A Fiat–Shamir Transformation From Duplex Sponges

Alessandro Chiesa
alessandro.chiesa@epfl.ch
EPFL

Michele Orrù
m@orru.net
CNRS

July 9, 2025

Abstract

We analyze a variant of the Fiat–Shamir transformation based on an ideal permutation. The transformation relies on the popular duplex sponge paradigm, and minimizes the number of calls to the permutation (given the information to absorb/squeeze). This closely models deployed variants of the Fiat–Shamir transformation, and our analysis provides concrete security bounds to guide security parameters in practice. Our results contribute to the ongoing community-wide effort to achieve rigorous, and ultimately formally verified, security proofs for deployed cryptographic proofs. Along the way, we find that indifferentiability (a property proven for many modes of operation, including the duplex sponge) is ill-suited for establishing the knowledge soundness and zero knowledge of a non-interactive argument.

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

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 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 security analyses do 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 *support input/output blocks of fixed length*, so practitioners consider (more complex) variants that support sufficiently large inputs/outputs via multiple calls to the fixed-size hash function. These variants follow "modes of operation" that in some cases are known to be *indifferentiable* [MRH04] from the random oracle they emulate. However, indifferentiability is inadequate to establish knowledge soundness and zero knowledge (we discuss in detail why). This 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), as well as the ongoing community-wide effort to work towards formally-verifier security analyses of cryptographic proofs [Eth24].

Permutation-based cryptography. Permutation-based cryptography has developed into a viable alternative to traditional block-cipher-based cryptography, leading to elegant constructions 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 sponge construction 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 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.

The 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, the transformation we consider 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 prior transformation variants (some are deployed). 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; (iii) the SAFE framework [AKMQ23; KMM23; dus]; and (iv) 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]. Notably, $\mathbf{DSFS}$ avoids the use of padding or IO patterns, leading to greater simplicity and efficiency.

Our main contribution is a rigorous security analysis of $\mathbf{DSFS}$, which we discuss next. See Section 2.3 for why indifferentiability of the underlying duplex sponge construction does not suffice for our purposes.

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

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

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]).

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 $\frac{25t^2}{|\Sigma|^c}$ 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*

$$\varepsilon_{\mathsf{NARG}}(t) \le \varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(t) + \frac{25t^2}{|\Sigma|^c} \,.$$

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*

$$\kappa_{\mathsf{NARG}}(t) \le \kappa^{\mathrm{sr}}_{\mathsf{IP}}(t) + \frac{25t^2}{|\Sigma|^c} \,.$$

We outline the ideas underlying the theorem in Section 2.4, and formally state and prove it in Section 6.

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 2.5, 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 $\ell_{\mathsf{V}}(i) = 100$ and $\mathcal{M}_{\mathsf{V},i}$ is a 100-bit prime field then every map from $\Sigma^{\ell_{\mathsf{V}}(i)}$ to $\mathcal{M}_{\mathsf{V},i}$ introduces some bias (i.e., does not map the uniform distribution on $\Sigma^{\ell_{\mathsf{V}}(i)}$ to the uniform distribution on $\mathcal{M}_{\mathsf{V},i}$).

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

$$t \cdot \max_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i} + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}$$

appears as an additional additive error in Theorem 1, and that

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

appears as an additional additive error in 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 **DSFS** in Rust, and make it open-source under a BSD license. Our library, called `spongefish` (duplex sponge Fiat–Shamir), is available at 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 C.1). This avoids manual (and error-prone) conversions that are typically seen in the wild.

We elaborate on our library in Section 8. (Our library does not currently 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).

DSFS 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 2.6.

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: the 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.

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. There are differing variants of the Fiat–Shamir transformation 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. The SAFE API works on any field element and is used by Dusk, a privacy-oriented blockchain protocol [dus]. 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 describe DSFS. In Section 2.4 we outline the proof of Theorem 1 (soundness and knowledge soundness) and Section 2.5 we outline the proof of Theorem 2 (zero knowledge). Finally, in Section 2.6 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 2.5). 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 IP verifier messages $(\rho_i \in \mathcal{M}_{\mathbf{V},i})_{i\in[\mathsf{k}]}$ via queries to the oracles $\boldsymbol{f}$. (In fact, $\mathcal{P}$ does not need to derive ρ_{k}.)
- *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 2.5.

(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, relative to $\mathbf{FS}[\mathsf{IP}]$, 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 $\mathbf{FS}[\mathsf{IP}]$, 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 $\mathbf{HCFS}[\mathsf{IP}]$, 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 support sufficiently large inputs and outputs via multiple calls to a hash function with fixed-size inputs/outputs. 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 $\mathbf{DSFS}[\mathsf{IP},\mathsf{cdc}]$ 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 (without affecting IP's security).

[4] Specifically, we consider the duplex sponge *in overwrite mode* rather than canonical duplex sponges "in XOR mode" as it is simpler. Our analysis can be adapted for related constructions such as XOR mode, and different padding functions.

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

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/labels of the prover and verifier messages (our results guarantee this is secure). Moreover, differently from **HCFS**, **DSFS** uses only a single oracle and, after squeezing a verifier message, r symbols can be 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 verifier $\mathcal{V}$ initialize the sponge state as $\mathsf{st}_0 := \mathsf{DS.Start}^p(\mathbb{x})$, 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 which may favor alternative methods to hash $\mathbb{x}$, rather than $\mathsf{DS.Absorb}^p(\mathsf{st}_*, \varphi_0(\mathbb{x}))$.[8] This choice is convenient in practice, and appears in implementations [Ark] and the literature [KMM23, Fig. 2].

[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.

[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.

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 Limitations of indifferentiability

Indifferentiability is a relaxation of indistinguishability that facilitates security reductions [MRH04], and is a popular security notion proved for modes of operations for hash functions. Several variants of the sponge construction are known to be indifferentiable from a random oracle [BDPV08; BDPVA12; KMM23; AB24], including the flavor of the duplex sponge construction used in DS. Roughly, in our setting, such a guarantee means that there exists a probabilistic stateful algorithm $\mathcal{S}_{\mathsf{DS}}$ such that, for every t-query distinguisher D,

$$\varepsilon_{\text{ind}}(t) := \left| \Pr\left[D^{\mathsf{DS}^p, p, p^{-1}} = 1\right] - \Pr\left[D^{(f_i)_{i\in[\mathsf{k}]}, \mathcal{S}_{\mathsf{DS}}^{(f_i)_{i\in[\mathsf{k}]}}} = 1\right] \right| = O\left(\frac{t^2}{|\Sigma|^c}\right) ,$$

where $(f_i)_{i\in[\mathsf{k}]}$ are the random oracles for the basic variant **FS** (the oracles we wish to emulate).

However, while one can use indifferentiability to prove that the soundness errors of **FS** and **DSFS** differ by at most $\varepsilon_{\text{ind}}(t)$ (as one may expect), we see no way to prove analogous statements for the knowledge soundness errors or zero knowledge errors of **FS** and **DSFS**. In other words:

Indifferentiability does not suffice to establish either knowledge soundness or zero knowledge.

Below we give high-level intuition for why this is the case, and in Appendix A we discuss indifferentiability in detail (since the fact that indifferentiability does not suffice for our results may come as a surprise to some).

- *Knowledge soundness.* The knowledge soundness property of a non-interactive argument in an oracle model involves the malicious prover's query-answer trace, as the knowledge extractor crucially uses it to deduce a witness (see Definition 3.6). Indifferentiability ensures that an adversary cannot tell the difference between query access to the duplex sponge DS^p and oracles $(f_i)_{i\in[\mathsf{k}]}$ (while given access, respectively, to (p, p^{-1}) and a stateful emulation of these via $\mathcal{S}_{\mathsf{DS}}^{(f_i)_{i\in[\mathsf{k}]}}$). But, in order to design a knowledge extractor for **DSFS** based on one for **FS**, we would also need a way to translate query-answer traces relative to p, p^{-1} into corresponding query-answer traces relative to $(f_i)_{i\in[\mathsf{k}]}$. *Indifferentiability provides no such capabilities.*

  In Section 2.4 we discuss how, in order to establish Theorem 1, we prove a lemma that can be viewed as a qualitatively stronger variant of indifferentiability. In Appendix A.2.2 we make our indifferentiability notion explicit, and show how it can generically transfer knowledge soundness from one non-interactive argument to another. While **DSFS**'s security may appear evident, proving our lemma demands considerable effort.

- *Zero knowledge.* The zero knowledge property of a non-interactive argument in an oracle model requires programming the oracle (see Definition 7.4). However, *indifferentiability offers no guarantees in the presence of programming*. Separately, even if indifferentiability could somehow be used (say, by enriching it with a programming interface), it would only achieve an additive loss of $\varepsilon_{\text{ind}}(t) = O(\frac{t^2}{|\Sigma|^c})$, whereas Theorem 2 achieves an additive loss of $O(\frac{t}{|\Sigma|^c})$.

  In Section 2.5 we discuss how we establish Theorem 2 via a direct analysis. In Appendix A.2.3 we discuss in more detail how indifferentiability is not the right notion to transfer zero knowledge.

2.4 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 verifier of $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$, and $(\mathcal{P}_{\mathsf{std}}, \mathcal{V}_{\mathsf{std}})$ be the argument prover and 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 $\mathcal{D}_{\mathsf{IP}}$). We must consider adversaries that have access to the inverse of p because, in practice, permutations are 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 $\mathbf{FS}[\mathsf{IP}]$.

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:*

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

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 $\mathsf{DSFS}[\mathsf{IP}, \mathsf{cdc}]$.

Establishing the key lemma. The technical core of our security reduction is upper bounding the statistical distance between the two distributions in Equation 1: we prove that if $\tilde{\mathcal{P}}$ makes at most t queries then the statistical distance is at most

$$\eta_\star(t) := \frac{25t^2}{|\Sigma|^c} + t \cdot \max_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i} + \sum_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i} .$$

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 $(g_i)_{i\in[\mathsf{k}]}$. 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 bookkeeping: 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 it finds a collision in a capacity segment, backtracking might not find a single sequence of rate segments. We carefully prove that this event happens with probability at most $\frac{25t^2}{|\Sigma|^c}$.

  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 blocks of length r, and builds a sequence of queries and answers that mimic the duplex sponge in order 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 measure the probability of this event and show that it does not contribute to the overall error of the reduction, for query bounds $t > \sum_{i\in[\mathsf{k}]} \lceil \ell_{\mathbf{V}}(i)/r \rceil$.

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

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

  incurring a statistical distance that is at most $t \cdot \max_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i} + \sum_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}$.

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 $\mathcal{M}_{\mathbf{V},i}$. We sketch why, for every $i \in [\mathsf{k}]$, 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 φ_i^{-1} ultimately affects the running time of the security reduction, due to the need to compute φ_i^{-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 $\varepsilon_{\mathsf{cdc}_i}$.

The malicious argument prover $\tilde{\mathcal{P}}$ makes at most t queries, each incurring a bias of up to the maximum $\max_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}$; in addition, the argument verifier $\mathcal{V}$ makes the equivalent of one query for each f_i, incurring the bias $\sum_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}$.

Throughout, the inverse permutation p^{-1} does not help the adversary; intuitively, this follows from the PRP–PRF switching lemma [BR06], though in our analysis need a slightly stronger statement to account for different query bounds for different oracles. The final bound for $\eta_\star(t)$ follows by adding the individual errors.

2.5 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] analyzes the variants $\mathsf{FS}[\mathsf{IP}]$ and $\mathsf{HCFS}[\mathsf{IP}]$ 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 $\mathsf{DSFS}[\mathsf{IP},\mathsf{cdc}]$ 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 $\mathsf{DSFS}[\mathsf{IP},\mathsf{cdc}]$, 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 $\mathsf{st}_0 := \mathsf{Absorb}^{p}(\mathsf{st}'_0,\boldsymbol{\tau})$; 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 bound the statistical distance between the real-world distribution and simulated-world distribution, by taking into account the effect of the (single) salt and the biases incurred by the decoding maps. The upper bound we prove is

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

We split our analysis in intermediate hybrids, isolating different distinguishing advantages.

- First, we show that, after absorbing $\boldsymbol{\tau}\leftarrow\mathcal{U}(\Sigma^{\delta})$, the sponge state is unpredictable: if $\delta>c$ then the probability that the adversary guesses the capacity segment resulting after absorbing $\lceil\frac{\delta}{r}\rceil$ blocks is at most $\frac{t}{|\Sigma|^{c}}$; else if $\delta\le c$ then this probability is (trivially) at most $\frac{t}{|\Sigma|^{\delta}}$.
- Then, we show that the $\sum_{i\in[\mathsf{k}]}\lceil\ell_{\mathbf{V}}(i)/r\rceil$ permutation states associated to the verifier messages are unpredictable. The simulator re-programs these, and the probability that a t-bound adversary queries any one of these is at most $\frac{t}{|\Sigma|^{r+c}}$.
- Finally, we consider the biases arising from decoding maps to the k verifier messages. This involves, for each $i\in[\mathsf{k}]$, 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 is 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 2.2 (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].

2.6 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]):

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

We can replace the second 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:

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

where $\mathsf{cdc} = \big((\varphi_i)_{i\in[\mathsf{k}]}, (\psi_i)_{i\in[\mathsf{k}]}\big)$ is an appropriate codec for $\mathbf{iBCS}[\mathsf{IOP}]$, i.e., for encoding Merkle commitments (rather than IOP prover messages) and decoding IOP verifier messages for IOP.[9]

Upper bounds on the soundness and knowledge soundness follow from Theorem 1, replacing the error $\frac{t^2}{2^\lambda}$ of $\mathbf{HCFS}$ with that of $\mathbf{DSFS}$; the multi-extraction error κ_{MT} that arises in $\mathbf{iBCS}$ remains. Similarly, an upper bound on zero knowledge follows from Theorem 2; the hiding error z_{MT} that arises in $\mathbf{iBCS}$ remains. Below we highlight in blue the contributions from $\mathbf{DSFS}$ that arise from our results.

Corollary 1 (informal)**.** *Let* IOP *be a public-coin IOP and let* $\mathsf{NARG} := \mathbf{DSFS}[\mathbf{iBCS}[\mathsf{IOP}], \mathsf{cdc}]$.

- *If* IOP *has state-restoration soundness error* $\varepsilon^{\mathrm{sr}}_{\mathrm{IOP}}$ *then* NARG *has soundness error* $\varepsilon_{\mathsf{NARG}}$ *such that*

$$\varepsilon_{\mathsf{NARG}}(t) \le \varepsilon^{\mathrm{sr}}_{\mathrm{IOP}}(t) + \kappa_{\mathrm{MT}}(t, \mathsf{l}, (t+1)\cdot\mathsf{k}, \mathsf{k}) + \frac{25t^2}{|\Sigma|^c} + t\cdot\max_{i\in[\mathsf{k}]}\varepsilon_{\mathsf{cdc},i} + \sum_{i\in[\mathsf{k}]}\varepsilon_{\mathsf{cdc},i} \,.$$

- *If* IOP *has state-restoration knowledge soundness error* $\kappa^{\mathrm{sr}}_{\mathrm{IOP}}$ *then* NARG *has knowledge soundness error* κ_{NARG} *such that*

$$\kappa_{\mathsf{NARG}}(t) \le \kappa^{\mathrm{sr}}_{\mathrm{IOP}}(t) + \kappa_{\mathrm{MT}}(t, \mathsf{l}, (t+1)\cdot\mathsf{k}, \mathsf{k}) + \frac{25t^2}{|\Sigma|^c} + t\cdot\max_{i\in[\mathsf{k}]}\varepsilon_{\mathsf{cdc},i} + \sum_{i\in[\mathsf{k}]}\varepsilon_{\mathsf{cdc},i} \,.$$

- *If* IOP *has honest-verifier zero-knowledge error* z_{IOP} *then* NARG *has adaptive zero-knowledge error* z_{NARG} *such that*

$$z_{\mathsf{NARG}}(t) \le z_{\mathrm{IOP}} + z_{\mathrm{MT}}(\mathsf{l}, \delta, \mathsf{q}, t) + \frac{t}{|\Sigma|^{\min\{\delta,c\}}} + \frac{t\cdot\sum_{i\in[\mathsf{k}]}\lceil \ell_{\mathbf{V}}(i)/r\rceil}{|\Sigma|^{r+c}} + \max_{i\in[\mathsf{k}]}\varepsilon_{\mathsf{cdc},i} \,.$$

[9] In $\mathbf{iBCS}[\mathsf{IOP}]$ the prover's final message in round $\mathsf{k}+1$, which contains answers and openings, is not absorbed, so there are no corresponding codec functions for this round.

