+1,\dots,0\}\colon\\ (\text{`}p^{-1}\text{'}, (\boldsymbol{s}_{\mathrm{R},j},\cdot),(\boldsymbol{s}_{\mathrm{R},j-1},\cdot)) \in \mathsf{tr}\end{array}\;\middle|\;\begin{array}{l}(h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\mathbb{w},\mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}}\\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta)\\ (\boldsymbol{s}_{\mathrm{R},-L_\delta+1}\|\cdots\|\boldsymbol{s}_{\mathrm{R},0}) \coloneqq (\boldsymbol{\tau}\|0^{r-(\delta \bmod r)})\end{array}\right]$$
$$\leq \frac{t_p}{|\Sigma|^\delta} + \frac{t_{p^{-1}}}{|\Sigma|^\delta}\,.$$

(b) Suppose that some blocks of the salt do not appear in the trace. Denote with j^* the last such index:

$$j^* \coloneqq \max_{j\in\{-L_\delta+1,\dots,0\}} \colon \begin{array}{l}\nexists\, \boldsymbol{s}_{\mathrm{C},-L_\delta+2},\dots,\boldsymbol{s}_{\mathrm{C},-1} \in \Sigma^c \text{ such that}\\ (\text{`}p\text{'}, (\boldsymbol{s}_{\mathrm{R},j-1},\boldsymbol{s}_{\mathrm{C},j-1}),(\boldsymbol{s}_{\mathrm{R},j},\boldsymbol{s}_{\mathrm{C},j})) \notin \mathsf{tr}\\ \wedge\, (\text{`}p^{-1}\text{'}, (\boldsymbol{s}_{\mathrm{R},j},\boldsymbol{s}_{\mathrm{C},j}),(\boldsymbol{s}_{\mathrm{R},j-1},\boldsymbol{s}_{\mathrm{C},j-1})) \notin \mathsf{tr}\,.\end{array} \tag{22}$$

In this case, while absorbing the salt in line 5, the capacity state $\boldsymbol{s}_{\mathrm{C},j^*+1}$ is sampled uniformly at random from Σ^c, and coincides with a query to the permutation oracle from the adversary. That is,

$$\Pr\left[E_{\mathrm{start}}(\mathsf{tr},\boldsymbol{s}_0)\;\middle|\;\begin{array}{l}(h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\mathbb{w},\mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}}\\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta)\\ \mathsf{st}_0' \coloneqq \mathsf{DS.Start}^h(\mathbb{x})\\ \mathsf{st}_0 \coloneqq \mathsf{DS.Absorb}^p(\mathsf{st}_0',\boldsymbol{\tau})\end{array}\right]$$
$$\leq \Pr\left[\begin{array}{l}\exists\, j^* \text{ as per Eq. 22 } \wedge\\ (\text{`}p\text{'},\cdot,(\cdot,\boldsymbol{s}_{\mathrm{C},j^*})) \in \mathsf{tr}\end{array}\;\middle|\;\begin{array}{l}(h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\mathbb{w},\mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}}\\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta)\\ \boldsymbol{s}_{\mathrm{C},-L_\delta+1},\dots,\boldsymbol{s}_{\mathrm{C},0} \leftarrow \mathcal{U}(\Sigma^c)\\ (\boldsymbol{s}_{\mathrm{R},-L_\delta+1}\|\cdots\|\boldsymbol{s}_{\mathrm{R},0}) \coloneqq (\boldsymbol{\tau}\|0^{r-(\delta \bmod r)})\end{array}\right]$$
$$+ \Pr\left[\begin{array}{l}\exists\, j^* \text{as per Eq. 22 } \wedge\\ (\text{`}p^{-1}\text{'},(\cdot,\boldsymbol{s}_{\mathrm{C},j^*}),\cdot) \in \mathsf{tr}\end{array}\;\middle|\;\begin{array}{l}(h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)\\ (\mathbb{x},\mathbb{w},\mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}}\\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta)\\ \boldsymbol{s}_{\mathrm{C},-L_\delta+1},\dots,\boldsymbol{s}_{\mathrm{C},0} \leftarrow \mathcal{U}(\Sigma^c)\\ (\boldsymbol{s}_{\mathrm{R},-L_\delta+1}\|\cdots\|\boldsymbol{s}_{\mathrm{R},0}) \coloneqq (\boldsymbol{\tau}\|0^{r-(\delta \bmod r)})\end{array}\right]$$
$$\leq \frac{t_p}{|\Sigma|^c} + \frac{t_{p^{-1}}}{|\Sigma|^c}\,.$$

Putting together all above (mutually-exclusive) cases, we conclude that:

$$\Pr\left[E_{\text{start}}(\mathsf{tr}, \boldsymbol{s}_0) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^{\delta}) \\ \mathsf{st}_0' \coloneqq \mathsf{DS.Start}^{h}(\mathbb{x}) \\ \mathsf{st}_0 \coloneqq \mathsf{DS.Absorb}^{p}(\mathsf{st}_0', \boldsymbol{\tau}) \end{array} \right]$$

$$\leq \max\left\{ \frac{t_p}{|\Sigma|^{\delta}} + \frac{t_{p^{-1}}}{|\Sigma|^{\delta}},\ \frac{t_p}{|\Sigma|^{\delta}} + \frac{t_{p^{-1}}}{|\Sigma|^{\delta}},\ \frac{t_p}{|\Sigma|^{c}} + \frac{t_{p^{-1}}}{|\Sigma|^{c}} \right\} = \frac{t_p + t_{p^{-1}}}{|\Sigma|^{\min(\delta, c)}} .$$

□

Claim 7.8. *The following holds:*

$$\Pr\left[E_{\text{squeeze}}(\mathsf{tr}, (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{L_{\mathbf{V}}})) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{s}_1, \ldots, \boldsymbol{s}_{L_{\mathbf{V}}} \leftarrow \mathcal{U}(\Sigma^{r+c}) \end{array} \right] \leq (t_p + t_{p^{-1}}) \cdot \frac{L_{\mathbf{V}}}{|\Sigma|^{r+c}} .$$