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 $\{1, \dots, n\}$. 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. With $\mathrm{Im}(f)$ we denote the image of the function 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 $\mathsf{tr}[x] := y$ 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 total order. This is only needed in order to perform dichotomic search in logarithmic time over a list of Σ-tuples (see Sections 5.2 and 5.3).

We conclude with a simple lemma that we use multiple times.

Lemma 3.2. *Let $\psi\colon B \to A$ be a surjective function and let ψ^{-1} be the procedure that samples a preimage of ψ uniformly at random. Then:*

$$\Delta\left(\mathcal{U}(B), \psi^{-1} \circ \psi \circ \mathcal{U}(B)\right) = 0\,.$$

Proof. Let $\bar{\boldsymbol{x}} := \left\{\boldsymbol{x}' \in B \colon \psi(\boldsymbol{x}) = \psi(\boldsymbol{x}')\right\}$ be the set of representatives of $\boldsymbol{x}$ in B/ψ. Recall that $\psi^{-1}(y)$ samples uniformly at random a preimage of y. Then:

$$\begin{aligned}
&\Delta\left(\mathcal{U}(B), (\psi^{-1} \circ \psi \circ \mathcal{U})(B)\right) \\
&\quad= \sum_{\boldsymbol{x}\in B} \left| \Pr\left[\boldsymbol{x}' = \boldsymbol{x} \;\middle|\; \boldsymbol{x}' \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathrm{V}}(i)})\right] - \Pr\left[\boldsymbol{x}'' = \boldsymbol{x} \;\middle|\; \begin{array}{c} \boldsymbol{x}' \leftarrow \mathcal{U}(B) \\ y := \psi(\boldsymbol{x}') \\ \boldsymbol{x}'' \leftarrow \psi^{-1}(y) \end{array}\right] \right| \\
&\quad= \sum_{\boldsymbol{x}\in B} \left| \frac{1}{|B|} - \Pr\left[\boldsymbol{x}' = \boldsymbol{x} \mid \boldsymbol{x}' \leftarrow \mathcal{U}(\bar{\boldsymbol{x}})\right] \cdot \Pr\left[\psi(\boldsymbol{x}') = \psi(\boldsymbol{x}) \;\middle|\; \boldsymbol{x}' \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathrm{V}}(i)})\right] \right| \\
&\quad= \sum_{\boldsymbol{x}\in B} \left| \frac{1}{|B|} - \frac{1}{|\bar{\boldsymbol{x}}|} \cdot \Pr\left[\psi(\boldsymbol{x}') = \psi(\boldsymbol{x}) \mid \boldsymbol{x}' \leftarrow \mathcal{U}(B)\right] \right| \\
&\quad= \sum_{\boldsymbol{x}\in B} \left| \frac{1}{|B|} - \frac{1}{|\bar{\boldsymbol{x}}|} \cdot \frac{|\bar{\boldsymbol{x}}|}{|B|} \right| = 0\,.
\end{aligned}$$

□

3.3 Duplex sponge in overwrite mode

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

- a random function $h\colon \{0,1\}^{\leq n} \to \Sigma^c$, 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 $\mathsf{st}_0 = (\boldsymbol{s}, i_{\mathrm{A}}, i_{\mathrm{S}}) = \left((0^r, \boldsymbol{s}_{\mathrm{C}}), 0, r\right)$ 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 3.3. *For $r, c, n \in \mathbb{N}$, let $h\colon \{0,1\}^{\leq n} \to \Sigma^{c}$ 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.*

- $\mathsf{st}_0 := \mathsf{DS}.\mathsf{Start}^{h}(x)$.
  Given as input $x \in \{0,1\}^{\leq n}$, 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:
  $$\mathsf{st}_0 := (\boldsymbol{s}, i_{A}, i_{S}) = \big((0^{r}, \boldsymbol{s}_{\mathrm{C}}), 0, r\big) \in \Sigma^{r+c} \times [0,r] \times [0,r]\,.$$

- $\mathsf{st}' := \mathsf{DS}.\mathsf{Absorb}^{p}(\mathsf{st}, \boldsymbol{\chi})$.
  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 $0 \leq i_A < r$ then:
       – Let $\chi$ 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 $\chi$.
       – Set $i'_A := i_A + 1$ (increment the absorbing index).
       – Set $\mathsf{st}' := (\boldsymbol{s}', i'_A, i'_S)$.
       – Output $\mathsf{DS}.\mathsf{Absorb}(\mathsf{st}', \boldsymbol{\chi}')$.
   (b) If $i_A = r$ then:
       – Set $\boldsymbol{s}' := p(\boldsymbol{s})$.
       – Set $i'_A := 0$ (reset the absorbing index).
       – Set $\mathsf{st}' := (\boldsymbol{s}', i'_A, i'_S)$.
       – Output $\mathsf{DS}.\mathsf{Absorb}(\mathsf{st}', \boldsymbol{\chi})$.
```

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

```
1. If $\ell = 0$ set $\boldsymbol{\rho}$ to the empty string, and output $(\boldsymbol{\rho}, \mathsf{st})$.
2. Otherwise:
   (a) Set $i'_A := 0$ (the absorbing index is not used while squeezing).
   (b) If $0 \leq i_S < r$ then:
       – Set $i'_S := i_S + 1$ (increment the squeezing index).
       – Set $\mathsf{st}' := (\boldsymbol{s}, i'_A, i'_S)$.
       – Compute $(\boldsymbol{\rho}', \mathsf{st}') := \mathsf{DS}.\mathsf{Squeeze}(\mathsf{st}', \ell - 1)$.
       – Output $(\boldsymbol{s}_{i_S} \| \boldsymbol{\rho}', \mathsf{st}')$.
   (c) If $i_S = r$ then:
       – Set $\boldsymbol{s}' := p(\boldsymbol{s})$.
       – Set $i'_S := 0$ (reset the squeezing index).
```

– *Set* $\mathsf{st}' := (\boldsymbol{s}', i'_A, i'_S)$.
– *Output* $\mathsf{DS.Squeeze}(\mathsf{st}', \ell)$.

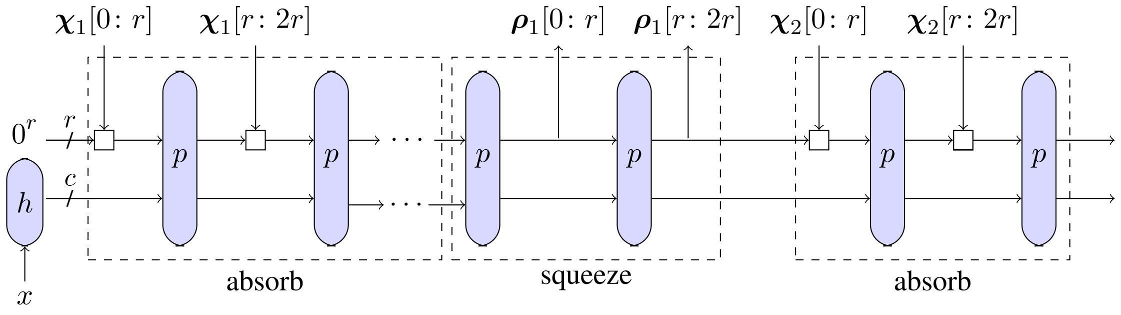


Figure 1: Diagram of the duplex sponge in Construction 3.3.

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 3.4. *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 3.5. *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 3.6. *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** $\kappa_{\mathsf{NARG}}(\lambda, t, n)$ *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* $n \in \mathbb{N}$*,*

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

Definition 3.7. *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* $n \in \mathbb{N}$*,*

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

Definition 3.8. *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* $n \in \mathbb{N}$*,*

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

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 3.9. $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *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 3.10. $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *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* $\tilde{\mathbf{P}}$

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

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

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 3.12. *The* **IP state-restoration game** *for $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ with salt sizes $\boldsymbol{\delta} := (\delta_1, \dots, \delta_\mathsf{k}) \in \mathbb{N}^\mathsf{k}$, random oracles $\boldsymbol{f} = (f_i)_{i \in [\mathsf{k}]} \in \mathcal{U}((\mathcal{Z}_i(\delta_i, n))_{i=1}^{\mathsf{k}})$, and IP state-restoration prover $\tilde{\mathbf{P}}^{\mathrm{sr}}$ is defined below.*

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

1. *Repeat the following until $\tilde{\mathbf{P}}^{\mathrm{sr}}$ decides to exit the loop.*
   (a) *$\tilde{\mathbf{P}}^{\mathrm{sr}}$ outputs $(\mathbb{x}, (\tau_1, \dots, \tau_i), (\alpha_1, \dots, \alpha_i))$, where $\mathbb{x}$ is an instance, $(\alpha_1, \dots, \alpha_i)$ are IP prover messages, and $(\tau_1, \dots, \tau_i)$ are salt strings in $\{0,1\}^{\delta_1} \times \cdots \times \{0,1\}^{\delta_i}$.*
   (b) *Set $\rho_i := f_i(\mathbb{x}, (\tau_1, \dots, \tau_i), (\alpha_1, \dots, \alpha_i))$.*
   (c) *Send ρ_i to $\tilde{\mathbf{P}}^{\mathrm{sr}}$.*
2. *$\tilde{\mathbf{P}}^{\mathrm{sr}}$ outputs $(\mathbb{x}, (\tau_1, \dots, \tau_\mathsf{k}), (\alpha_1, \dots, \alpha_\mathsf{k}))$, where $\mathbb{x}$ is an instance, $(\alpha_1, \dots, \alpha_\mathsf{k})$ are IP prover messages, and $(\tau_1, \dots, \tau_\mathsf{k}) \in \{0,1\}^{\delta_1} \times \cdots \times \{0,1\}^{\delta_\mathsf{k}}$ are salt strings.*
3. *For every $i \in [\mathsf{k}]$, set $\rho_i := f_i(\mathbb{x}, (\tau_1, \dots, \tau_i), (\alpha_1, \dots, \alpha_i))$.*
4. *Output $(\mathbb{x}, (\tau_i)_{i \in [\mathsf{k}]}, (\alpha_i)_{i \in [\mathsf{k}]}, (\rho_i)_{i \in [\mathsf{k}]})$.*

We denote by $\mathsf{tr}^{\mathrm{sr}}$ the list of move-response pairs of the form $((\mathbb{x}, (\tau_1, \dots, \tau_i), (\alpha_1, \dots, \alpha_i)), \rho_i)$ performed in the loop. We show $\mathsf{tr}^{\mathrm{sr}}$ in an execution of the IP state-restoration game using the following notation:

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

$\tilde{\mathbf{P}}^{\mathrm{sr}}$ 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 3.13. *$\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ has* **state-restoration soundness error** *$\varepsilon_{\mathsf{IP}}^{\mathrm{sr}}$ if for all salt sizes $\boldsymbol{\delta} := (\delta_1, \dots, \delta_\mathsf{k}) \in \mathbb{N}^\mathsf{k}$, move budget $t \in \mathbb{N}$, t-move malicious IP state-restoration prover $\tilde{\mathbf{P}}^{\mathrm{sr}}$, and instance size bound $n \in \mathbb{N}$:*

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

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 $(\mathbb{x}, (\tau_1, \dots, \tau_{\mathsf{k}}), (\alpha_1, \dots, \alpha_{\mathsf{k}}))$ 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 3.14. $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *has* **straightline state-restoration knowledge soundness error** $\kappa_{\mathsf{IP}}^{\mathrm{sr}}$ *if there exists a polynomial-time deterministic algorithm* $\mathbf{E}^{\mathrm{sr}}$ *(the* extractor*) such that for all salt sizes* $\boldsymbol{\delta} := (\delta_1, \dots, \delta_{\mathsf{k}}) \in \mathbb{N}^{\mathsf{k}}$*, move budget* $t \in \mathbb{N}$*, t-move deterministic IP state-restoration prover* $\tilde{\mathbf{P}}^{\mathrm{sr}}$*, and instance size bound n:*

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

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 3.15. *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 all salt sizes* $\boldsymbol{\delta} := (\delta_1, \dots, \delta_{\mathsf{k}}) \in \mathbb{N}^{\mathsf{k}}$ *and instance size bound n:*

$$\Pr\left[\begin{array}{l|l} |\mathbb{x}| \le n & \boldsymbol{f} = (f_i)_{i\in[\mathsf{k}]} \leftarrow \mathcal{U}((\mathcal{Z}_i(\delta_i, n))_{i=1}^{\mathsf{k}}) \\ \wedge\ \mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}\big) = 0 & (\mathbb{x}, (\tau_i)_{i\in[\mathsf{k}]}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}) \leftarrow \mathrm{SR}_{\mathsf{IP},\boldsymbol{f},\tilde{\mathbf{P}}^{\mathrm{sr}}}(\boldsymbol{\delta}) \end{array}\right] \le \delta_{\tilde{\mathbf{P}}^{\mathrm{sr}}}(\boldsymbol{\delta}, n)\,.$$

Definition 3.16. $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ *has* **rewinding state-restoration knowledge soundness error** $\kappa_{\mathsf{IP}}^{\mathrm{sr}}$ **with extraction time** $\mathbf{et}_{\mathsf{IP}}^{\mathrm{sr}}$ *if there exists a probabilistic algorithm* $\mathbf{E}^{\mathrm{sr}}$ *(the* extractor*) such that for all salt sizes* $\boldsymbol{\delta} := (\delta_1, \dots, \delta_{\mathsf{k}}) \in \mathbb{N}^{\mathsf{k}}$*, move budget* $t \in \mathbb{N}$*, t-move deterministic IP state-restoration prover* $\tilde{\mathbf{P}}^{\mathrm{sr}}$ *with failure probability* $\delta_{\tilde{\mathbf{P}}^{\mathrm{sr}}}$ *and running time* $\tau_{\tilde{\mathbf{P}}^{\mathrm{sr}}}$*, and instance size bound n:*

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

Moreover, $\mathbf{E}^{\mathrm{sr}}$ *runs in expected time* $\mathbf{et}_{\mathsf{IP}}^{\mathrm{sr}}(\boldsymbol{\delta}, t, n, \delta_{\tilde{\mathbf{P}}^{\mathrm{sr}}}(\boldsymbol{\delta}, n), \tau_{\tilde{\mathbf{P}}^{\mathrm{sr}}}(\boldsymbol{\delta}, n))$ *(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: letting $\delta_1 := \delta$ and $\delta_2, \dots, \delta_{\mathsf{k}} := 0$, for every security parameter $\lambda \in \mathbb{N}$ and an instance size bound $n \in \mathbb{N}$,

$$\mathcal{D}_{\mathsf{IP}}(\lambda, n) := \mathcal{U}(\mathcal{Z}_1(\delta_1, n), \dots, \mathcal{Z}_{\mathsf{k}}(\delta_{\mathsf{k}}, n)) \tag{4}$$

where $\mathcal{Z}_1(\delta_1, n), \ldots, \mathcal{Z}_{\mathsf{k}}(\delta_{\mathsf{k}}, n)$ are the function spaces in Definition 3.11. 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 3.17. *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 $i = 1, \ldots, \mathsf{k}$, derive the i-th IP verifier message ρ_i (as $\mathcal{P}$ does):*
     $$\rho_i := f_i\big(\mathbb{x}, \tau, \alpha_1, \ldots, \alpha_i\big) .$$
  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 3.18 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 3.19 (also adapted from [CY24]) does the same for knowledge soundness. (For ease of notation, below we use the salt size $\delta \in \mathbb{N}$ rather than the salt size vector $\boldsymbol{\delta} := (\delta_1, \ldots, \delta_{\mathsf{k}}) \in \mathbb{N}^{\mathsf{k}}$ where $\delta_1 = \delta$ and $\delta_2, \ldots, \delta_{\mathsf{k}} = 0$.)

Theorem 3.18. *If $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ has state-restoration soundness error $\varepsilon_{\mathsf{IP}}^{\mathrm{sr}}$ (see Definition 3.14) then $\mathsf{NARG} := \mathsf{FS}[\mathsf{IP}, \delta]$ in Construction 3.17 has soundness error $\varepsilon_{\mathsf{NARG}}$ (see Definition 3.5) 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 3.19. *If IP has rewinding state-restoration knowledge soundness error $\kappa_{\mathsf{IP}}^{\mathrm{sr}}$ with extraction time $\mathbf{et}_{\mathsf{IP}}$ (see Definition 3.16) then $\mathsf{NARG} := \mathsf{FS}[\mathsf{IP}, \delta]$ in Construction 3.17 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 3.8) such that, for every security parameter $\lambda \in \mathbb{N}$, query bound $t \in \mathbb{N}$, and instance size bound $n \in \mathbb{N}$,*

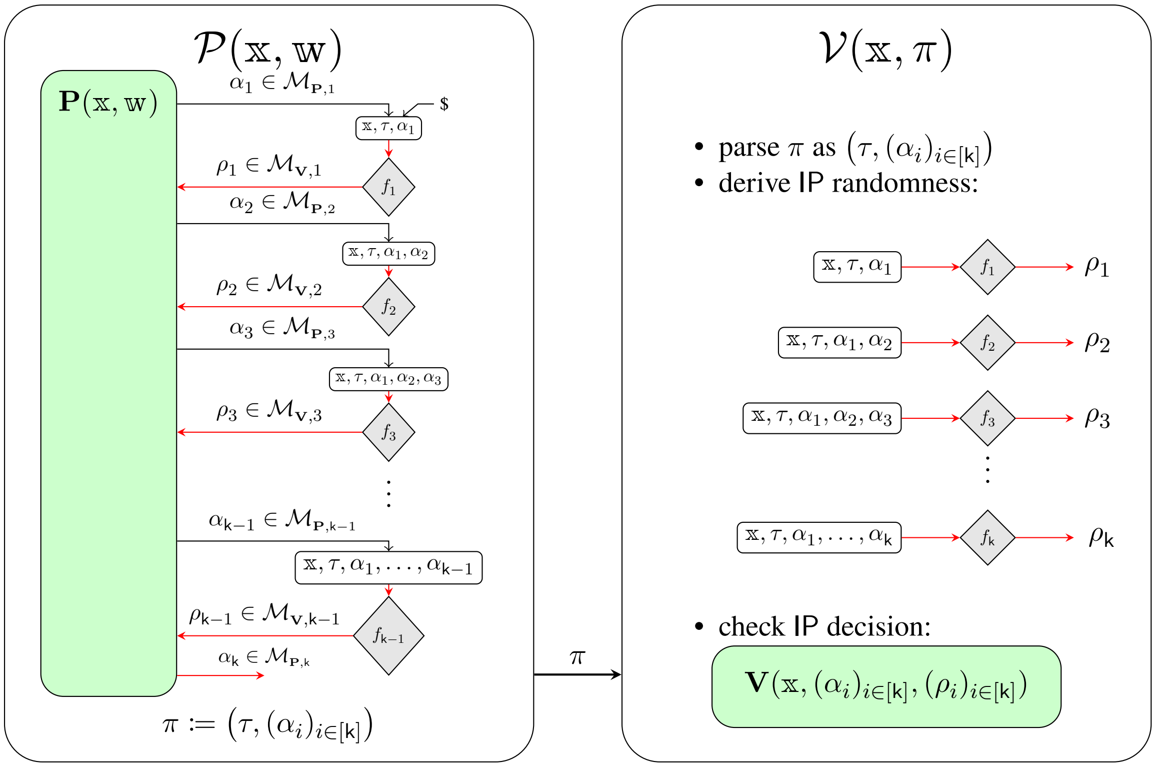

$\mathcal{P}(\mathbb{x}, \mathbb{w})$
$\mathbf{P}(\mathbb{x}, \mathbb{w})$
$\alpha_1 \in \mathcal{M}_{\mathbf{P},1}$
$
$\mathbb{x}, \tau, \alpha_1$
f_1
$\rho_1 \in \mathcal{M}_{\mathbf{V},1}$
$\alpha_2 \in \mathcal{M}_{\mathbf{P},2}$
$\mathbb{x}, \tau, \alpha_1, \alpha_2$
f_2
$\rho_2 \in \mathcal{M}_{\mathbf{V},2}$
$\alpha_3 \in \mathcal{M}_{\mathbf{P},3}$
$\mathbb{x}, \tau, \alpha_1, \alpha_2, \alpha_3$
f_3
$\rho_3 \in \mathcal{M}_{\mathbf{V},3}$
$\alpha_{\mathsf{k}-1} \in \mathcal{M}_{\mathbf{P},\mathsf{k}-1}$
$\mathbb{x}, \tau, \alpha_1, \ldots, \alpha_{\mathsf{k}-1}$
$f_{\mathsf{k}-1}$
$\rho_{\mathsf{k}-1} \in \mathcal{M}_{\mathbf{V},\mathsf{k}-1}$
$\alpha_{\mathsf{k}} \in \mathcal{M}_{\mathbf{P},\mathsf{k}}$
$\pi \coloneqq (\tau, (\alpha_i)_{i\in[\mathsf{k}]})$
π
$\mathcal{V}(\mathbb{x}, \pi)$
• parse π as $(\tau, (\alpha_i)_{i\in[\mathsf{k}]})$
• derive IP randomness:
$\mathbb{x}, \tau, \alpha_1$ f_1 ρ_1
$\mathbb{x}, \tau, \alpha_1, \alpha_2$ f_2 ρ_2
$\mathbb{x}, \tau, \alpha_1, \alpha_2, \alpha_3$ f_3 ρ_3
$\mathbb{x}, \tau, \alpha_1, \ldots, \alpha_{\mathsf{k}}$ f_{k} ρ_{k}
• check IP decision:
$\mathbf{V}(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]})$

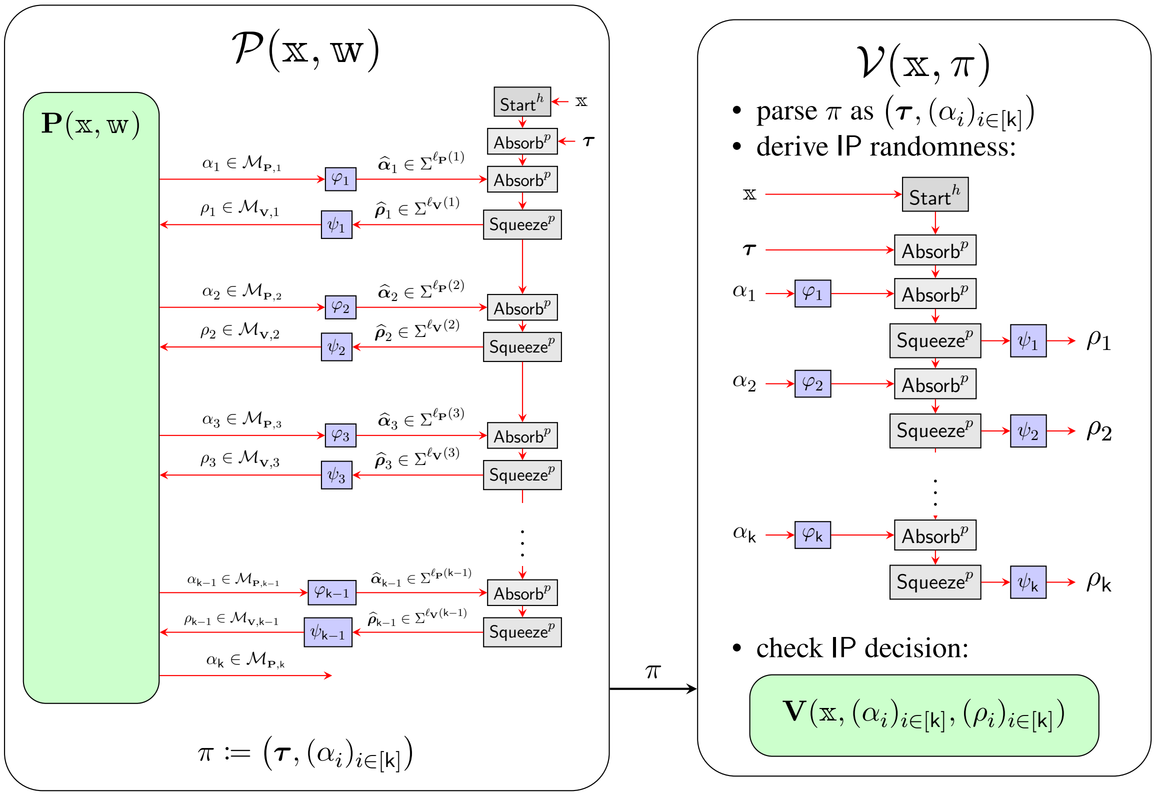
Figure 2: Diagram of $\mathsf{FS}[\mathsf{IP}, \delta]$ as defined in Construction 3.17.

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

Moreover, if the IP state-restoration extractor is straightline (see Definition 3.14) then the NARG extractor is also straightline (see Definition 3.6). In this case:

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

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



$\mathcal{P}(\mathbb{x}, \mathbb{w})$
$\mathbf{P}(\mathbb{x}, \mathbb{w})$
Start^h
$\mathbb{x}$
Absorb^p
$\boldsymbol{\tau}$
$\alpha_1 \in \mathcal{M}_{\mathbf{P},1}$
φ_1
$\widehat{\boldsymbol{\alpha}}_1 \in \Sigma^{\ell_{\mathbf{P}}(1)}$
$\rho_1 \in \mathcal{M}_{\mathbf{V},1}$
ψ_1
$\widehat{\boldsymbol{\rho}}_1 \in \Sigma^{\ell_{\mathbf{V}}(1)}$
Squeeze^p
$\alpha_2 \in \mathcal{M}_{\mathbf{P},2}$
φ_2
$\widehat{\boldsymbol{\alpha}}_2 \in \Sigma^{\ell_{\mathbf{P}}(2)}$
$\rho_2 \in \mathcal{M}_{\mathbf{V},2}$
ψ_2
$\widehat{\boldsymbol{\rho}}_2 \in \Sigma^{\ell_{\mathbf{V}}(2)}$
$\alpha_3 \in \mathcal{M}_{\mathbf{P},3}$
φ_3
$\widehat{\boldsymbol{\alpha}}_3 \in \Sigma^{\ell_{\mathbf{P}}(3)}$
$\rho_3 \in \mathcal{M}_{\mathbf{V},3}$
ψ_3
$\widehat{\boldsymbol{\rho}}_3 \in \Sigma^{\ell_{\mathbf{V}}(3)}$
$\alpha_{\mathsf{k}-1} \in \mathcal{M}_{\mathbf{P},\mathsf{k}-1}$
$\varphi_{\mathsf{k}-1}$
$\widehat{\boldsymbol{\alpha}}_{\mathsf{k}-1} \in \Sigma^{\ell_{\mathbf{P}}(\mathsf{k}-1)}$
$\rho_{\mathsf{k}-1} \in \mathcal{M}_{\mathbf{V},\mathsf{k}-1}$
$\psi_{\mathsf{k}-1}$
$\widehat{\boldsymbol{\rho}}_{\mathsf{k}-1} \in \Sigma^{\ell_{\mathbf{V}}(\mathsf{k}-1)}$
$\alpha_{\mathsf{k}} \in \mathcal{M}_{\mathbf{P},\mathsf{k}}$
$\pi \coloneqq \left(\boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]}\right)$
π
$\mathcal{V}(\mathbb{x}, \pi)$
• parse π as $\left(\boldsymbol{\tau}, (\alpha_i)_{i\in[\mathsf{k}]}\right)$
• derive IP randomness:
$\mathbb{x}$
Start^h
$\boldsymbol{\tau}$
Absorb^p
α_1
φ_1
Absorb^p
Squeeze^p
ψ_1
ρ_1
α_2
φ_2
Absorb^p
Squeeze^p
ψ_2
ρ_2
α_{k}
φ_{k}
Absorb^p
Squeeze^p
ψ_{k}
ρ_{k}
• check IP decision:
$\mathbf{V}(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]})$

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. The inverse permutation p^{-1} will only be used by the adversary, so it does not appear in the construction.

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

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

- *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 $\mathsf{DS.Start}$;
  - 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 $\mathsf{DS.Absorb}$;
  - 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 $\mathsf{DS.Squeeze}$;
  - 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 $\mathsf{DS.Absorb}$;
  - 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 $\mathsf{DS.Squeeze}$;
  - 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. This fits the template of double indifferentiability that we discuss, abstractly, in Appendix A.2.2 (see Definition A.7).