Proof. The adversary $\mathcal{A}$ makes at most $t_p, t_{p^{-1}}$ queries to the oracles p, p^{-1} respectively, and has to guess any of the $L_{\mathbf{V}}$ states that have been sampled uniformly at random. In other words:

$$\Pr\left[E_{\text{squeeze}}(\mathsf{tr}, (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{L_{\mathbf{V}}})) \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{s}_1, \ldots, \boldsymbol{s}_{L_{\mathbf{V}}} \leftarrow \mathcal{U}(\Sigma^{r+c}) \end{array} \right]$$

$$\leq \Pr\left[\exists\, \iota \in [L_{\mathbf{V}} - 1] \colon (\text{`}p\text{'}, \boldsymbol{s}_{\iota}, \cdot) \in \mathsf{tr} \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{s}_1, \ldots, \boldsymbol{s}_{L_{\mathbf{V}}} \leftarrow \mathcal{U}(\Sigma^{r+c}) \end{array} \right]$$

$$\leq (t_p + t_{p^{-1}}) \left(1 - \left(1 - \frac{1}{|\Sigma|^{r+c}} \right)^{L_{\mathbf{V}}} \right)$$

$$\leq (t_p + t_{p^{-1}}) \cdot \frac{L_{\mathbf{V}}}{|\Sigma|^{r+c}} .$$

□

Hyb_2. In this hybrid we modify line 11. In each round $i \in [\mathsf{k}]$, we sample $\widehat{\boldsymbol{\rho}}_i \in \Sigma^{\ell_{\mathbf{V}}(i)}$ in a different way: sample $\widehat{\boldsymbol{\rho}}_i' \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)})$, set $\rho_i \coloneqq \psi_i(\widehat{\boldsymbol{\rho}}_i')$, and sample $\widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i)$.

Claim 7.9. *Hyb_1 and Hyb_2 are perfectly indistinguishable.*

Proof. We argue that, for every $i \in [\mathsf{k}]$, the distributions of $\widehat{\boldsymbol{\rho}}_i \in \Sigma^{\ell_{\mathbf{V}}(i)}$ in Hyb_2 and in Hyb_3 are identical. By definition of statistical distance, we have:

$$\Delta \left(\mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)}), (\psi_i^{-1} \circ \psi_i \circ \mathcal{U})(\Sigma^{\ell_{\mathbf{V}}(i)}) \right)$$

$$= \sum_{\boldsymbol{x} \in \Sigma^{\ell_{\mathbf{V}}(i)}} \left| \Pr\left[\boldsymbol{x}' = \boldsymbol{x} \;\middle|\; \boldsymbol{x}' \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)})\right] - \Pr\left[\boldsymbol{x}'' = \boldsymbol{x} \;\middle|\; \begin{array}{c} \boldsymbol{x}' \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)}) \\ y := \psi_i(\boldsymbol{x}') \\ \boldsymbol{x}'' \leftarrow \psi_i^{-1}(y) \end{array}\right] \right|$$

Let $\bar{\boldsymbol{x}} := \left\{\boldsymbol{x}' \in \Sigma^{\ell_{\mathbf{V}}(i)} : \psi_i(\boldsymbol{x}) = \psi_i(\boldsymbol{x}')\right\}$ be the set of representatives of $\boldsymbol{x}$ in $\Sigma^{\ell_{\mathbf{V}}(i)}/\psi_i$. Recall that $\psi_i^{-1}(y)$ samples uniformly at random a preimage of y. Then:

$$\Delta\left(\mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)}), (\psi_i^{-1} \circ \psi_i \circ \mathcal{U})(\Sigma^{\ell_{\mathbf{V}}(i)})\right)$$

$$= \sum_{\boldsymbol{x} \in \Sigma^{\ell_{\mathbf{V}}(i)}} \left| \frac{1}{|\Sigma|^{\ell_{\mathbf{V}}(i)}} - \Pr\left[\boldsymbol{x}' = \boldsymbol{x} \mid \boldsymbol{x}' \leftarrow \mathcal{U}(\bar{\boldsymbol{x}})\right] \cdot \Pr\left[\psi_i(\boldsymbol{x}') = \psi_i(\boldsymbol{x}) \;\middle|\; \boldsymbol{x}' \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)})\right] \right|$$

$$= \sum_{\boldsymbol{x} \in \Sigma^{\ell_{\mathbf{V}}(i)}} \left| \frac{1}{|\Sigma|^{\ell_{\mathbf{V}}(i)}} - \frac{1}{|\bar{\boldsymbol{x}}|} \cdot \Pr\left[\psi_i(\boldsymbol{x}') = \psi_i(\boldsymbol{x}) \;\middle|\; \boldsymbol{x}' \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)})\right] \right|$$

$$= \sum_{\boldsymbol{x} \in \Sigma^{\ell_{\mathbf{V}}(i)}} \left| \frac{1}{|\Sigma|^{\ell_{\mathbf{V}}(i)}} - \frac{1}{|\bar{\boldsymbol{x}}|} \cdot \frac{|\bar{\boldsymbol{x}}|}{|\Sigma|^{\ell_{\mathbf{V}}(i)}} \right| = 0\,.$$

□

Hyb_3. In this hybrid we modify line 11 (relative to Hyb_2). Specifically, we compute the encoded verifier message $\widehat{\boldsymbol{\rho}}_i$ as follows: sample a verifier message $\rho_i \leftarrow \mathcal{U}(\mathcal{M}_{\mathbf{V},i})$ and set $\widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i)$.

Claim 7.10. *The statistical distance between Hyb_3 and Hyb_2 is at most $\sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n)$.*

Proof. The IP prover uses the verifier messages $(\rho_i)_{i\in[\mathsf{k}]}$ to compute its messages $(\alpha_i)_{i\in[\mathsf{k}]}$. (The last verifier message ρ_{k} does not affect the prover messages.) Observe that:

- In Hyb_2, $(\rho_i)_{i\in[\mathsf{k}]}$ are decodings of random encoded verifier messages: $\rho_i := \psi_i(\widehat{\boldsymbol{\rho}}_i)$ for $\widehat{\boldsymbol{\rho}}_i \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)})$.
- In Hyb_3, $(\rho_i)_{i\in[\mathsf{k}]}$ are random verifier messages: $\rho_i \leftarrow \mathcal{U}(\mathcal{M}_{\mathbf{V},i})$.

The claim follows from the fact that each ψ_i is $\varepsilon_{\mathsf{cdc},i}(\lambda, n)$-biased. □

Hyb_4. In this hybrid we replace the IP prover $\mathbf{P}$ with the IP simulator $\mathbf{S}$. Instead of computing the prover messages $(\alpha_i)_{i\in[\mathsf{k}]}$ as in hybrid Hyb_4 (line 8), we compute $(\alpha_i)_{i\in[\mathsf{k}]}$ using the IP simulator $\mathbf{S}$ (line 6); the verifier messages sampled in line 11 are replaced with the simulated verifier messages from line 6. By inspection, this hybrid corresponds to the distribution $\mathcal{D}_{\mathrm{sim}}$.

Claim 7.11. *The statistical distance between Hyb_4 and Hyb_3 is at most $z_{\mathsf{IP}}(n)$.*

Proof. The statistical distance is the distance between real and simulated messages, hence is at most $z_{\mathsf{IP}}(n)$ by the definition of honest-verifier zero knowledge for IPs (Definition 7.3). □

1 : $(h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathbb{S}}(\lambda, n)$

2 : $(\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}}$

3 : $\boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^{\delta})$

4 : $\mathsf{st}'_0 \coloneqq \mathsf{DS.Start}^h(\mathbb{x})$

5 : $\mathsf{st}_0 \coloneqq \mathsf{DS.Absorb}^p(\mathsf{st}'_0, \boldsymbol{\tau})$

6 : $\left(\mathbb{x}, (\rho_i)_{i \in [\mathsf{k}]}, (\alpha_i)_{i \in [\mathsf{k}]}\right) \leftarrow \mathbf{S}(\mathbb{x})$

7 : **for** $i = 1, \ldots, \mathsf{k}$

8 : $\quad (\alpha_i, \mathsf{aux}_i) \coloneqq \mathbf{P}(\mathbb{x}, \mathbb{w})$ **if** $i = 1$ **else** $\mathbf{P}(\mathsf{aux}_{i-1}, \rho_{i-1})$

9 : $\quad \widehat{\boldsymbol{\alpha}}_i \coloneqq \varphi_i(\alpha_i)$

10 : $\quad \mathsf{st}'_i \coloneqq \mathsf{DS.Absorb}^p(\mathsf{st}_{i-1}, \widehat{\boldsymbol{\alpha}}_i)$

11 : $\quad (\widehat{\boldsymbol{\rho}}_i, \mathsf{st}'_i) \coloneqq \mathsf{DS.Squeeze}^p(\mathsf{st}'_i, \ell_{\mathbf{V}}(i)); \quad \rho_i \coloneqq \psi_i(\widehat{\boldsymbol{\rho}}_i)$

$\quad \widehat{\boldsymbol{\rho}}_i \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)});\ \rho_i \coloneqq \psi_i(\widehat{\boldsymbol{\rho}}_i);\quad (\mu_{p,i}, \mu_{p^{-1},i}, \mathsf{st}_i) \coloneqq \mathsf{ProgramBlocks}(i, \mathsf{st}'_i, \widehat{\boldsymbol{\rho}}_i)$

$\quad \widehat{\boldsymbol{\rho}}'_i \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)});\ \rho_i \coloneqq \psi_i(\widehat{\boldsymbol{\rho}}'_i);\ \widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i);\ (\mu_{p,i}, \mu_{p^{-1},i}, \mathsf{st}_i) \coloneqq \mathsf{ProgramBlocks}(i, \mathsf{st}'_i, \widehat{\boldsymbol{\rho}}_i)$

$\quad \rho_i \leftarrow \mathcal{U}(\mathcal{M}_{\mathbf{V},i});\ \widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i);\quad (\mu_{p,i}, \mu_{p^{-1},i}, \mathsf{st}_i) \coloneqq \mathsf{ProgramBlocks}(i, \mathsf{st}'_i, \widehat{\boldsymbol{\rho}}_i)$

$\quad \widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i);\quad (\mu_{p,i}, \mu_{p^{-1},i}, \mathsf{st}_i) \coloneqq \mathsf{ProgramBlocks}(i, \mathsf{st}'_i, \widehat{\boldsymbol{\rho}}_i)$

12 : $\mu_h \coloneqq (\,)$

13 : $\mu_p \coloneqq \mu_{p,1} \| \cdots \| \mu_{p,\mathsf{k}}$

14 : $\mu_{p^{-1}} \coloneqq \mu_{p^{-1},1} \| \cdots \| \mu_{p^{-1},\mathsf{k}}$

15 : $\pi \coloneqq (\boldsymbol{\tau}, (\alpha_i)_{i \in [\mathsf{k}]})$

16 : $y \leftarrow \mathcal{A}^{(h,p,p^{-1})}(\mathsf{aux}, \pi) \quad \mathcal{A}^{(h,p,p^{-1})[\mu_h, \mu_p, \mu_{p^{-1}}]}(\mathsf{aux}, \pi)$

Figure 5: Hybrid experiments Hyb_0, Hyb_1, Hyb_2, Hyb_3, Hyb_4 for Theorem 7.1.

8 Implementation

We implemented $\mathsf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ (Construction 4.3) in Rust as an open-source library called spongefish (duplex sponge Fiat–Shamir), released under a BSD-3-Clause license and available at https://github.com/arkworks-rs/spongefish.