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 argument in the $\mathcal{D}_{\mathsf{IP}}$-oracle model in Construction 3.17.

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 5.4, into a $\theta_\star(t_h, t_p, t_{p^{-1}})$-query malicious argument prover $\tilde{\mathcal{P}}_{\mathsf{std}}$ for $\mathcal{V}_{\mathsf{std}}$ that "behaves the same" as $\mathcal{P}$ up to an additive error $\eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))$; both $\theta_\star(t_h, t_p, t_{p^{-1}})$ and $\eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))$ are defined below. 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*

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

have statistical distance at most:

$$\begin{aligned}
&\eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) := \\
&\quad \frac{7(t_h + t_p + t_{p^{-1}})^2 + 28(L+1)(t_h + t_p + t_{p^{-1}}) + 14(L+1)^2 - 3(t_h + t_p + t_{p^{-1}}) - 13(L+1)}{2\,|\Sigma|^c} + \\
&\qquad \theta_\star(t_h, t_p, t_{p^{-1}}) \cdot \max_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n)\,.
\end{aligned} \tag{5}$$

where

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

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

We prove the lemma in Section 5.8. Before that, we define the two algorithms D2SAlgo and D2STrace in Sections 5.4 and 5.5. In turn, these rely on the procedures BackTrack and LookAhead defined in Sections 5.2 and 5.3.

Throughout, we use the following notation (for various block numbers):

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

and

$$L_{\mathbf{P}} \coloneqq \sum_{i \in [\mathsf{k}]} L_{\mathbf{P}}(i) \quad , \quad L_{\mathbf{V}} \coloneqq \sum_{i \in [\mathsf{k}]} L_{\mathbf{V}}(i) \quad , \quad L \coloneqq L_{\mathbf{P}} + L_{\mathbf{V}} \, . \tag{8}$$

Multiple times we use the fact that the query-answer trace $\mathsf{tr}_{\mathcal{V}}$ of an execution of the argument verifier $\mathcal{V}^{h,p}$ contains exactly 1 query to h, $L_{\mathbf{P}}$ queries to p for absorbing all the IP prover messages, and $L_{\mathbf{V}}$ queries to p for squeezing all the IP verifier messages (for a total of L queries to p).

5.1 Data structure for query-answer trace

Definition 5.2. *Let $h \colon \{0,1\}^{\leq n} \to \Sigma^c$, $p \colon \Sigma^{r+c} \to \Sigma^{r+c}$, and $p^{-1} \colon \Sigma^{r+c} \to \Sigma^{r+c}$ be functions. A* (h, p, p^{-1})**-trace** tr *is a list of tuples of the following form:*
- $(\text{‘}h\text{’}, \mathbb{x}, \boldsymbol{s}_{\mathrm{C}})$ *to denote a query $\mathbb{x} \in \{0,1\}^{\leq n}$ to h with answer $\boldsymbol{s}_{\mathrm{C}} \in \Sigma^c$,*
- $(\text{‘}p\text{’}, \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ *to denote a query $\boldsymbol{s}_{\text{in}} \in \Sigma^{r+c}$ to p with answer $\boldsymbol{s}_{\text{out}} \in \Sigma^{r+c}$,*
- $(\text{‘}p^{-1}\text{’}, \boldsymbol{s}_{\text{out}}, \boldsymbol{s}_{\text{in}})$ *to denote a query $\boldsymbol{s}_{\text{out}} \in \Sigma^{r+c}$ to p^{-1} with answer $\boldsymbol{s}_{\text{in}} \in \Sigma^{r+c}$.*

We use "$\cdot$" as a placeholder for any variable matching the above patterns. For a (h, p, p^{-1})-trace tr, we let $\mathsf{tr}_h^{<j}$ be the first $j-1$ entries in tr of the form $(\text{‘}h\text{’}, \cdot, \cdot)$, $\mathsf{tr}_p^{<j}$ be the first $j-1$ entries in tr of the form $(\text{‘}p\text{’}, \cdot, \cdot)$, and $\mathsf{tr}_{p^{-1}}^{<j}$ be the first $j-1$ entries in tr of the form $(\text{‘}p^{-1}\text{’}, \cdot, \cdot)$.

Sorted query-answer trace. We describe a data structure $\mathsf{tr}_{\triangledown}$ that (efficiently) supports searching for queries or answers in (h, p, p^{-1})-traces. We use this data structure in D2SAlgo and D2STrace.

- $\mathsf{tr}_{\triangledown}.\mathsf{h}$ supports efficient insertions and lookups of query-answer pairs for h.
  - $\mathsf{tr}_{\triangledown}.\mathsf{h}.\mathsf{add}(\mathbb{x}, \boldsymbol{s}_{\mathrm{C}})$ adds the query-answer pair $(\mathbb{x}, \boldsymbol{s}_{\mathrm{C}}) \in \{0,1\}^{\leq n} \times \Sigma^c$ to the data structure.
  - $\mathsf{tr}_{\triangledown}.\mathsf{h}.\mathsf{inlu}(\mathbb{x})$ returns the answer stored for the query $\mathbb{x}$ (or $\perp$ if there are zero or multiple matches).
  - $\mathsf{tr}_{\triangledown}.\mathsf{h}.\mathsf{outlu}(\boldsymbol{s}_{\mathrm{C}})$ returns the query stored for the answer $\boldsymbol{s}_{\mathrm{C}}$ (or $\perp$ if there are zero or multiple matches).

  We write $\mathsf{tr}_{\triangledown}.\mathsf{h}$ to denote the entries sorted by insertion time, and by $|\mathsf{tr}_{\triangledown}.\mathsf{h}|$ the number of entries. These methods can be efficiently performed by maintaining three lists, sorted (respectively) by insertion time, by input query, and by answer. An insertion or lookup takes time $O(\log |\mathsf{tr}_{\triangledown}.\mathsf{h}| \cdot c \cdot \log |\Sigma| + n)$.

- $\mathsf{tr}_{\triangledown}.\mathsf{p}$ supports efficient insertions and lookups of query-answer pairs for p (equivalently, p^{-1}).
  - $\mathsf{tr}_{\triangledown}.\mathsf{p}.\mathsf{add}(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ adds the query-answer pair $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \Sigma^{r+c} \times \Sigma^{r+c}$ to the data structure.
  - $\mathsf{tr}_{\triangledown}.\mathsf{p}.\mathsf{inlu}(\boldsymbol{s}_{\text{in}})$ returns the answer stored for the query $\boldsymbol{s}_{\text{in}}$ (or $\perp$ if there are zero or multiple matches).
  - $\mathsf{tr}_{\triangledown}.\mathsf{p}.\mathsf{outlu}(\boldsymbol{s}_{\text{out}})$ returns the query stored for the answer $\boldsymbol{s}_{\text{out}}$ (or $\perp$ if there are zero or multiple matches).

  We write $\mathsf{tr}_{\triangledown}.\mathsf{p}$ to denote the entries sorted by insertion time, and by $|\mathsf{tr}_{\triangledown}.\mathsf{p}|$ the number of entries. These methods can be efficiently performed by maintaining three lists, sorted (respectively) by insertion time, by input query (capacity segment, then rate segment), and by answer (capacity segment, then rate segment). An insertion or lookup takes time $O(\log |\mathsf{tr}_{\triangledown}.\mathsf{p}| \cdot (r+c) \cdot \log |\Sigma| + (r+c) \cdot \log |\Sigma|)$.

Sorting implies, in particular, that the permutation alphabet Σ has an order over its elements.

5.2 Backtracking procedure

We define the sequences of queries in a given query-answer trace tr that led to the capacity segment $\boldsymbol{s}_{\mathrm{C}}$ in a given permutation state $\boldsymbol{s} = (\boldsymbol{s}_{\mathrm{R}}, \boldsymbol{s}_{\mathrm{C}}) \in \Sigma^{r+c}$, and then formally describe $\mathsf{BackTrack}$.

Definition 5.3. *Let $\mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s}) = \left\{ S^{(k)} \right\}_k$ be the set of sequences where each sequence*

$$S^{(k)} := (\mathbb{x}^{(k)}, \boldsymbol{s}_{\mathrm{in},0}^{(k)}, \boldsymbol{s}_{\mathrm{out},0}^{(k)}, \boldsymbol{s}_{\mathrm{in},1}^{(k)}, \boldsymbol{s}_{\mathrm{out},1}^{(k)}, \dots, \boldsymbol{s}_{\mathrm{in},m_k-1}^{(k)}, \boldsymbol{s}_{\mathrm{out},m_k-1}^{(k)}, \boldsymbol{s}_{\mathrm{in},m_k}^{(k)}) \tag{9}$$

is such that:

- $\boldsymbol{s}_{\mathrm{in},m_k}^{(k)} = \boldsymbol{s}$ *(the last permutation state is the given one);*
- $(\text{‘}h\text{’}, \mathbb{x}^{(k)}, \boldsymbol{s}_{\mathrm{C,in},0}^{(k)}) \in \mathsf{tr}$*;*
- $\forall\, \iota \in [0, m_k-1]\,,\ (\text{‘}p\text{’}, \boldsymbol{s}_{\mathrm{in},\iota}^{(k)}, \boldsymbol{s}_{\mathrm{out},\iota}^{(k)}) \in \mathsf{tr} \vee (\text{‘}p^{-1}\text{’}, \boldsymbol{s}_{\mathrm{out},\iota}^{(k)}, \boldsymbol{s}_{\mathrm{in},\iota}^{(k)}) \in \mathsf{tr}$ *(input-output states agree with p);*
- $\forall\, \iota \in [0, m_k-1]\,,\ \boldsymbol{s}_{\mathrm{C,out},\iota}^{(k)} = \boldsymbol{s}_{\mathrm{C,in},\iota+1}^{(k)}$ *(the capacity segment is shared across queries);*
- $\forall\, \iota \in [0, m_k-1]\,,\ \boldsymbol{s}_{\mathrm{C,in},\iota}^{(k)} \neq \boldsymbol{s}_{\mathrm{C,out},\iota}^{(k)}$ *(no “loops” across query and answer capacity segments);*
- *the length of the sequence is maximal (no other sequence in $\mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s})$ is strictly contained in $S^{(k)}$).*

Definition 5.4. *Let $\mathcal{J}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s}) = \left\{ J^{(k)} \right\}_k$ be the set of index lists associated to $\mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s}) = \left\{ S^{(k)} \right\}_k$, i.e.,*

$$J^{(k)} := (j_h^{(k)}, j_0^{(k)}, j_1^{(k)}, \dots, j_{m_k}^{(k)}) \tag{10}$$

where for every k:

- $j_h^{(k)}$ *is the index in* tr *of the first occurrence of* $(\text{‘}h\text{’}, \mathbb{x}^{(k)}, \boldsymbol{s}_{\mathrm{C,in},0}^{(k)})$*;*
- $\forall\, \iota \in [0, m_k-1],\ j_\iota^{(k)}$ *is the index in* tr *of the first occurrence of* $(\text{‘}p\text{’}, \boldsymbol{s}_{\mathrm{in},\iota}^{(k)}, \boldsymbol{s}_{\mathrm{out},\iota}^{(k)})$ *or* $(\text{‘}p^{-1}\text{’}, \boldsymbol{s}_{\mathrm{out},\iota}^{(k)}, \boldsymbol{s}_{\mathrm{in},\iota}^{(k)})$*;*
- $j_{m_k}^{(k)} = |\mathsf{tr}| + 1$ *(the last index is a convenience value associated to an upcoming query with input $\boldsymbol{s}$).*

The backtracking procedure $\mathsf{BackTrack}(\mathsf{tr}, \mathsf{tr}_\nabla, \boldsymbol{s})$

- takes as input a (h, p, p^{-1})-trace tr, associated data structure tr_∇ (see Section 5.1), and permutation state $\boldsymbol{s}$, and
- returns a special symbol (err or none) or a tuple $(i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \dots, \widehat{\boldsymbol{\alpha}}_i))$ 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 $\widehat{\boldsymbol{\alpha}} = (\widehat{\boldsymbol{\alpha}}_1, \dots, \widehat{\boldsymbol{\alpha}}_i) \in \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)}$ of encoded prover messages.

Informally, $\mathsf{BackTrack}$ uses the trace to recover from the permutation state $\boldsymbol{s}$ the list of elements absorbed so far, returning the current round index i, absorbed instance $\mathbb{x}$, salt string $\boldsymbol{\tau}$, and encoded prover messages $\widehat{\boldsymbol{\alpha}}$.

$\mathsf{BackTrack}(\mathsf{tr}, \mathsf{tr}_\nabla, \boldsymbol{s})$:

1. Initialize an empty list $\mathsf{Outs} := []$.
2. Compute the set $\mathcal{S}_{\mathsf{BT}} := \mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s})$ according to Definition 5.3:

$$\mathcal{S}_{\mathsf{BT}} := \left\{ S^{(k)} \right\}_k = \left\{ (\mathbb{x}^{(k)}, \boldsymbol{s}_{\mathrm{in},0}^{(k)}, \boldsymbol{s}_{\mathrm{out},0}^{(k)}, \boldsymbol{s}_{\mathrm{in},1}^{(k)}, \boldsymbol{s}_{\mathrm{out},1}^{(k)}, \dots, \boldsymbol{s}_{\mathrm{in},m_k-1}^{(k)}, \boldsymbol{s}_{\mathrm{out},m_k-1}^{(k)}, \boldsymbol{s}_{\mathrm{in},m_k}^{(k)}) \right\}_k . \tag{11}$$

3. For every $k \in [|\mathcal{S}_{\mathsf{BT}}|]$, assemble a candidate salt

$$\boldsymbol{\tau}^{(k)} := (\boldsymbol{s}_{\mathrm{R,in},0}^{(k)} \| \cdots \| \boldsymbol{s}_{\mathrm{R,in},L_\delta-1}^{(k)})[0 \colon \delta] \in \Sigma^\delta .$$

   If $\delta > r$ (which implies $L_\delta > 1$), check that $\boldsymbol{s}_{\mathrm{R,in},L_\delta-1}^{(k)}[\delta \bmod r \colon r] = \boldsymbol{s}_{\mathrm{R,out},L_\delta-2}^{(k)}[\delta \bmod r \colon r]$.

4. For every $k \in [|\mathcal{S}_{\mathsf{BT}}|]$, assemble a candidate list of encoded prover messages.
   (a) For every $i \in [\mathsf{k}]$:
      i. Let $L_{\mathsf{ptr}}(i) := L_\delta + \sum_{\iota<i}(L_{\mathbf{P}}(\iota) + L_{\mathbf{V}}(\iota))$ be the current "block offset".
      ii. If $L_{\mathsf{ptr}}(i) + L_{\mathbf{P}}(i) > m_k + 1$, then the k-th candidate is invalid.
      Remove the k-th element from $\mathcal{S}_{\mathsf{BT}}$ and continue to the next element in $\mathcal{S}_{\mathsf{BT}}$ (if any).
      iii. Else (if $L_{\mathsf{ptr}}(i) + L_{\mathbf{P}}(i) \le m_k + 1$):
         A. Use the next $L_{\mathbf{P}}(i)$ rate segments to define

$$\widehat{\boldsymbol{\alpha}}_i^{(k)} := (\boldsymbol{s}^{(k)}_{\mathrm{R,in},L_{\mathsf{ptr}}(i)} \| \boldsymbol{s}^{(k)}_{\mathrm{R,in},L_{\mathsf{ptr}}(i)+1} \| \cdots \| \boldsymbol{s}^{(k)}_{\mathrm{R,in},L_{\mathsf{ptr}}(i)+L_{\mathbf{P}}(i)-1})[0\colon \ell_{\mathbf{P}}(i)] \in \Sigma^{\ell_{\mathbf{P}}(i)} . \tag{12}$$

         B. Define $\mathbf{z}_i^{(k)} := \boldsymbol{s}^{(k)}_{\mathrm{R,in},L_{\mathsf{ptr}}(i)+L_{\mathbf{P}}(i)}[\ell_{\mathbf{P}}(i) \bmod r \colon r]$.
         C. Check that the remainder of the message is equal to the previous output:

$$\mathbf{z}_i^{(k)} = \boldsymbol{s}^{(k)}_{\mathrm{R,out},L_{\mathsf{ptr}}(i)+L_{\mathbf{P}}(i)-1}[\ell_{\mathbf{P}}(i) \bmod r \colon r] .$$

         If the above does not hold, remove the k-th element from $\mathcal{S}_{\mathsf{BT}}$ and continue to the next element in $\mathcal{S}_{\mathsf{BT}}$ (if any).
         D. If $L_{\mathsf{ptr}}(i) + L_{\mathbf{P}}(i) = m_k + 1$, store $(i, \mathbb{x}^{(k)}, \boldsymbol{\tau}^{(k)}, (\widehat{\boldsymbol{\alpha}}_1^{(k)}, \ldots, \widehat{\boldsymbol{\alpha}}_i^{(k)}))$ into Outs and continue to the next element in $\mathcal{S}_{\mathsf{BT}}$.
         E. If $L_{\mathsf{ptr}}(i) + L_{\mathbf{P}}(i) + L_{\mathbf{V}}(i) < m_k + 1$, check that the 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):

$$\begin{aligned} &\boldsymbol{s}^{(k)}_{\mathrm{R,in},L_{\mathsf{ptr}}(i)+L_{\mathbf{P}}(i)+1} \| \cdots \| \boldsymbol{s}^{(k)}_{\mathrm{R,in},L_{\mathsf{ptr}}(i)+L_{\mathbf{P}}(i)+L_{\mathbf{V}}(i)} \\ &\quad = \boldsymbol{s}^{(k)}_{\mathrm{R,out},L_{\mathsf{ptr}}(i)+L_{\mathbf{P}}(i)} \| \cdots \| \boldsymbol{s}^{(k)}_{\mathrm{R,out},L_{\mathsf{ptr}}(i)+L_{\mathbf{P}}(i)+L_{\mathbf{V}}(i)-1} . \end{aligned}$$

         If the above does not hold, remove the k-th element from $\mathcal{S}_{\mathsf{BT}}$ and continue to the next element in $\mathcal{S}_{\mathsf{BT}}$ (if any). Else, continue to the next index $i+1$ (if any).
         F. If $L_{\mathsf{ptr}}(i) + L_{\mathbf{P}}(i) + L_{\mathbf{V}}(i) \ge m_k + 1$, then continue to the next element in $\mathcal{S}_{\mathsf{BT}}$ (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, $(i, \mathbb{x}^{(1)}, \boldsymbol{\tau}^{(1)}, (\widehat{\boldsymbol{\alpha}}_1^{(1)}, \ldots, \widehat{\boldsymbol{\alpha}}_i^{(1)}))$.

Time complexity. The time complexity of $\mathsf{BackTrack}$ is dominated by the time to construct $\mathcal{S}_{\mathsf{BT}}$ in Equation 11. If an element in $\mathcal{S}_{\mathsf{BT}}$ matches the duplex sponge construction, then it is stored in Outs. By the return condition in Item 5, $\mathsf{BackTrack}$ can stop when Outs has two elements, so $\mathsf{BackTrack}$ can be implemented so to look for at most one element. An element in $\mathcal{S}_{\mathsf{BT}}$ contains a statement and a tuple of permutation states of length at most $|\mathsf{tr}_{\nabla}.\mathsf{p}|$. The sequence of permutation states can be computed by backtracking. This process involves the following operations.

• Looking up the last capacity segment in $\mathsf{tr}_{\nabla}.\mathsf{p}$ (with the initial segment being the input to backtrack).

  Each lookup in $\mathsf{tr}_{\nabla}.\mathsf{p}$ is for elements of length $(r+c)\cdot\log|\Sigma|$ and takes time $\log|\mathsf{tr}_{\nabla}.\mathsf{p}|$ via dichotomic search. If an element is found, the associated permutation state (of length $(r+c)\cdot\log|\Sigma|$) is added to $\mathcal{S}_{\mathsf{BT}}$.

- Continuing the search until reaching a permutation state that is not in $\mathsf{tr}_\nabla.\mathsf{p}$ but is in $\mathsf{tr}_\nabla.\mathsf{h}$.

  Since all elements looked up are different (by Definition 5.3, each permutation state in a sequence $S^{(k)} \in \mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s})$ has different input capacity segments), at most $|\mathsf{tr}_\nabla.\mathsf{p}|$ lookups are needed to find the last capacity segment. The last lookup is done over the trace $\mathsf{tr}_\nabla.\mathsf{h}$ and takes at most time $\log|\mathsf{tr}_\nabla.\mathsf{h}| \cdot (c \cdot \log|\Sigma|) + n$. If the lookup returns an entry, the instance $\mathbb{x}$ (of length n) is copied into $\mathcal{S}_{\mathsf{BT}}$.

Overall, the total running time is

$$O\left(|\mathsf{tr}_\nabla.\mathsf{p}| \cdot \log|\mathsf{tr}_\nabla.\mathsf{p}| \cdot (r+c) \cdot \log|\Sigma| + \log|\mathsf{tr}_\nabla.\mathsf{h}| \cdot (c \cdot \log|\Sigma|) + n\right) . \tag{13}$$

5.3 Lookahead procedure

The lookahead procedure $\mathsf{LookAhead}(\mathsf{tr}_\nabla.\mathsf{p}, \boldsymbol{s}, i)$

- takes as input the data structure $\mathsf{tr}_\nabla.\mathsf{p}$ as defined in Section 5.1, a permutation state $\boldsymbol{s} = (\boldsymbol{s}_{\mathrm{R}}, \boldsymbol{s}_{\mathrm{C}}) \in \Sigma^{r+c}$, and 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, $\mathsf{LookAhead}$ recovers an encoded verifier message $\widehat{\boldsymbol{\rho}}_i$ obtained from a hash chain starting at $\boldsymbol{s}$.

$\mathsf{LookAhead}(\mathsf{tr}_\nabla.\mathsf{p}, \boldsymbol{s}, i)$:

1. Parse the query-answer list of $\mathsf{tr}_\nabla.\mathsf{p}$ into a list

$$\mathcal{S}_{\mathsf{LA}} \coloneqq \left(S^{(k)}_{\mathsf{LA}}\right)_k = \left(\boldsymbol{s}^{(k)}_{\mathrm{in},0}, \boldsymbol{s}^{(k)}_{\mathrm{out},0}, \boldsymbol{s}^{(k)}_{\mathrm{in},1}, \boldsymbol{s}^{(k)}_{\mathrm{out},1}, \ldots, \boldsymbol{s}^{(k)}_{\mathrm{in},m_k-1}, \boldsymbol{s}^{(k)}_{\mathrm{out},m_k-1}\right)_k \tag{14}$$

   where, for every $k \in [|\mathcal{S}_{\mathsf{LA}}|]$:
   (a) $m_k \leq L_{\mathbf{V}}(i)$ (i.e., the length of the chain is at most $L_{\mathbf{V}}(i)$);
   (b) $\boldsymbol{s}^{(k)}_{\mathrm{in},0} = \boldsymbol{s}$ (i.e., the first input state equals the permutation state received as input);
   (c) $\forall\, \iota \in [m_k - 1]$, $\boldsymbol{s}^{(k)}_{\mathrm{out},\iota-1} = \boldsymbol{s}^{(k)}_{\mathrm{in},\iota}$ (i.e., the $(\iota-1)$-th output state equals the ι-th input state);
   (d) $\forall\, \iota \in [0, m_k - 1]$, $\boldsymbol{s}^{(k)}_{\mathrm{C,in},\iota} \neq \boldsymbol{s}^{(k)}_{\mathrm{C,out},\iota}$ (no "loops" across query and answer capacity segments);
   (e) the length the sequence is maximal (among sequences satisfying the above properties).
2. Final output:
   (a) If $\mathcal{S}_{\mathsf{LA}}$ contains more than one element, then return err.
   (b) If $\mathcal{S}_{\mathsf{LA}}$ is empty, then return none.
   (c) Else, $\mathcal{S}_{\mathsf{LA}}$ contains a single element $S^{(1)}_{\mathsf{LA}}$. Sample $L_{\mathbf{V}}(i) - m_1$ additional random permutation states $\boldsymbol{s}^{(1)}_{\mathrm{R,out},m_1}, \ldots, \boldsymbol{s}^{(1)}_{\mathrm{R,out},L_{\mathbf{V}}(i)-1} \leftarrow \mathcal{U}(\Sigma^{r+c})$. Return

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

Time complexity. The time of $\mathsf{LookAhead}$ is dominated by Item 1c. Producing an element in $\mathcal{S}_{\mathsf{LA}}$ involves, at worst, $L_{\mathbf{V}}(i)$ lookups by input in $\mathsf{tr}_\nabla.\mathsf{p}$ (via its method $\mathsf{tr}_\nabla.\mathsf{p.inlu}$). The search stops if it encounters two conflicting chains. Since there are no loops (the query and answer capacity segments are different), at most $|\mathsf{tr}_\nabla.\mathsf{p}|$ lookups by input lead to the last capacity segment. Therefore, the overall time complexity is

$$O(L_{\mathbf{V}}(i) \cdot \log|\mathsf{tr}_\nabla.\mathsf{p}| \cdot (r+c) \cdot \log|\Sigma|) . \tag{15}$$

5.4 Prover transformation

The auxiliary procedure D2SAlgo translates queries of a malicious prover $\tilde{\mathcal{P}}$ for $\mathcal{V}$ (intended for oracles sampled from $\mathcal{D}_{\mathfrak{S}}(\lambda, n)$) into corresponding oracle queries of a malicious prover $\tilde{\mathcal{P}}_{\mathsf{std}}$ for $\mathcal{V}_{\mathsf{std}}$ (intended for oracles sampled from $\mathcal{D}_{\mathsf{IP}}(\lambda, n)$). To reduce the complexity of this step, we first define D2SQuery to deal with most of the complexity of translating the oracle queries, and then define D2SAlgo as a thin wrapper around it, dealing with the encoding and decoding of messages.

Oracle wrapper D2SQuery. The oracle D2SQuery relies on an "intermediate" oracle distribution $\mathcal{D}_{\Sigma}(\lambda, n)$ to answer oracle queries for (h, p, p^{-1}). It is defined as follows

$$\mathcal{D}_{\Sigma}(\lambda, n) := \mathcal{U}\left((\{0,1\}^{\leq n} \times \Sigma^{\delta} \times \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)} \to \Sigma^{\ell_{\mathbf{V}}(i)})_{i \in [\mathsf{k}]} \right) . \tag{16}$$

Oracles sampled from $\mathcal{D}_{\Sigma}(\lambda, n)$ are denoted as g_i for $i \in [\mathsf{k}]$. The oracle D2SQuery works as follows.

1. Initialize the following.
   - A list tr that will store query-answer pairs for h and p, stored as tuples $(`h', \mathbb{x}, \boldsymbol{s}_{\mathrm{C}})$ for $h$ and tuples $(`p', \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$ for p, ordered by query time of the adversary.
   - A list Cache_p that will store query-answer pairs $(\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$, sorted lexicographically by input.
   - A data structure tr_{∇} that will maintain entries from tr, allowing for efficient lookups (see Section 5.1).
2. For every query $\mathbb{x} \in \{0,1\}^{\leq n}$ to the oracle h:
   (a) Lookup $\boldsymbol{s}_{\mathrm{C,out}} := \mathsf{tr}_{\nabla}.\mathsf{h.inlu}(\mathbb{x}) \in \Sigma^{c}$.
   (b) If the lookup returned $\perp$, sample $\boldsymbol{s}_{\mathrm{C,out}} \leftarrow \mathcal{U}(\Sigma^{c})$ and run $\mathsf{tr}_{\nabla}.\mathsf{h.add}(\mathbb{x}, \boldsymbol{s}_{\mathrm{C,out}})$.
   (c) Add $(`h', \mathbb{x}, \boldsymbol{s}_{\mathrm{C,out}})$ to tr.
   (d) Return $\boldsymbol{s}_{\mathrm{C,out}}$.
3. For every query $\boldsymbol{s}_{\mathrm{out}} \in \Sigma^{r+c}$ to the oracle p^{-1}:
   (a) Lookup $\boldsymbol{s}_{\mathrm{in}} := \mathsf{tr}_{\nabla}.\mathsf{p.outlu}(\boldsymbol{s}_{\mathrm{out}}) \in \Sigma^{r+c}$.
   (b) If the lookup returned $\perp$, sample $\boldsymbol{s}_{\mathrm{in}} \leftarrow \mathcal{U}(\Sigma^{r+c})$ and run $\mathsf{tr}_{\nabla}.\mathsf{p.add}(\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$.
   (c) Add $(`p', \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$ to tr.
   (d) Return $\boldsymbol{s}_{\mathrm{in}}$.
4. For every query $\boldsymbol{s}_{\mathrm{in}} = (\boldsymbol{s}_{\mathrm{R,in}}, \boldsymbol{s}_{\mathrm{C,in}}) \in \Sigma^{r+c}$ to the oracle p:
   (a) Run $\mathsf{BackTrack}(\mathsf{tr}, \mathsf{tr}_{\nabla}, \boldsymbol{s}_{\mathrm{in}})$, and proceed as below according to its output.
   (b) If BackTrack's output is err then abort.
   (c) If BackTrack's output is none, then:
      i. If $\exists\, \boldsymbol{s}_{\mathrm{out}} \in \Sigma^{r+c}$: $(\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}) \in \mathsf{Cache}_p$, remove the first occurrence of $(\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$ from Cache_p and let the first such match be denoted $\boldsymbol{s}_{\mathrm{out}}$.
      ii. Else, let $\boldsymbol{s}_{\mathrm{out}} := \mathsf{tr}_{\nabla}.\mathsf{p.inlu}(\boldsymbol{s}_{\mathrm{in}}) \in \Sigma^{r+c}$, if any.
      iii. Else, sample $\boldsymbol{s}_{\mathrm{out}} \leftarrow \mathcal{U}(\Sigma^{r+c})$ and run $\mathsf{tr}_{\nabla}.\mathsf{p.add}(\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$.
   (d) If BackTrack's output is a tuple $(i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i))$ such that $\exists\, \iota \in [i]$, $\widehat{\boldsymbol{\alpha}}_{\iota} \notin \mathrm{Im}(\varphi_{\iota})$ then:
      i. Let $\boldsymbol{s}_{\mathrm{out}} := \mathsf{tr}_{\nabla}.\mathsf{p.inlu}(\boldsymbol{s}_{\mathrm{in}}) \in \Sigma^{r+c}$, if any.
      ii. Else, sample $\boldsymbol{s}_{\mathrm{out}} \leftarrow \mathcal{U}(\Sigma^{r+c})$ and run $\mathsf{tr}_{\nabla}.\mathsf{p.add}(\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$.
   (e) If BackTrack's output is a tuple $(i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i))$ such that $\forall\, \iota \in [i]$, $\widehat{\boldsymbol{\alpha}}_{\iota} \in \mathrm{Im}(\varphi_{\iota})$ then:
      i. Set $\widehat{\boldsymbol{\rho}}_i := g_i(\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i)$.
      ii. Let $\boldsymbol{s}_{\mathrm{out}} \in \Sigma^{r+c} = \mathsf{tr}_{\nabla}.\mathsf{p.inlu}(\boldsymbol{s}_{\mathrm{in}})$, if any.
      iii. Else
         A. Set $\ell_z := r - (\ell_{\mathbf{V}}(i) \bmod r)$ and sample a random remainder $\mathbf{z} \leftarrow \mathcal{U}(\Sigma^{\ell_z})$.

B. Parse $\widehat{\boldsymbol{\rho}}_i \| \mathbf{z}$ as rate segments $\boldsymbol{s}_{\mathrm{R}}^{(0)}, \ldots, \boldsymbol{s}_{\mathrm{R}}^{(L_{\mathbf{V}}(i)-1)} \in \Sigma^r$.

C. Sample random capacities $\boldsymbol{s}_{\mathrm{C}}^{(0)}, \ldots, \boldsymbol{s}_{\mathrm{C}}^{(L_{\mathbf{V}}(i)-1)} \leftarrow \mathcal{U}(\Sigma^c)$.

D. Append to Cache_p the following pairs in $\Sigma^{r+c} \times \Sigma^{r+c}$:

$$\left((\boldsymbol{s}_{\mathrm{R}}^{(0)}, \boldsymbol{s}_{\mathrm{C}}^{(0)}), (\boldsymbol{s}_{\mathrm{R}}^{(1)}, \boldsymbol{s}_{\mathrm{C}}^{(1)})\right), \ \ldots, \ \left((\boldsymbol{s}_{\mathrm{R}}^{(L_{\mathbf{V}}(i)-2)}, \boldsymbol{s}_{\mathrm{C}}^{(L_{\mathbf{V}}(i)-2)}), (\boldsymbol{s}_{\mathrm{R}}^{(L_{\mathbf{V}}(i))-1}, \boldsymbol{s}_{\mathrm{C}}^{(L_{\mathbf{V}}(i)-1)})\right) .$$

E. Set $\boldsymbol{s}_{\mathrm{out}} := (\boldsymbol{s}_{\mathrm{R}}^{(0)}, \boldsymbol{s}_{\mathrm{C}}^{(0)})$ and run $\mathsf{tr}_{\nabla}.\mathsf{p}.\mathsf{add}(\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$.

(f) Add ('p', $\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}$) to tr.

(g) Return $\boldsymbol{s}_{\mathrm{out}}$.

Time complexity of D2SQuery. We discuss the time of D2SQuery depending on the type of query.

- *Query to h.* D2SQuery does a lookup for a previously-answered query in $\mathsf{tr}_{\nabla}.\mathsf{h}$ and stores the query in tr. Hence
$$t_{\mathsf{D2SQuery},h} = O((c \cdot \log|\Sigma|) \cdot \log|\mathsf{tr}_{\nabla}.\mathsf{h}| + n) .$$

- *Query to p.* The runtime of D2SQuery for a query to p is dominated by the time to compute BackTrack with inputs tr_{∇}, as well as the time to insert at most $\max_{i \in [\mathsf{k}]} L_{\mathbf{V}}(i)$ pairs in $\Sigma^{r+c} \times \Sigma^{r+c}$ in Cache_p. Hence
$$\begin{aligned} &t_{\mathsf{D2SQuery},p} \\ &= O(t_{\mathsf{BackTrack}} + (r+c) \cdot \log|\Sigma| \cdot \max_{i \in [\mathsf{k}]} L_{\mathbf{V}}(i)) \\ &= O\left(|\mathsf{tr}_{\nabla}.\mathsf{p}| \cdot \log|\mathsf{tr}_{\nabla}.\mathsf{p}| \cdot (r+c) \cdot \log|\Sigma| + \log|\mathsf{tr}_{\nabla}.\mathsf{h}| \cdot (c \cdot \log|\Sigma|) + n + (r+c) \cdot \log|\Sigma| \cdot \max_{i \in [\mathsf{k}]} L_{\mathbf{V}}(i)\right) . \end{aligned}$$
(via the expression for $t_{\mathsf{BackTrack}}$ in Equation 13)

- *Query to p^{-1}.* The time of D2SQuery is dominated by the lookup for a previously-answered query in $\mathsf{tr}_{\nabla}.\mathsf{p}$:
$$t_{\mathsf{D2SQuery},p^{-1}} = O((r+c) \cdot \log|\mathsf{tr}_{\nabla}.\mathsf{p}|) .$$

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