The library is type-safe and relies on Rust's traits, using generics, associated types, and trait bounds to offer a generic implementation that behaves consistently across different permutation functions.

8.1 Core library and software stack

The core of our library, illustrated as a software stack in Figure 6, consists of the following components.

- A trait Unit defines the alphabet Σ for the permutation (and hash function). We only require that a Unit can be set to zero (via zeroize::Zeroize) and converted into bytes.
- A trait Permutation<U: Unit> defines the permutation state, of size Permutation::R (the rate) plus Permutation::C (the capacity), in units U. The trait is publicly exposed, so that one can use arbitrary permutation functions. As examples, we provide a Keccak implementation that replies on the existing keccak crate, and a Poseidon implementation that relies on the arkworks library. Moreover, hash function designers can implement their own permutation functions without having to re-implement the duplex sponge construction and the Fiat–Shamir transformation.
- A DuplexSponge<P: Permutation> implementation is given, which implements the duplex construction in overwrite mode as described in Construction 3.2, on the top on an arbitrary permutation P.

On the top of the duplex sponge implementation, we build two structures to implement the argument prover and argument verifier in Construction 4.3.

- A PrivateProverState<P: Permutation> structure, which contains the NARG prover state. At its core, this structure allows absorbing/squeezing units, and internally update the argument string. It contains: (i) the current sponge state, (ii) the NARG string serialized so far, and (iii) the private coins of the prover.
- A VerifierState<P: Permutation> structure, which implements the NARG verifier. This structure takes as input an argument string, and during the verifier execution, it deserializes the string into prover messages (as units), and feeds them into the duplex sponge to generate the verifier messages.

We exploit some of Rust's type safety features to make the library resilient against misuse. For instance, it is not possible to clone the prover state during the execution. Another example: by design, the NARG string contains only those prover messages that have been included in the Fiat–Shamir transformation.

Codecs. Via *extension traits*[12] we add the ability for prover and verifier to absorb and squeeze from arbitrary domains. In particular, a submodule codecs implements the following.

- An extension ProverMessageField<F> that provides an encoding map φ translating prover messages from a field F into units U (the permutation alphabet). Supported field types are either in ark-ff (within arkworks-rs/algebra) or ff (within zkcrypto/ff).
- An extension VerifierMessageField<F>, that provides a negligibly-biased map ψ that sends uniformly-distributed units into uniformly-distributed elements of a field F following Lemma B.1. The extension supports bytes (U = u8) and field elements (U = F). The extension is implemented for both arkworks and zkcrypto/ff.

[12] Extension traits are a programming pattern that enables adding methods to an existing type outside of the crate defining that type.

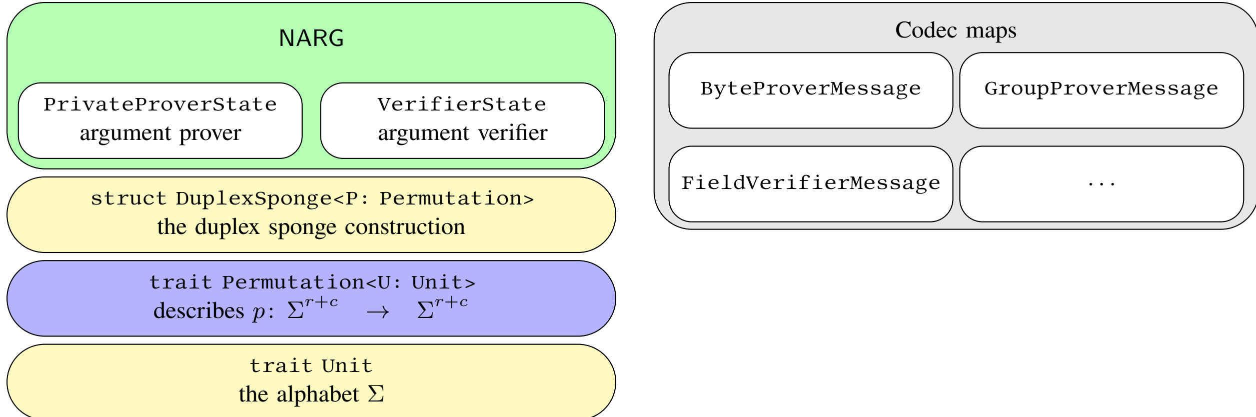

Figure 6: Overview of the software stack developed in the library.

- An extension GroupProverMessage<G> that provides an encoding map φ that compresses and serializes elliptic curve group elements into the unit.
- An extension ByteProverMessage that allows to map units (squeezed from the duplex sponge) into bytes. In the case of binary units the implementation is trivial, as it relies on squeezing field native units. In the case of algebraic permutations, we follow Lemma B.1 to make sure that the squeezed verifier messages are indistinguishable from uniformly distributed ones.

8.2 Concrete security and ergonomics

In the following, we explore some design choices and its trade-offs in security and usability.

Byte-level interface. Cryptographic libraries typically expose cryptographic objects as "opaque" byte payloads (e.g., see PKCS#11 and the NaCL cryptography library). We take a similar approach in our library, and make PrivateProverState the argument string themselves, which is exposed to the user as a byte sequence. On the other end, VerifierState reads incrementally the IP prover messages from the NARG string recovering the IP messages and re-computes the verifier messages in the correct sequence. This is beneficial for two reasons: (i) The argument string object is guaranteed to contain all messages that have been provided as input to the Fiat–Shamir transformation and to include those messages in the correct round.[13] (ii) serialization is internally made by the library, offloading the burden of delicately re-mapping verifier messages without introducing noticeable biases, in a way that can be used also by other libraries.

Customization label. It is common in Fiat–Shamir implementations to provide a label that uniquely identifies the scope where the NARG is being used. We provide a language to uniquely describe a protocol where, in addition to familiar names, the prover will describe the coded $\mathsf{cdc}(\lambda, n)$ and the hash function used. A concrete customization label example for Appendix A using Poseidon with rate 3 and capacity 1 for over BLS12-381's coordinate field is:

```
"example.com <A2S1A1>Poseidon-3-1"
```

Indicating that the proof acts over the domain example.com, the protocol will absorb 1 element, squeeze one element, and finally absorb one element, using Poseidon-3-1 as hash function.