```
D2SAlgo^f(A):
1. For every $i \in [\mathsf{k}]$, initialize an empty list $\mathsf{tr}_i$.
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 $\mathsf{tr}_i$, respond with the corresponding answer.
       Otherwise, proceed as follows.
   (b) For every $\iota \in [i]$, set $\alpha_\iota := \varphi_\iota^{-1}(\widehat{\boldsymbol{\alpha}}_\iota)$ (compute the unique preimage of $\alpha_\iota$ under $\varphi_\iota$).
       (By Item 4e, each $\alpha_\iota$ contained in a query from Item 4(e)i is in the image space of $\varphi_\iota$.)
   (c) Let $\check{\tau} := \mathsf{bin}(\boldsymbol{\tau}) \in \{0,1\}^{\delta_\star}$ be the binary representation of $\boldsymbol{\tau} \in \Sigma^{\delta}$, where $\delta_\star := \log|\Sigma| \cdot \delta$.
   (d) Set $\rho_i := f_i(\mathbb{x}, \check{\tau}, \alpha_1, \ldots, \alpha_i)$.
   (e) Sample $\widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i)$ (sample a preimage of $\rho_i$ under $\psi_i$).
   (f) Add the query-answer pair $((\mathbb{x}, \check{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i), \widehat{\boldsymbol{\rho}}_i)$ to $\mathsf{tr}_i$.
   (g) Respond to $\mathcal{A}$ with $\widehat{\boldsymbol{\rho}}_i$.
```

4. Let $\mathcal{A}$'s output be $(\mathbb{x}, \pi_{\mathcal{A}})$, and parse $\pi_{\mathcal{A}}$ as $(\boldsymbol{\tau}, \alpha_1, \ldots, \alpha_{\mathsf{k}})$.
5. Set $\breve{\pi} := (\breve{\tau}, \alpha_1, \ldots, \alpha_{\mathsf{k}})$ where $\breve{\tau} := \mathsf{bin}(\boldsymbol{\tau})$.
6. Output $(\mathbb{x}, \breve{\pi})$.

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

$$\mathsf{D2SAlgo}^{\boldsymbol{f}}(\mathcal{A}) := \mathcal{A}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}} . \tag{17}$$

Query complexity of D2SAlgo. Intuitively, D2SAlgo groups queries to p into a single query to $\boldsymbol{f}$ (if any). If D2SQuery does not abort, $\mathcal{A}$ must perform at least $L_{\mathbf{P}}(i)$ queries to p in order to trigger a query to f_i, but can later trigger additional queries to f_i via additional queries to p by keeping the same message prefix. Separately, queries to h or p^{-1} do not trigger queries to $\boldsymbol{f}$. Hence the number of queries by D2SAlgo($\mathcal{A}$) to $\boldsymbol{f}$ is at most

$$\theta_\star(t_h, t_p, t_{p^{-1}}) := t_p .$$

Time complexity of D2SAlgo. D2SAlgo incurs an additive cost for each oracle query made by $\mathcal{A}$. The cost is given by the runtime of D2SQuery, and the time to compute (for each $i \in [\mathsf{k}]$ queried by D2SQuery) ψ_i^{-1} and φ_ι^{-1} for all $\iota \leq i$. Let:

- $t_{\boldsymbol{\varphi}^{-1}}$ be a bound on the time to compute each φ_i^{-1},
- $t_{\boldsymbol{\psi}^{-1}}$ be a bound on the time to compute each ψ_i^{-1},
- t_{bin} be the time to compute the binary encoding of the input salt (note that $t_{\mathsf{bin}} = \delta \log |\Sigma|$ as the salt consists of δ symbols in Σ),
- $t_{\mathsf{D2SQuery},h}$, $t_{\mathsf{D2SQuery},p}$, $t_{\mathsf{D2SQuery},p^{-1}}$ be bounds for D2SQuery to respond to queries to h, p, p^{-1}.

The time complexity of D2SAlgo is:

$$\begin{aligned} &O\left(t_h \cdot t_{\mathsf{D2SQuery},h} + t_p \cdot (t_{\mathsf{D2SQuery},p} + t_{\mathsf{bin}} + t_{\boldsymbol{\psi}^{-1}} + t_{\boldsymbol{\varphi}^{-1}}) + t_{p^{-1}} \cdot t_{\mathsf{D2SQuery},p^{-1}}\right) \\ &\leq O\Big((t_h + t_p + t_{p^{-1}}) \cdot (t_p + t_{p^{-1}}) \cdot \log\left(t_h(t_p + t_{p^{-1}})\right) \cdot (r + c) \cdot \log|\Sigma| + n \\ &\qquad + (t_p + t_{p^{-1}})(t_{\boldsymbol{\psi}^{-1}} + t_{\boldsymbol{\varphi}^{-1}} + \delta \cdot \log|\Sigma|))\Big) , \end{aligned} \tag{18}$$

provided $t_p + t_{p^{-1}} \geq \max_{i \in [\mathsf{k}]} L_{\mathbf{V}}(i)$ (used in the bound for $t_{\mathsf{D2SQuery},p}$).

5.5 Trace transformation

Before defining D2STrace we introduce StdTrace, re-mapping query-answer traces into input-output traces from $\mathcal{D}_\Sigma(\lambda, n)$ (see Equation 16). Then, we define D2STrace as a thin wrapper around it dealing with encoding and decoding of messages.

5.5.1 Auxiliary procedure StdTrace

The procedure StdTrace works as follows.