[13] In most implementations today, the prover returns a "NARG string" that contains all prover messages, but ensuring their inclusion in the Fiat–Shamir transformation is left to the programmer.

Private randomness of the argument prover. A common pattern in signature scheme implementations is generating deterministically proofs seeding the prover randomness with the witness (e.g., see RFC6979 [Por13]). Deterministic signatures have the advantage of retaining the cryptographic security features associated with digital signatures, but can be more easily implemented in various environments, since they do not need access to a source of high-quality randomness [Por13]. The argument prover `PrivateProverState` internally holds the state of two sponges: one duplex sponge acting over a generic alphabet `U` that generates the verifier messages of the IP protocol, and a (private) duplex sponge. Both sponges absorb the (public) prover's messages, but in addition the private duplex sponge can be re-seeded with external input (e.g., the witness) and used to provide randomness that is dependent on the private information and the current IP protocol state.

A Example: codecs for Schnorr's protocol

We review Schnorr's protocol: we describe a codec for binary permutation functions (where the alphabet is $\Sigma = \{0, 1\}$), and a codec for algebraic permutation functions (where the alphabet is a field).

Let $\mathbb{G}$ be an additive elliptic curve group of prime order r where DL is hard, and let $G \in \mathbb{G}$ be a generator for $\mathbb{G}$.[14] Let $\mathbb{Z}_p$ be the coordinate field of the elliptic curve. Schnorr's protocol is a 3-message IP for the relation $\mathcal{R} := \left\{(X, x) \in \mathbb{G} \times \mathbb{Z}_p : X = xG\right\}$. In particular, $\mathsf{k} = 2$ and the second round consists of a prover message and no verifier message. To prove knowledge of $x \in \mathbb{Z}_r$ such that $X = xG$, the prover sends $K := kG$, the verifier sends a challenge $c \in \mathbb{Z}_r$, and the prover sends $s := k + cx \bmod r$. The verifier accepts if $sG = K + cX$. In our notation:

- $\mathbb{x} = X \in \mathbb{G}$ with $n := \lceil \log_2 p \rceil + 1$, the size of the "x" coordinate and the sign of the "y" coordinate;
- $\mathcal{M}_{\mathbf{P},1} = \mathbb{Z}_p^2$ is the message space of the first prover message, seen as the affine representation of an elliptic curve point;
- $\mathcal{M}_{\mathbf{V},1} = \mathbb{Z}_r$ is the message space of the verifier message;
- $\mathcal{M}_{\mathbf{P},2} = \mathbb{Z}_r$ is the message space of the second prover message.

Binary codec. An example of a binary permutation function is Keccak-$f[1600]$ [Sha], which has alphabet $\Sigma = \{0, 1\}$, capacity 512, and rate 1088. The binary codec for Schnorr's protocol is a tuple $(\ell_{\mathbf{P}}, \ell_{\mathbf{V}}, \boldsymbol{\varphi}, \boldsymbol{\psi})$ where:

- $\varphi_1 \colon \mathbb{Z}_p^2 \to \{0, 1\}^{\ell_{\mathbf{P}}(1)}$, with $\ell_{\mathbf{P}}(1) := \lceil \log p \rceil + 1$, is the big-endian binary encoding of the "x" coordinate and the sign of the "y" coordinate. Note that φ_1 is injective, and its preimage is efficiently computable using the elliptic curve's equation.
- $\psi_1 \colon \{0, 1\}^{\ell_{\mathbf{V}}(1)} \to \mathbb{Z}_r$, with $\ell_{\mathbf{V}}(1) = \lceil \log r \rceil + \lambda$, is the big-endian decoding of the given binary string interpreted as an integer modulo r (i.e., $\boldsymbol{b} \in \{0, 1\}^{\ell_{\mathbf{V}}(1)}$ is mapped to $\sum_{i=1}^{\ell_{\mathbf{V}}(1)} b_i 2^{i-1} \bmod r$). The additive term λ ensures that ψ_1 has bias at most $2^{-\lambda}$ (see Lemma B.1).
- $\varphi_2 \colon \mathbb{Z}_r \to \{0, 1\}^{\ell_{\mathbf{P}}(2)}$, with $\ell_{\mathbf{P}}(2) := \lceil \log r \rceil$ is the big-endian binary encoding of the second prover message.

Algebraic codec. Algebraic permutation functions typically have a larger alphabet. An example is Poseidon [GKRRS21], whose alphabet can be set to be the field $\mathbb{Z}_p$ (the elliptic curve's field of definition), with capacity 1 and rate 2. The algebraic codec for Schnorr's protocol is a tuple $(\ell_{\mathbf{P}}, \ell_{\mathbf{V}}, \boldsymbol{\varphi}, \boldsymbol{\psi})$ where:

- $\varphi_1 \colon \mathbb{Z}_p^2 \to \mathbb{Z}_p^2$ (with $\ell_{\mathbf{P}}(1) = 2$) is the identity function.
- $\psi_1 \colon \mathbb{Z}_p \to \mathbb{Z}_r$ (with $\ell_{\mathbf{V}}(1) = 1$) maps x to $x \bmod r$. The bias of this map is $2\frac{p \bmod r}{pr}(p - (p \mod r))$, by Lemma B.1. If we consider a pairing-friendly elliptic curve such as BLS12-381, the bias is at most 2^{-126}.
- $\varphi_2 \colon \mathbb{Z}_r \to \{0, 1\}^{\ell_{\mathbf{P}}(2)}$, with $\ell_{\mathbf{P}}(2) = \lceil \log r / \log p \rceil$, is the big-endian binary encoding of the response. In typical pairing-friendly elliptic curves, such as BLS12-381, this map is the "identity" function, mapping an element of $\mathbb{Z}_r$, seen as an integer $[0, r-1]$, into $\mathbb{Z}_q$.

A common concern, when using algebraic hashes, is the number of invocations of the permutation function. In this example, the permutation function is invoked only once: the construction DSFS absorbs two $\mathbb{Z}_p$ elements, writing two elements in the rate (see Item 3a in Construction 3.2). At the end of the execution, the state has i_{A} and i_{S} equal to the rate of the sponge (and no call yet to the permutation function). Then, DSFS squeezes one $\mathbb{Z}_p$ element from the duplex sponge, which invokes the permutation function, sets $i_{\mathrm{S}} = 0$, and then reads the first element of the rate segment in the permutation state.

[14] The group choice is merely an example, and captures, for instance, the elliptic curves in the SEC2 standards [Bro10]. Other choices are possible (e.g., the subgroup of squares in $\mathbb{Z}_q^*$, where q is a safe prime).

B Bias of modular reduction

We state and prove a simple lemma about the bias of modular reduction, which is useful for bounding the bias of the distribution that arises from a common decoding strategy from binary strings to prime field elements.

Consider the setting where the verifier message is a random field element in a prime field $\mathbb{F}_p$, equivalently, a random integer in $[0, p{-}1]$. Moreover, suppose that the function (or permutation) used in the Fiat–Shamir transformation is binary, which means that one must somehow decode a field element from a (random) binary string x in $\{0,1\}^m$, for a sufficiently large m.

A common decoding strategy [Hao] is to interpret x as a base-2 integer and outputting its remainder modulo p, i.e., the decoding function $\psi \colon \{0,1\}^m \to [0, p{-}1]$ is

$$\psi(x) := \left(\sum_{i=1}^{m} x_i 2^{i-1} \right) \bmod p \,.$$

The lemma below directly implies that, for $m := \lceil \log p \rceil + \lambda$, the bias of ψ is at most $2^{-\lambda}$. The extra length λ ensures that $\mathcal{U}(\mathbb{F}_p) = \mathcal{U}([0, p{-}1])$ and $\psi(\mathcal{U}(\{0,1\}^m))$ are close enough. More generally, for positive integers a, b with $b \geq a$, the lemma below upper bounds the statistical distance between $\mathcal{U}([0, a{-}1])$ (the target distribution) and the distribution arising from $\mathcal{U}([0, b{-}1])$ reduced modulo a.

Lemma B.1. *Let a, b be positive integers with $b \geq a$, and set $r := b \pmod a$. Let $\psi_{a,b} \colon [0, b{-}1] \to [0, a{-}1]$ be the function that maps x to $x \bmod a$. Then*

$$\Delta\left(\mathcal{U}([0, a{-}1]), \psi_{a,b}(\mathcal{U}([0, b{-}1]))\right) \leq \frac{2r}{ab}(a - r) \,.$$

In particular:

- *if $a \mid b$ then the statistical distance is 0, and*
- *if $\lceil \log_2 b \rceil \geq \lceil \log_2 a \rceil + \lambda$ then the statistical distance is at most $2^{-\lambda}$.*

Proof. Let $b = q \cdot a + r$ with $0 \leq r < a$. For every $y \in [0, a{-}1]$, the probability that $\psi_{a,b}(\mathcal{U}([0, b{-}1]))$ is equal to y is $\frac{q+1}{b}$ if $0 \leq y < r$, and $\frac{q}{b}$ if $r \leq y < a$. Therefore:

$$\begin{aligned}
&\Delta\left(\mathcal{U}([0, a{-}1]), \psi_{a,b}(\mathcal{U}([0, b{-}1]))\right) \\
&= \sum_{k \in [0, r-1]} \left| \frac{1}{a} - \frac{q+1}{b} \right| + \sum_{k \in [r, a-1]} \left| \frac{1}{a} - \frac{q}{b} \right| \\
&= r \left| \frac{b - a(q+1)}{ab} \right| + (a - r) \left| \frac{b - qa}{ab} \right| \\
&= \frac{1}{ab} \cdot \left(r|b - qa - a| + (a - r)|r| \right) \\
&= \frac{2r}{ab} \left(a - r \right) \,.
\end{aligned}$$

The case where $a \mid b$ is straightforward since it implies $r = 0$. The case where $\lceil \log_2 b \rceil \geq \lceil \log_2 a \rceil + \lambda$ follows the fact that $r(a - r) \leq a^2/2$:

$$\frac{2r}{ab}(a - r) \leq \frac{a^2}{a^2 2^\lambda} \leq \frac{1}{2^\lambda} \,.$$

□

Acknowledgments

This work was partially supported by the Ethereum Foundation.

References

[AFK22] T. Attema, S. Fehr, and M. Klooß. “Fiat-Shamir Transformation of Multi-round Interactive Proofs”. In: *Proceedings of the 20th International Conference on Theory of Cryptography*. Vol. 13747. TCC ’22. 2022, pp. 113–142. DOI: 10.1007/978-3-031-22318-1_5.

[AGRRT16] M. R. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. “MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity”. In: *Proceedings of the 22nd International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT ’16. 2016, pp. 191–219. DOI: 10.1007/978-3-662-53887-6_7.

[Ark] *A library of useful cryptographic primitives*. https://github.com/arkworks-rs/crypto-primitives.

[ark] arkworks. *arkworks: an ecosystem for developing and programming with zkSNARKs*. URL: https://github.com/arkworks-rs.

[AY25] G. Arnon and E. Yogev. *Towards a White-Box Secure Fiat-Shamir Transformation*. Cryptology ePrint Archive, Paper 2025/329. 2025. URL: https://eprint.iacr.org/2025/329.

[Azt] Aztec. *Aztec monorepo*. https://github.com/AztecProtocol/aztec-packages/.

[Bar01] B. Barak. “How to Go Beyond the Black-Box Simulation Barrier”. In: *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*. FOCS ’01. 2001, pp. 106–115.