```
StdTrace(tr):
1. Initialize a list $\mathsf{tr}_{\mathsf{std}}$, that will maintain query-answer pairs for $\boldsymbol{g}$, sorted by query time.
```

2. Initialize a list $\mathsf{tr}^{\mathsf{LA}}_{\mathsf{std}}$ that will maintain query-answer pairs for $\boldsymbol{g}$, sorted lexicographically by input and then by output.
3. Initialize the data structure tr_{∇} for tr as defined in Section 5.1:
   (a) For each entry in tr of the form $(`h', \mathbb{x}, \boldsymbol{s}_{\mathrm{C,out}})$, run $\mathsf{tr}_{\nabla}.\mathsf{h}.\mathsf{add}(\mathbb{x}, \boldsymbol{s}_{\mathrm{C,out}})$.
   (b) For each entry in tr of the form $(`p', \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$, run $\mathsf{tr}_{\nabla}.\mathsf{p}.\mathsf{add}(\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$.
   (c) For each entry in tr of the form $(`p^{-1}', \boldsymbol{s}_{\mathrm{out}}, \boldsymbol{s}_{\mathrm{in}})$, ignore the entry and continue.
4. For each $j \in [|\mathsf{tr}|]$:
   (a) If tr_j is of the form $(`p', \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})$:
      i. Run $\mathsf{BackTrack}(\mathsf{tr}, \mathsf{tr}_{\nabla}, \boldsymbol{s}_{\mathrm{in}})$, and do the following based on its output.
      ii. If the output is err, then <u>abort</u>.
      iii. If the output is none, or a tuple $(i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \dots, \widehat{\boldsymbol{\alpha}}_i))$ such that $\exists\, \iota \in [i]$, $\widehat{\boldsymbol{\alpha}}_\iota \notin \mathrm{Im}(\varphi_\iota)$, then continue to the next element in tr.
      iv. Else the output is a tuple $(i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \dots, \widehat{\boldsymbol{\alpha}}_i))$ such that $\forall\, \iota \in [i]$, $\widehat{\boldsymbol{\alpha}}_\iota \in \mathrm{Im}(\varphi_\iota)$, so then:
         A. If $\mathsf{tr}^{\mathsf{LA}}_{\mathsf{std}}$ contains a query-answer pair of the form $\big((i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \dots, \widehat{\boldsymbol{\alpha}}_i)), \cdot\big)$ then add such a pair to $\mathsf{tr}_{\mathsf{std}}$.
         B. Else, run $\mathsf{LookAhead}(\mathsf{tr}_{\nabla}.\mathsf{p}, \boldsymbol{s}_{\mathrm{in}}, i)$, and do the following based on its output.
         C. If LookAhead outputs err, then <u>abort</u>.
         D. Else, denote LookAhead's output as $\widehat{\boldsymbol{\rho}}_i \in \Sigma^{\ell_{\mathbf{V}}(i)}$ and add the query-answer pair $\big((i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \dots, \widehat{\boldsymbol{\alpha}}_i)), \widehat{\boldsymbol{\rho}}_i\big)$ to $\mathsf{tr}^{\mathsf{LA}}_{\mathsf{std}}$ and to $\mathsf{tr}_{\mathsf{std}}$. (The output is not none as $(`p', \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}) \in \mathsf{tr}$.)
   (b) If tr_j is of the form $(`p^{-1}', \boldsymbol{s}_{\mathrm{out}}, \boldsymbol{s}_{\mathrm{in}})$ or $(`h', \mathbb{x}, \boldsymbol{s}_{\mathrm{C,out}})$, continue.
5. Return $\mathsf{tr}_{\mathsf{std}}$.

Time complexity. The time complexity of StdTrace is dominated by the following items.

- The time t_{sort} to build tr_{∇} (see Item 3b), of size at most $|\mathsf{tr}|$. (Nit: this procedure does not rely on input lookups for $\mathsf{tr}_{\nabla}.\mathsf{h}$.)
- The time $t_{\mathsf{BackTrack}}$ for BackTrack, executed for every query to p in tr (see Item 4(a)i). A bound for the running time of BackTrack is given in Equation 13.
- The time $t_{\mathsf{LookAhead}}$ of LookAhead, executed every time BackTrack returns a tuple (see Item 4(a)iv). A bound for the running time of LookAhead is given in Equation 15.
- The time t_{std} to look up a tuple in $\mathsf{tr}^{\mathsf{LA}}_{\mathsf{std}}$. Since its entries are elements $\big((i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \dots, \widehat{\boldsymbol{\alpha}}_i)), \widehat{\boldsymbol{\rho}}_i\big) \in [\mathsf{k}] \times \{0,1\}^{\leq n} \times \Sigma^{\delta} \times \mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i} \times \mathcal{M}_{\mathbf{V},i}$,

$$t_{\mathrm{std}} \leq \log|\mathsf{tr}| \cdot \Big(\log \mathsf{k} + n + \big(\delta + \textstyle\sum_{\iota \in [i]} L_{\mathbf{P}}(\iota) + L_{\mathbf{V}}(i)\big) \log|\Sigma|\Big) \ . \tag{19}$$

Therefore:

$$\begin{aligned} & t_{\mathsf{StdTrace}} \\ & = O(t_{\mathrm{sort}} + |\mathsf{tr}| \cdot (t_{\mathsf{BackTrack}} + t_{\mathsf{LookAhead}} + t_{\mathrm{std}})) \\ & = O\left(|\mathsf{tr}| \cdot (t_{\mathsf{BackTrack}} + t_{\mathsf{LookAhead}} + t_{\mathrm{std}})\right) \ . \quad \text{(by Equations 13, 15 and 19)} \end{aligned} \tag{20}$$

5.5.2 Auxiliary procedure D2STrace

The algorithm D2STrace is defined as follows.