[BBHMR19] J. Bartusek, L. Bronfman, J. Holmgren, F. Ma, and R. D. Rothblum. “On the (In)security of Kilian-Based SNARGs”. In: *Proceedings of the 14th Theory of Cryptography Conference*. TCC ’19. 2019, pp. 522–551. ISBN: 978-3-030-36033-7.

[BCS16] E. Ben-Sasson, A. Chiesa, and N. Spooner. “Interactive Oracle Proofs”. In: *Proceedings of the 14th Theory of Cryptography Conference*. TCC ’16. 2016, pp. 31–60.

[BDHPVAVK18] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer. “Farfalle: parallel permutation-based cryptography”. In: FSE ’18 (2018).

[BDPV08] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. “On the Indifferentiability of the Sponge Construction”. In: *Proceedings of the 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’08. 2008, pp. 181–197. DOI: 10.1007/978-3-540-78967-3_11.

[BDPVA12] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. “Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications”. In: *Selected Areas in Cryptography*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 320–337. ISBN: 978-3-642-28496-0.

[BKM20] Z. Brakerski, V. Koppula, and T. Mour. “NIZK from LPN and Trapdoor Hash via Correlation Intractability for Approximable Relations”. In: *Proceedings of the 40th Annual International Cryptology Conference*. CRYPTO ’20. 2020, pp. 738–767. DOI: 10.1007/978-3-030-56877-1_26.

[BLHG] S. Bowe, Y. T. Lai, D. E. Hopwood, and J. Grigg. *The Halo2 zero-knowledge proving system*. https://github.com/zcash/halo2.

[Bou+23] C. Bouvier et al. “New Design Techniques for Efficient Arithmetization-Oriented Hash Functions: Anemoi Permutations and Jive Compression Mode”. In: *Proceedings of the 43th Annual International Cryptology Conference*. CRYPTO ’23. 2023. ISBN: 978-3-031-38547-6. DOI: 10.1007/978-3-031-38548-3_17.

[Bro10] D. L. R. Brown. *Standards for Efficient Cryptography — SEC 2: Recommended Elliptic Curve Domain Parameters*. Ed. by C. Research. https://www.secg.org/sec2-v2.pdf. Version 2.0. 2010.

[Can+19] R. Canetti et al. "Fiat-Shamir: from practice to theory". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. STOC '19. 2019, pp. 1082–1090. DOI: 10.1145/3313276.3316380.

[CCHLRR18] R. Canetti, Y. Chen, J. Holmgren, A. Lombardi, G. N. Rothblum, and R. D. Rothblum. "Fiat-Shamir From Simpler Assumptions". In: *IACR Cryptol. ePrint Arch.* (2018), p. 1004. URL: https://eprint.iacr.org/2018/1004.

[CCR16] R. Canetti, Y. Chen, and L. Reyzin. "On the Correlation Intractability of Obfuscated Pseudorandom Functions". In: *Proceedings of the 14th Theory of Cryptography Conference*. TCC '16. 2016, pp. 389–415. DOI: 10.1007/978-3-662-49096-9_17.

[CCRR18] R. Canetti, Y. Chen, L. Reyzin, and R. D. Rothblum. "Fiat-Shamir and Correlation Intractability from Strong KDM-Secure Encryption". In: *Proceedings of the 37th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT '18. 2018, pp. 91–122. DOI: 10.1007/978-3-319-78381-9_4.

[CDGLN18] J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. "The Wonderful World of Global Random Oracles". In: *Proceedings of the 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT'18. 2018, pp. 280–312. DOI: 10.1007/978-3-319-78381-9_11.

[CF24] A. Chiesa and G. Fenzi. "zkSNARKs in the ROM with Unconditional UC-Security". In: *Proceedings of the 22nd International Theory of Cryptography Conference*. TCC '24. 2024, pp. 67–89. DOI: 10.1007/978-3-031-78011-0_3.

[CGH04] R. Canetti, O. Goldreich, and S. Halevi. "The random oracle methodology, revisited". In: *Journal of the ACM* 51.4 (2004), pp. 557–594.

[CGJJZ23] A. R. Choudhuri, S. Garg, A. Jain, Z. Jin, and J. Zhang. "Correlation Intractability an SNARGs from Sub-exponential DDH". In: *Proceedings of the 43rd Annual International Cryptology Conference*. CRYPTO '23. 2023, pp. 635–668. DOI: 10.1007/978-3-031-38551-3_20.

[CJS14] R. Canetti, A. Jain, and A. Scafuro. "Practical UC security with a Global Random Oracle". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. 2014, pp. 597–608. DOI: 10.1145/2660267.2660374.

[CKGW] D. Connolly, C. Komlo, I. Goldberg, and C. A. Wood. *The Flexible Round-Optimized Schnorr Threshold (FROST) Protocol for Two-Round Schnorr Signatures*. RFC 9591. DOI: 10.17487/RFC9591. URL: https://www.rfc-editor.org/info/rfc9591.

[CMS19] A. Chiesa, P. Manohar, and N. Spooner. "Succinct Arguments in the Quantum Random Oracle Model". In: *17th International Theory of Cryptography Conference*. TCC 2019. 2019, pp. 1–29. DOI: 10.1007/978-3-030-36033-7_1.

[CY24] A. Chiesa and E. Yogev. *Building Cryptographic Proofs from Hash Functions*. 2024. URL: https://snargsbook.org/.

[DFHSW] A. Davidson, A. Faz-Hernandez, N. Sullivan, and C. A. Wood. *Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups*. RFC 9497. DOI: 10.17487/RFC9497. URL: https://www.rfc-editor.org/info/rfc9497.

[DFMS19] J. Don, S. Fehr, C. Majenz, and C. Schaffner. "Security of the Fiat-Shamir Transformation in the Quantum Random-Oracle Model". In: *Proceedings of the 39th Annual International Cryptology Conference*. CRYPTO '19. Springer, 2019, pp. 356–383. DOI: 10.1007/978-3-030-26951-7_13.

[FS86] A. Fiat and A. Shamir. "How to prove yourself: practical solutions to identification and signature problems". In: *Proceedings of the 6th Annual International Cryptology Conference*. CRYPTO '86. 1986, pp. 186–194.

[GK03] S. Goldwasser and Y. T. Kalai. "On the (In)security of the Fiat-Shamir Paradigm". In: *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '03. 2003, pp. 102–113.

[GKRRS21] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems". In: *30th USENIX Security Symposium*. USENIX Security '21. 2021, pp. 519–535.

[Ham17] M. Hamburg. "The STROBE protocol framework". In: *IACR Cryptol. ePrint Arch.* (2017), p. 3. URL: http://eprint.iacr.org/2017/003.

[Hao] F. Hao. *Schnorr Non-interactive Zero-Knowledge Proof*. RFC 8235. DOI: 10.17487/RFC8235. URL: https://www.rfc-editor.org/info/rfc8235.

[HL18] J. Holmgren and A. Lombardi. "Cryptographic Hashing from Strong One-Way Functions (Or: One-Way Product Functions and Their Applications)". In: *Proceedings of the 59th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '18. 2018, pp. 850–858. DOI: 10.1109/FOCS.2018.00085.

[HLR21] J. Holmgren, A. Lombardi, and R. D. Rothblum. "Fiat–Shamir via list-recoverable codes (or: parallel repetition of GMW is not zero-knowledge)". In: *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. STOC '21. 2021, pp. 750–760. DOI: 10.1145/3406325.3451116.

[JJ21] A. Jain and Z. Jin. "Non-interactive Zero Knowledge from Sub-exponential DDH". In: *Proceedings of the 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '21. 2021, pp. 3–32. DOI: 10.1007/978-3-030-77870-5_1.

[JKKZ21] R. Jawale, Y. T. Kalai, D. Khurana, and R. Zhang. "SNARGs for bounded depth computations and PPAD hardness from sub-exponential LWE". In: STOC '21. 2021, pp. 708–721. DOI: 10.1145/3406325.3451055.

[KLV23] Y. T. Kalai, A. Lombardi, and V. Vaikuntanathan. "SNARGs and PPAD Hardness from the Decisional Diffie-Hellman Assumption". In: *Proceedings of the 41th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '23. 2023, pp. 470–498. DOI: 10.1007/978-3-031-30617-4_16.

[KRR17] Y. T. Kalai, G. N. Rothblum, and R. D. Rothblum. "From Obfuscation to the Security of Fiat-Shamir for Proofs". In: *Proceedings of the 37th Annual International Conference on Advances in Cryptology*. CRYPTO '17. 2017, pp. 224–251. ISBN: 978-3-319-63715-0.

[KRS25] D. Khovratovich, R. D. Rothblum, and L. Soukhanov. *How to Prove False Statements: Practical Attacks on Fiat-Shamir*. Cryptology ePrint Archive, Paper 2025/118. 2025. URL: https://eprint.iacr.org/2025/118.

[LKWL] T. Looker, V. Kalos, A. Whitehead, and M. Lodder. *The BBS Signature Scheme*. Internet-Draft draft-irtf-cfrg-bbs-signatures-07. Work in Progress. Internet Engineering Task Force. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-bbs-signatures/07/.

[LZ19] Q. Liu and M. Zhandry. "Revisiting Post-quantum Fiat-Shamir". In: *Proceedings of the 39th Annual International Cryptology Conference*. CRYPTO '19. 2019, pp. 326–355. DOI: 10.1007/978-3-030-26951-7_12.

[MF21] A. Mittelbach and M. Fischlin. *The Theory of Hash Functions and Random Oracles - An Approach to Modern Cryptography*. Information Security and Cryptography. 2021. ISBN: 978-3-030-63286-1. DOI: 10.1007/978-3-030-63287-8.

[MRH04] U. M. Maurer, R. Renner, and C. Holenstein. “Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology”. In: *Proceedings of the first Theory of Cryptography Conference*. TCC ’04. 2004, pp. 21–39. DOI: 10.1007/978-3-540-24638-1_2.

[Pas03] R. Pass. “On Deniability in the Common Reference String and Random Oracle Model”. In: *Proceedings of the 23rd Annual International Cryptology Conference*. CRYPTO ’03. 2003, pp. 316–337.

[Por13] T. Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979. 2013. DOI: 10.17487/RFC6979. URL: https://www.rfc-editor.org/info/rfc6979.

[PS19] C. Peikert and S. Shiehian. “Noninteractive Zero Knowledge for NP from (Plain) Learning with Errors”. In: *Proceedings of the 39th Annual International Cryptology Conference*. CRYPTO ’19. 2019, pp. 89–114. DOI: 10.1007/978-3-030-26948-7_4.

[Set] S. Setty. *Nova: High-speed recursive arguments from folding schemes*. https://github.com/microsoft/Nova/.

[Sha] *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. National Institute of Standards and Technology, NIST FIPS PUB 202, U.S. Department of Commerce. 2015.

[Sig] Signal Foundation. *libsignal’s proof of knowledge stateful hash object*. https://github.com/signalapp/libsignal/tree/main/rust/poksho/.

[Sta] StarkWare. *Cairo*. https://github.com/starkware-libs/cairo-lang/.

[Val] H. de Valence. *Merlin: Composable proof transcripts for public-coin arguments of knowledge*. https://github.com/dalek-cryptography/merlin. Version 1.0.

[Wee09] H. Wee. “Zero Knowledge in the Random Oracle Model, Revisited”. In: *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT ’09. Springer Berlin Heidelberg, 2009, pp. 417–434. DOI: 10.1007/978-3-642-10366-7_25.

[Wri] O. Wright. *Decree Fiat Shamir Library*. https://github.com/trailofbits/decree. 0.1.0.

[YZ21] T. Yamakawa and M. Zhandry. “Classical vs Quantum Random Oracles”. In: *Proceedings of the 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT’21. 2021, pp. 568–597. DOI: 10.1007/978-3-030-77886-6_20.