```
D2STrace(tr):
1. Initialize an empty list $\mathsf{tr}_{\mathsf{std}}$.
2. Set $\widehat{\mathsf{tr}_{\mathsf{std}}} \coloneqq \mathsf{StdTrace}(\mathsf{tr})$.
3. For each tuple $((i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \dots, \widehat{\boldsymbol{\alpha}}_i)), \widehat{\boldsymbol{\rho}}_i)$ in $\widehat{\mathsf{tr}_{\mathsf{std}}}$, where $i \in [\mathsf{k}]$, $\mathbb{x} \in \{0,1\}^{\leq n}$, $\boldsymbol{\tau} \in \Sigma^{\delta}$, $\widehat{\boldsymbol{\alpha}}_1 \in \Sigma^{\ell_{\mathbf{P}}(1)}, \dots, \widehat{\boldsymbol{\alpha}}_i \in \Sigma^{\ell_{\mathbf{P}}(i)}$, $\widehat{\boldsymbol{\rho}}_i \in \Sigma^{\ell_{\mathbf{V}}(i)}$:
   – Compute the (unique) preimage $(\alpha_1, \dots, \alpha_i)$ of $(\widehat{\boldsymbol{\alpha}}_1, \dots, \widehat{\boldsymbol{\alpha}}_i)$ under $\boldsymbol{\varphi}$:
         $$\alpha_1 \coloneqq {\varphi_1}^{-1}(\widehat{\boldsymbol{\alpha}}_1) \in \mathcal{M}_{\mathbf{P},1}, \dots, \alpha_i \coloneqq {\varphi_i}^{-1}(\widehat{\boldsymbol{\alpha}}_i) \in \mathcal{M}_{\mathbf{P},i}\,.$$
   – Decode the encoded verifier message: $\rho_i \coloneqq \psi_i(\widehat{\boldsymbol{\rho}}_i) \in \mathcal{M}_{\mathbf{V},i}$.
   – Map the salt string to a binary string: $\check{\tau} \coloneqq \mathsf{bin}(\boldsymbol{\tau}) \in \{0,1\}^{\delta_\star}$.
   – Append $(i, (\mathbb{x}, \check{\tau}, (\alpha_1, \dots, \alpha_i)), \rho_i)$ to $\mathsf{tr}_{\mathsf{std}}$.
4. Return $\mathsf{tr}_{\mathsf{std}}$.
```

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

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

Time complexity. D2STrace is a thin wrapper over StdTrace that re-maps the prover and verifer messages using from the alphabet Σ to the prover and verifier messages spaces, via the codec. Let:

- $t_{\boldsymbol{\varphi}^{-1}}$ be the total time to compute each φ_i^{-1} for $i \in [\mathsf{k}]$,
- $t_{\boldsymbol{\psi}^{-1}}$ be the total time to compute each ψ_i^{-1} for $i \in [\mathsf{k}]$,
- $t_{\mathsf{bin}} = \delta \log |\Sigma|$ be the time to compute the binary encoding of the input salt, consisting of δ symbols in Σ.

Let $t_{\mathsf{BackTrack}}$ and $t_{\mathsf{LookAhead}}$ be the times to compute BackTrack and LookAhead. Then

$$\begin{aligned}
&t_{\mathsf{D2STrace}} \\
&= O\left(t_{\mathsf{StdTrace}} + \left|\widehat{\mathsf{tr}_{\mathsf{std}}}\right| \cdot (t_{\boldsymbol{\psi}^{-1}} + t_{\boldsymbol{\varphi}^{-1}} + t_{\mathsf{bin}})\right) \\
&= O\left(|\mathsf{tr}| \cdot (t_{\mathsf{BackTrack}} + t_{\mathsf{LookAhead}} + t_{\mathrm{std}}) + \left|\widehat{\mathsf{tr}_{\mathsf{std}}}\right| \cdot (t_{\boldsymbol{\psi}^{-1}} + t_{\boldsymbol{\varphi}^{-1}} + t_{\mathsf{bin}})\right) && \text{(by Equation 20)} \\
&= O\left(|\mathsf{tr}| \cdot (t_{\mathsf{BackTrack}} + t_{\mathsf{LookAhead}} + t_{\mathrm{std}}) + t_{\boldsymbol{\psi}^{-1}} + t_{\boldsymbol{\varphi}^{-1}} + \delta \log |\Sigma|\right)\,. && \text{(since } \left|\widehat{\mathsf{tr}_{\mathsf{std}}}\right| \leq |\mathsf{tr}|\text{)}
\end{aligned}$$

5.6 Bad events

Definition 5.5. *An entry* tr_j *in* tr *is* **redundant** *if one of the following holds:*

$$\mathsf{tr}_j = (\text{‘}h\text{’}, \mathbb{x}, \boldsymbol{s}_{\mathrm{C}}) \implies \exists\, j' < j\colon\ \mathsf{tr}_{j'} = (\text{‘}h\text{’}, \mathbb{x}, \boldsymbol{s}_{\mathrm{C}})\,, \tag{21}$$

$$\mathsf{tr}_j = (\text{‘}p\text{’}, \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}) \implies \exists\, j' < j\colon\ \mathsf{tr}_{j'} = (\text{‘}p\text{’}, \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}) \lor \mathsf{tr}_{j'} = (\text{‘}p^{-1}\text{’}, \boldsymbol{s}_{\mathrm{out}}, \boldsymbol{s}_{\mathrm{in}})\,, \tag{22}$$

$$\mathsf{tr}_j = (\text{‘}p^{-1}\text{’}, \boldsymbol{s}_{\mathrm{out}}, \boldsymbol{s}_{\mathrm{in}}) \implies \exists\, j' < j\colon\ \mathsf{tr}_{j'} = (\text{‘}p^{-1}\text{’}, \boldsymbol{s}_{\mathrm{out}}, \boldsymbol{s}_{\mathrm{in}}) \lor \mathsf{tr}_{j'} = (\text{‘}p\text{’}, \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}})\,. \tag{23}$$

Definition 5.6. *The* **base trace** $\bar{\mathsf{tr}}$ *of* tr *is obtained by removing all redundant entries.*

Definition 5.7. *The predicate* $E(\mathsf{tr})$ *checks if one of the following holds for the query-answer trace* tr.

- *There exists an output capacity segment in the base trace $\bar{\mathsf{tr}}$ that appears as a prior input/output capacity segment. Formally, one of the following predicates holds:*

$$E_h(\mathsf{tr}) \coloneqq \exists\, j > 0, \boldsymbol{s}_{\mathrm{C}} \in \Sigma^c : \bar{\mathsf{tr}}_j = (\text{‘}h\text{’}, \cdot, \boldsymbol{s}_{\mathrm{C}}) \textit{ and } \exists\, j' < j : \begin{array}{l} \bar{\mathsf{tr}}_{j'} = (\text{‘}h\text{’}, \cdot, \boldsymbol{s}_{\mathrm{C}}) \\ \vee\, \bar{\mathsf{tr}}_{j'} = (\text{‘}p\text{’}, \cdot, (\cdot, \boldsymbol{s}_{\mathrm{C}})) \\ \vee\, \bar{\mathsf{tr}}_{j'} = (\text{‘}p^{-1}\text{’}, \cdot, (\cdot, \boldsymbol{s}_{\mathrm{C}})) \\ \vee\, \bar{\mathsf{tr}}_{j'} = (\text{‘}p\text{’}, (\cdot, \boldsymbol{s}_{\mathrm{C}}), \cdot) \\ \vee\, \bar{\mathsf{tr}}_{j'} = (\text{‘}p^{-1}\text{’}, (\cdot, \boldsymbol{s}_{\mathrm{C}}), \cdot) \end{array} , \tag{24}$$

$$E_p(\mathsf{tr}) \coloneqq \exists\, j > 0, \boldsymbol{s}_{\mathrm{C}} \in \Sigma^c : \bar{\mathsf{tr}}_j = (\text{‘}p\text{’}, \cdot, (\cdot, \boldsymbol{s}_{\mathrm{C}})) \textit{ and } \begin{array}{l} \exists\, j' < j : \bar{\mathsf{tr}}_{j'} = (\text{‘}h\text{’}, \cdot, \boldsymbol{s}_{\mathrm{C}}) \\ \vee\, \exists\, j' < j : \bar{\mathsf{tr}}_{j'} = (\text{‘}p\text{’}, \cdot, (\cdot, \boldsymbol{s}_{\mathrm{C}})) \\ \vee\, \exists\, j' < j : \bar{\mathsf{tr}}_{j'} = (\text{‘}p^{-1}\text{’}, \cdot, (\cdot, \boldsymbol{s}_{\mathrm{C}})) \\ \vee\, \exists\, j' \le j : \bar{\mathsf{tr}}_{j'} = (\text{‘}p\text{’}, (\cdot, \boldsymbol{s}_{\mathrm{C}}), \cdot) \\ \vee\, \exists\, j' \le j : \bar{\mathsf{tr}}_{j'} = (\text{‘}p^{-1}\text{’}, (\cdot, \boldsymbol{s}_{\mathrm{C}}), \cdot) \end{array} , \tag{25}$$

$$E_{p^{-1}}(\mathsf{tr}) \coloneqq \exists\, j > 0, \boldsymbol{s}_{\mathrm{C}} \in \Sigma^c : \bar{\mathsf{tr}}_j = (\text{‘}p^{-1}\text{’}, \cdot, (\cdot, \boldsymbol{s}_{\mathrm{C}})) \textit{ and } \begin{array}{l} \exists\, j' < j : \bar{\mathsf{tr}}_{j'} = (\text{‘}h\text{’}, \cdot, \boldsymbol{s}_{\mathrm{C}}) \\ \vee\, \exists\, j' < j : \bar{\mathsf{tr}}_{j'} = (\text{‘}p\text{’}, \cdot, (\cdot, \boldsymbol{s}_{\mathrm{C}})) \\ \vee\, \exists\, j' < j : \bar{\mathsf{tr}}_{j'} = (\text{‘}p^{-1}\text{’}, \cdot, (\cdot, \boldsymbol{s}_{\mathrm{C}})) \\ \vee\, \exists\, j' \le j : \bar{\mathsf{tr}}_{j'} = (\text{‘}p\text{’}, (\cdot, \boldsymbol{s}_{\mathrm{C}}), \cdot) \\ \vee\, \exists\, j' \le j : \bar{\mathsf{tr}}_{j'} = (\text{‘}p^{-1}\text{’}, (\cdot, \boldsymbol{s}_{\mathrm{C}}), \cdot) \end{array} . \tag{26}$$

- *The same query to p leads to different answers, or there are inconsistent queries across p and p^{-1}:*

$$E_{\mathrm{func}}(\mathsf{tr}) \coloneqq \exists\, j > 0, \boldsymbol{s}_{\mathrm{in}} \in \Sigma^{r+c} : \bar{\mathsf{tr}}_j = (\text{‘}p\text{’}, \boldsymbol{s}_{\mathrm{in}}, \cdot) \textit{ and } \exists\, j' < j : \begin{array}{l} \bar{\mathsf{tr}}_{j'} = (\text{‘}p\text{’}, \boldsymbol{s}_{\mathrm{in}}, \cdot) \\ \vee\, \bar{\mathsf{tr}}_{j'} = (\text{‘}p^{-1}\text{’}, \cdot, \boldsymbol{s}_{\mathrm{in}}) \end{array} . \tag{27}$$

$E_{\mathrm{func}}(\mathsf{tr})$ never holds for a permutation p and its inverse p^{-1}, however in the analyses certain probabilistic stateful algorithms may, with small probability, cause such events.

We now bound the probability that this predicate happens, when considering oracles sampled from different distributions.

Lemma 5.8. *For every $(t_h, t_p, t_{p^{-1}})$-query algorithm $\tilde{\mathcal{P}}$ such that $t_p \ge L$,*

$$\max\left\{ \Pr\left[E(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \,\middle|\, \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x}, \pi) \end{array} \right], \Pr\left[E(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \,\middle|\, \begin{array}{l} \boldsymbol{g} \leftarrow \mathcal{D}_{\Sigma}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{g}}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}(\mathbb{x}, \pi) \end{array} \right] \right\}$$
$$\le \frac{7(t_h + 1 + t_p + L + t_{p^{-1}})^2 - 3(t_h + 1 + t_p + L + t_{p^{-1}})}{2\,|\Sigma|^c} .$$

Proof. $\tilde{\mathcal{P}}$ is a $(t_h, t_p, t_{p^{-1}})$-query algorithm, while $\mathcal{V}$ is a $(1, L, 0)$-query algorithm. Therefore, the base trace at the end of the execution of $\tilde{\mathcal{P}}$ and $\mathcal{V}$ has size (in both experiments) at most:

$$\left|\, \mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}} \,\right| \le t_h + 1 + t_p + L + t_{p^{-1}} .$$

For each $j \in [t_h + 1 + t_p + L + t_{p^{-1}}]$, let $E^{(j)}$ be the predicate that checks E for the (fixed) j-th query-answer pair in the base trace $\bar{\mathsf{tr}}$ of $\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}$. Therefore, we can write:

$$E(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) = \bigvee_{j \in [t_h+1+t_p+L+t_{p^{-1}}]} \left(E_h^{(j)}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \vee E_p^{(j)}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \vee E_{p^{-1}}^{(j)}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \vee E_{\mathrm{func}}^{(j)}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \right) .$$

Denote with $E_{\mathfrak{S}}$ the left-hand side experiment of which we are bounding the probability, and with E_{Σ} the right-hand side experiment. Note that, in E_{Σ}, if $\mathsf{D2SQuery}$ aborts then execution stops and the query-answer trace is stopped at that point. By Lemma 5.18, E will over the (partial) query-answer trace.

We bound the probability of the two events using sub-additivity of the function max and the probability measure, studying each sub-predicate below. We have:

$$\begin{aligned} &\max\{\Pr[E_{\mathfrak{S}}], \Pr[E_{\Sigma}]\} \\ &\leq \sum_{j \in [t_h+1+t_p+L+t_{p^{-1}}]} \max\left\{ \Pr\left[E_{\mathfrak{S}}^{(j)}\right], \Pr\left[E_{\Sigma}^{(j)}\right] \right\} , \end{aligned} \tag{28}$$

where each term is defined as follows:

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

$$\Pr\left[E_{\Sigma,h}^{(j)}\right] := \Pr\left[E_h^{(j)}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \;\middle|\; \begin{array}{l} \boldsymbol{g} \leftarrow \mathcal{D}_{\Sigma}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{g}}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}(\mathbb{x}, \pi) \end{array} \right] ,$$

and similarly for the remaining terms. Each sub-predicate is studied separately.

Predicate E_h (Equation 24). This predicate checks that an answer from h previously appears in the base trace $\bar{\mathsf{tr}}$ as an answer of h, an answer of p (or p^{-1}), or a query to p (or p^{-1}).
In both events, h's answers are sampled uniformly from Σ^c:

- In $E_{\mathfrak{S}}$, the oracle h is sampled uniformly from $\mathcal{U}(\{0,1\}^{\leq n} \to \Sigma^c)$, and so its output distribution is uniform;
- In E_{Σ} $\mathsf{D2SQuery}$ responds to queries to h performing lazy sampling in Item 2, and its outputs are uniform in Σ^c.

Before the j-th query, at most $(j-1)$ output capacity segments and $(j-1)$ input capacity segments appear in the base trace $\bar{\mathsf{tr}}$. Therefore:

$$\max\left\{ \Pr\left[E_{\mathfrak{S},h}^{(j)}\right], \Pr\left[E_{\Sigma,h}^{(j)}\right] \right\} \leq \frac{2(j-1)}{|\Sigma|^c} . \tag{29}$$

Predicate E_p (Equation 25). This predicate checks that an answer from p previously appears in the base trace $\bar{\mathsf{tr}}$ as an answer from h, an answer from p (or p^{-1}), or a query to p (or p^{-1}). In both events p's answers are uniformly distributed:

- In $E_{\mathfrak{S}}$, p is a random permutation over Σ^{r+c} and therefore its output is uniformly distributed.
- In E_{Σ}, $\mathsf{D2SQuery}$ responds to queries to p as follows:
  - In Items 4(c)iii and 4(d)ii they are sampled uniformly at random;

– In Items 4(c)i and 4(e)iiiE, the answer capacity segment is uniformly distributed, and the rate segment is an r-chunk of an answer by $\boldsymbol{g}$, and thus uniformly distributed as well.
– In Items 4(c)ii, 4(d)i and 4(e)ii the output is chosen to be consistent with a previous query to p or p^{-1}. Fresh queries to p^{-1} are distributed uniformly in Σ^{r+c} (cf. Item 3b).

Before the j-th query index at most $(j-1)$ output capacity segments appear in the base trace $\bar{\mathsf{tr}}$, and at most j input capacity segments (including the j-th input segment) in the base trace $\bar{\mathsf{tr}}$. Therefore:

$$\max\left\{\Pr\left[E^{(j)}_{\mathfrak{S},p}\right], \Pr\left[E^{(j)}_{\Sigma,p}\right]\right\} \leq \frac{2j-1}{|\Sigma|^c} . \tag{30}$$

Predicate $\boldsymbol{E_{p^{-1}}}$ (Equation 26). A similar reasoning as above applies to p^{-1} and $E_{p^{-1}}$, yielding

$$\max\left\{\Pr\left[E^{(j)}_{\mathfrak{S},p^{-1}}\right], \Pr\left[E^{(j)}_{\Sigma,p^{-1}}\right]\right\} \leq \frac{2j-1}{|\Sigma|^c} . \tag{31}$$

Predicate $\boldsymbol{E_{\text{func}}}$ (Equation 27). In $E_{\mathfrak{S}}$, the oracles p, p^{-1} are functions so the event E_{func} never holds for query-answer traces generated by $\tilde{\mathcal{P}}\|\mathcal{V}$. In E_Σ, instead, while all queries to p^{-1} are answered consistently, it may happen that, if the j-th query is to p with input $\boldsymbol{s}_{\text{in}}$, in Item 4(c)i we match an entry in Cache_p and never hit Item 4(c)ii, thus responding inconsistently to the j-th query. In this case, the resulting base trace $\bar{\mathsf{tr}}$ there exist $\boldsymbol{s}_{\text{out}} \neq \boldsymbol{s}'_{\text{out}}$ in Σ^{r+c} such that

$$\bar{\mathsf{tr}}_j = (\text{‘}p\text{’}, \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \text{ and } \exists\, j' < j : \begin{array}{l} \bar{\mathsf{tr}}_{j'} = (\text{‘}p\text{’}, \boldsymbol{s}_{\text{in}}, \boldsymbol{s}'_{\text{out}}) \\ \vee\, \bar{\mathsf{tr}}_{j'} = (\text{‘}p^{-1}\text{’}, \boldsymbol{s}'_{\text{out}}, \boldsymbol{s}_{\text{in}}) \end{array} , \tag{32}$$

making E_{func} hold. In particular, this means that after the j-th query is made, one of the following holds:

$$\begin{array}{l} (\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \mathsf{Cache}_p^{<j} \ \wedge\ (\text{‘}p\text{’}, \boldsymbol{s}_{\text{in}}, \boldsymbol{s}'_{\text{out}}) \in \mathsf{tr}^{<j} \text{ , or} \\ (\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \mathsf{Cache}_p^{<j} \ \wedge\ (\text{‘}p^{-1}\text{’}, \boldsymbol{s}'_{\text{out}}, \boldsymbol{s}_{\text{in}}) \in \mathsf{tr}^{<j} \text{ ,} \end{array} \tag{33}$$

where $\mathsf{tr}^{<j}$, $\mathsf{Cache}_p^{<j}$ are the states of the variables tr and Cache_p before the j-th query.

All pairs in Cache_p are sampled at random from $\Sigma^{r+c} \times \Sigma^{r+c}$. For every query to p for which $\mathsf{BackTrack}$ returns a tuple of the form $(i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i))$ (see Item 4e), $L_{\mathbf{V}}(i) - 1$ elements are added to Cache_p (see Item 4(e)iiiD). Recall that $L := \sum_{i \in [\mathsf{k}]} L_{\mathbf{P}}(i) + L_{\mathbf{V}}(i)$ and $L_{\mathbf{V}} := \max_{i \in [\mathsf{k}]} L_{\mathbf{V}}(i)$. Then:

$$\begin{aligned} &\max\left\{\Pr\left[E^{(j)}_{\mathfrak{S},\text{func}}\right], \Pr\left[E^{(j)}_{\Sigma,\text{func}}\right]\right\} \qquad (34)\\ &= \Pr\left[E^{(j)}_{\Sigma,\text{func}}\right] \\ &\leq \Pr\left[\begin{array}{l|l} \exists\, \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}, \boldsymbol{s}'_{\text{out}} \in \Sigma^{r+c} : & \boldsymbol{g} \leftarrow \mathcal{D}_\Sigma(\lambda, n) \\ (\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \mathsf{Cache}_p^{<j} & (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{g}}} \\ \wedge\, (\text{‘}p\text{’}, \boldsymbol{s}_{\text{in}}, \boldsymbol{s}'_{\text{out}}) \in \mathsf{tr}^{<j} & b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}(\mathbb{x}, \pi) \\ \wedge\, \boldsymbol{s}_{\text{out}} \neq \boldsymbol{s}'_{\text{out}} & \end{array}\right] \\ &\quad + \Pr\left[\begin{array}{l|l} \exists\, \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}, \boldsymbol{s}'_{\text{out}} \in \Sigma^{r+c} : & \boldsymbol{g} \leftarrow \mathcal{D}_\Sigma(\lambda, n) \\ (\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}) \in \mathsf{Cache}_p^{<j} & (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{g}}} \\ \wedge\, (\text{‘}p^{-1}\text{’}, \boldsymbol{s}'_{\text{out}}, \boldsymbol{s}_{\text{in}}) \in \mathsf{tr}^{<j} & b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}(\mathbb{x}, \pi) \\ \wedge\, \boldsymbol{s}_{\text{out}} \neq \boldsymbol{s}'_{\text{out}} & \end{array}\right] \end{aligned}$$

$$\leq \frac{\left|\mathsf{Cache}_p^{<j} \cap \mathsf{tr}^{<j}\right|}{|\Sigma|^{r+c}} + \frac{\left|\mathsf{Cache}_p^{<j} \cap \mathsf{tr}^{<j}\right|}{|\Sigma|^{r+c}}$$
$$\leq 2 \cdot \frac{\min\{(j-1)L_{\mathbf{V}}, (j-1)\}}{|\Sigma|^{r+c}}$$
$$= \frac{2(j-1)}{|\Sigma|^{r+c}} \quad \text{(since } L_{\mathbf{V}} \geq 1\text{)}$$
$$\leq \frac{j-1}{|\Sigma|^{c}},$$

since $\left|\Sigma^{r+c}\right| \geq 2$.

Conclusion. Wrapping up, we have:

$$\max\left\{ \Pr\left[E(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \;\middle|\; \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ (\mathbb{x},\pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x},\pi) \end{array} \right], \Pr\left[E(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \;\middle|\; \begin{array}{l} \boldsymbol{g} \leftarrow \mathcal{D}_{\Sigma}(\lambda,n) \\ (\mathbb{x},\pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{g}}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}(\mathbb{x},\pi) \end{array} \right] \right\} \tag{35}$$

$$\leq \sum_{j \in [t_h+1+t_p+L+t_{p^{-1}}]} \Big(\max\Big\{\Pr\Big[E_{\mathfrak{S},h}^{(j)}\Big], \Pr\Big[E_{\Sigma,h}^{(j)}\Big]\Big\} + \max\Big\{\Pr\Big[E_{\mathfrak{S},p}^{(j)}\Big], \Pr\Big[E_{\Sigma,p}^{(j)}\Big]\Big\}$$
$$+ \max\Big\{\Pr\Big[E_{\mathfrak{S},p^{-1}}^{(j)}\Big], \Pr\Big[E_{\Sigma,p^{-1}}^{(j)}\Big]\Big\} + \max\Big\{\Pr\Big[E_{\mathfrak{S},\mathrm{func}}^{(j)}\Big], \Pr\Big[E_{\Sigma,\mathrm{func}}^{(j)}\Big]\Big\}\Big) \quad \text{(by Equation 28)}$$
$$\leq \sum_{j \in [t_h+1+t_p+L+t_{p^{-1}}]} \Big(\frac{2(j-1)}{|\Sigma|^c} + \frac{2j-1}{|\Sigma|^c} + \frac{2j-1}{|\Sigma|^c} + \frac{j-1}{|\Sigma|^c} \Big) \quad \text{(by Eqs. 29 to 34)}$$
$$= \frac{7(t_h+1+t_p+L+t_{p^{-1}})^2 - 3(t_h+1+t_p+L+t_{p^{-1}})}{2\,|\Sigma|^c}. \qquad \square$$

Definition 5.9. *Given a (h,p,p^{-1})-trace tr, we define $E_{\mathrm{prp}}(\mathsf{tr})$ be the disjunction of the following predicates.*
1. $E_{\mathrm{col},p}(\mathsf{tr})$: there exist $(\text{`}p\text{'}, \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}), (\text{`}p\text{'}, \boldsymbol{s}'_{\mathrm{in}}, \boldsymbol{s}'_{\mathrm{out}}) \in \bar{\mathsf{tr}}$ such that $\boldsymbol{s}_{\mathrm{in}} \neq \boldsymbol{s}'_{\mathrm{in}}$ and $\boldsymbol{s}_{\mathrm{out}} = \boldsymbol{s}'_{\mathrm{out}}$.
2. $E_{\mathrm{col},p^{-1}}(\mathsf{tr})$: there exist $(\text{`}p^{-1}\text{'}, \boldsymbol{s}_{\mathrm{out}}, \boldsymbol{s}_{\mathrm{in}}), (\text{`}p^{-1}\text{'}, \boldsymbol{s}'_{\mathrm{out}}, \boldsymbol{s}'_{\mathrm{in}}) \in \bar{\mathsf{tr}}$ with $\boldsymbol{s}_{\mathrm{out}} \neq \boldsymbol{s}'_{\mathrm{out}}$ and $\boldsymbol{s}_{\mathrm{in}} = \boldsymbol{s}'_{\mathrm{in}}$.
3. $E_{\mathrm{col},p,p^{-1}}(\mathsf{tr})$: there exist $(\text{`}p\text{'}, \boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}), (\text{`}p^{-1}\text{'}, \boldsymbol{s}'_{\mathrm{out}}, \boldsymbol{s}'_{\mathrm{in}}) \in \bar{\mathsf{tr}}$ with $\boldsymbol{s}_{\mathrm{out}} = \boldsymbol{s}'_{\mathrm{out}}$ and $\boldsymbol{s}_{\mathrm{in}} \neq \boldsymbol{s}'_{\mathrm{in}}$.
4. $E_{\mathrm{col},p^{-1},p}(\mathsf{tr})$: there exist $(\text{`}p^{-1}\text{'}, \boldsymbol{s}_{\mathrm{out}}, \boldsymbol{s}_{\mathrm{in}}), (\text{`}p\text{'}, \boldsymbol{s}'_{\mathrm{in}}, \boldsymbol{s}'_{\mathrm{out}}) \in \bar{\mathsf{tr}}$ with $\boldsymbol{s}_{\mathrm{out}} \neq \boldsymbol{s}'_{\mathrm{out}}$ and $\boldsymbol{s}_{\mathrm{in}} = \boldsymbol{s}'_{\mathrm{in}}$.

Informally, if Item 1 or Item 3 hold, then p is onto and p^{-1} is not a function; if Item 2 or Item 4 hold, then p^{-1} is onto and p is not a function. We rely on the base trace to avoid ambiguity on what query triggered the collision in redundant queries.

Lemma 5.10. *Let tr be a (h,p,p^{-1})-trace. If $E(\mathsf{tr}) = 0$, then $E_{\mathrm{prp}}(\mathsf{tr}) = 0$.*

Definition 5.11. *Let tr be a (h,p,p^{-1})-trace and $\boldsymbol{s} \in \Sigma^{r+c}$ be a permutation state. Let $E_{\mathrm{inv}}(\mathsf{tr},\boldsymbol{s})$ be the predicate that checks if there is a sequence of permutation states $S^{(k)} \in \mathcal{S}_{\mathsf{BT}}(\mathsf{tr},\boldsymbol{s})$ constructed using p^{-1}:*

$$\exists\, J^{(k)} = (j_h^{(k)}, j_0^{(k)}, j_1^{(k)}, \ldots, j_{m_k}^{(k)}) \in \mathcal{J}_{\mathsf{BT}}(\mathsf{tr},\boldsymbol{s}) : \exists\, \iota \in [0, m_k - 1] \text{ s.t. } \mathsf{tr}_{j_\iota^{(k)}} = (\text{`}p^{-1}\text{'}, \cdot, \cdot)\,. \tag{36}$$

In particular the above condition has the following two sub-cases (which we refer to later on).

- *An inversion after a query to h:*

$$\exists\, J^{(k)} = (j_h^{(k)}, j_0^{(k)}, j_1^{(k)}, \dots, j_{m_k}^{(k)}) \in \mathcal{J}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s}) : \mathsf{tr}_{j_0^{(k)}} = (\text{`}p^{-1}\text{'}, \cdot, \cdot)\,. \tag{37}$$

- *An inversion after a query to p:*

$$\exists\, J^{(k)} = (j_h^{(k)}, j_0^{(k)}, j_1^{(k)}, \dots, j_{m_k}^{(k)}) \in \mathcal{J}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s}) : \exists\, \iota \in [m_k - 1] \text{ s.t. } \mathsf{tr}_{j_\iota^{(k)}} = (\text{`}p^{-1}\text{'}, \cdot, \cdot)\,. \tag{38}$$

Lemma 5.12. *Let tr be a (h, p, p^{-1})-trace and $\boldsymbol{s} \in \Sigma^{r+c}$ be a permutation state. If $E(\mathsf{tr}) = 0$, then $E_{\text{inv}}(\mathsf{tr}, \boldsymbol{s}) = 0$.*

Definition 5.13. *Let tr be a (h, p, p^{-1})-trace and $\boldsymbol{s} \in \Sigma^{r+c}$ be a permutation state. Let $E_{\text{fork}}(\mathsf{tr}, \boldsymbol{s})$ be the predicate that checks if there is a (capacity segment) collision for h or p: $|\mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s})| > 1$. We distinguish several (exhaustive) special cases of this predicate.*

- *Collisions of two outputs of h:*

$$E_{\text{fork},h}(\mathsf{tr}, \boldsymbol{s}) \coloneqq \exists\, S^{(1)}, S^{(2)} \in \mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s}) : \mathbb{x}^{(1)} \neq \mathbb{x}^{(2)} \wedge \boldsymbol{s}_{\text{C,in},0}^{(1)} = \boldsymbol{s}_{\text{C,in},0}^{(2)}\,. \tag{39}$$

- *Capacity segment collisions of two outputs of p:*

$$E_{\text{fork},p}(\mathsf{tr}, \boldsymbol{s}) \coloneqq \begin{array}{l} \exists\, S^{(1)}, S^{(2)} \in \mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s}), \iota_1 \in [0, m_1 - 1], \iota_2 \in [0, m_2 - 1] : \\ \boldsymbol{s}_{\text{in},\iota_1}^{(1)} \neq \boldsymbol{s}_{\text{in},\iota_2}^{(2)} \wedge \boldsymbol{s}_{\text{C,out},\iota_1}^{(1)} = \boldsymbol{s}_{\text{C,out},\iota_2}^{(2)} \end{array}\,. \tag{40}$$

- *Collisions of h with the output capacity segment of a query to p:*

$$E_{\text{fork},h,p}(\mathsf{tr}, \boldsymbol{s}) \coloneqq \exists\, S^{(1)}, S^{(2)} \in \mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s}), \iota \in [m_2 - 1] : \boldsymbol{s}_{\text{C,in},0}^{(1)} = \boldsymbol{s}_{\text{C,out},\iota}^{(2)}\,. \tag{41}$$

Lemma 5.14. *Let tr be a (h, p, p^{-1})-trace and $\boldsymbol{s} \in \Sigma^{r+c}$ be a permutation state. If $E(\mathsf{tr}) = 0$, then $E_{\text{fork}}(\mathsf{tr}, \boldsymbol{s}) = 0$.*

Definition 5.15. $E_{\text{time}}(\mathsf{tr}, \boldsymbol{s}) \coloneqq E_{\text{time},h}(\mathsf{tr}, \boldsymbol{s}) \vee E_{\text{time},p}(\mathsf{tr}, \boldsymbol{s})$ *checks if there is an index list $J^{(k)} \in \mathcal{J}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s})$ out of order.*

- $E_{\text{time},h}(\mathsf{tr}, \boldsymbol{s})$*: the query to h is out of order, i.e.,*

$$\exists\, J^{(k)} = (j_h^{(k)}, j_0^{(k)}, j_1^{(k)}, \dots, j_{m_k}^{(k)}) \in \mathcal{J}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s}) : j_h^{(k)} > j_0^{(k)}\,. \tag{42}$$

- $E_{\text{time},p}(\mathsf{tr}, \boldsymbol{s})$*: a query to p is out of order, i.e.,*

$$\exists\, J^{(k)} = (j_h^{(k)}, j_0^{(k)}, j_1^{(k)}, \dots, j_{m_k}^{(k)}) \in \mathcal{J}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s}) : \exists\, \iota \in [m_k - 1] \text{ s.t. } j_{\iota-1}^{(k)} > j_\iota^{(k)}\,. \tag{43}$$

Lemma 5.16. *Let tr be a (h, p, p^{-1})-query answer trace, and $\boldsymbol{s} \in \Sigma^{r+c}$ be a permutation state. If $E(\mathsf{tr}) = 0$, then $E_{\text{time}}(\mathsf{tr}, \boldsymbol{s}) = 0$.*

5.7 Analysis of aborts

We prove (the contrapositive of) two implications that we use in Section 5.8.
- Lemma 5.17: if $\mathsf{StdTrace}$ aborts, then $E(\mathsf{tr})$ holds.
- Lemma 5.18: if $(\tilde{\mathcal{P}}\|\mathcal{V})^{\mathsf{D2SQuery}}$ aborts then $E(\mathsf{tr})$ holds.

Lemma 5.17. *For every (h, p, p^{-1})-trace tr, if $E(\mathsf{tr}) = 0$ then $\mathsf{StdTrace}(\mathsf{tr})$ does not abort.*

Proof. $\mathsf{StdTrace}$ aborts in one of the following (mutually-exclusive) cases.

- In Item 4(a)ii, which checks that $\mathsf{BackTrack}(\mathsf{tr}, \mathsf{tr}_\nabla, \boldsymbol{s}_{\text{in}})$ returns err for an entry $(`p', \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}})$ in tr. Since $E(\mathsf{tr}) = 0$, then by Lemmas 5.10, 5.12 and 5.14 we have $E_{\text{inv}}(\mathsf{tr}, \boldsymbol{s}) = 0$, $E_{\text{prp}}(\mathsf{tr}) = 0$, and $E_{\text{fork}}(\mathsf{tr}, \boldsymbol{s}) = 0$, and then by Claim 5.19 $\mathsf{BackTrack}$ does not return err.

- In Item 4(a)ivC, which checks that $\mathsf{LookAhead}(\mathsf{tr}_p^{\mathsf{LA}}, \boldsymbol{s}_{\text{in}}, i)$ returns err. Since $E(\mathsf{tr}) = 0$, then by Lemma 5.10, $E_{\text{prp}}(\mathsf{tr}) = 0$, and then by Lemma 5.17 $\mathsf{LookAhead}$ does not return err. □

Lemma 5.18. *For every $(t_h, t_p, t_{p^{-1}})$-query algorithm $\mathcal{A}$, let $\mathsf{tr}_{\mathcal{A}}$ be the query-answer trace resulting from the execution of $\mathcal{A}$ with $\mathsf{D2SQuery}$ oracle access (cf. Section 5.4). Since $E(\mathsf{tr}_{\mathcal{A}}) = 0$, then by Lemma 5.18 $\mathcal{A}^{\mathsf{D2SQuery}}$ does not abort.*

Proof. The procedure $\mathsf{D2SQuery}$ aborts in Item 4b (cf. Section 5.4), which checks that $\mathsf{BackTrack}(\mathsf{tr}, \mathsf{tr}_\nabla, \boldsymbol{s})$ returns err. Since $E(\mathsf{tr}) = 0$, then by Lemmas 5.10, 5.12 and 5.14 we have $E_{\text{inv}}(\mathsf{tr}, \boldsymbol{s}) = 0$, $E_{\text{prp}}(\mathsf{tr}) = 0$, and $E_{\text{fork}}(\mathsf{tr}, \boldsymbol{s}) = 0$, and then by Claim 5.19 $\mathsf{BackTrack}$ does not return err. □

Claim 5.19. *Let tr be a query-answer trace for oracles h, p, p^{-1}, and let tr_∇ be its associated data structure (see Section 5.1). Let $\boldsymbol{s} \in \Sigma^{r+c}$ be a permutation state. If $E_{\text{inv}}(\mathsf{tr}, \boldsymbol{s}) = 0$, $E_{\text{prp}}(\mathsf{tr}) = 0$, and $E_{\text{fork}}(\mathsf{tr}, \boldsymbol{s}) = 0$, then $\mathsf{BackTrack}(\mathsf{tr}, \mathsf{tr}_\nabla, \boldsymbol{s}) \neq \mathsf{err}$.*

Proof. If $\mathsf{BackTrack}(\mathsf{tr}, \mathsf{tr}_\nabla, \boldsymbol{s})$ returns err in Item 5, then Outs contains more than one element (see Section 5.2). The size of Outs is bounded by the size of $\mathcal{S}_{\mathsf{BT}}$ (Outs is initialized in Item 1, and new elements are added in Item 4(a)iiiD). We argue that $\mathcal{S}_{\mathsf{BT}}$ contains at most one element by showing that $S^{(1)} = S^{(2)}$ for every $S^{(1)}, S^{(2)} \in \mathcal{S}_{\mathsf{BT}}$.

Consider two elements in $\mathcal{S}_{\mathsf{BT}}$:

$$S^{(1)} = (\mathbb{x}^{(1)}, \boldsymbol{s}^{(1)}_{\text{in},0}, \boldsymbol{s}^{(1)}_{\text{out},0}, \boldsymbol{s}^{(1)}_{\text{in},1}, \boldsymbol{s}^{(1)}_{\text{out},1}, \ldots, \boldsymbol{s}^{(1)}_{\text{in},m_1-1}, \boldsymbol{s}^{(1)}_{\text{out},m_1-1}, \boldsymbol{s}^{(1)}_{\text{in},m_1})$$
$$S^{(2)} = (\mathbb{x}^{(2)}, \boldsymbol{s}^{(2)}_{\text{in},0}, \boldsymbol{s}^{(2)}_{\text{out},0}, \boldsymbol{s}^{(2)}_{\text{in},1}, \boldsymbol{s}^{(2)}_{\text{out},1}, \ldots, \boldsymbol{s}^{(2)}_{\text{in},m_2-1}, \boldsymbol{s}^{(2)}_{\text{out},m_2-1}, \boldsymbol{s}^{(2)}_{\text{in},m_2})$$

We note the following:

- $S^{(1)}, S^{(2)}$ have the same end element. $\boldsymbol{s}^{(1)}_{\text{in},m_1} = \boldsymbol{s}^{(2)}_{\text{in},m_2} = \boldsymbol{s}$. This is by construction of $\mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s})$.

- The two sequences share the last $\min\{m_1, m_2\}$ permutations states.
  Let $\iota \in [0, \min\{m_1, m_2\}-1]$. We show that if $\boldsymbol{s}^{(1)}_{\text{in},m_1-\iota+1} = \boldsymbol{s}^{(2)}_{\text{in},m_2-\iota+1}$, then
$$\boldsymbol{s}^{(1)}_{\text{out},m_1-\iota} = \boldsymbol{s}^{(2)}_{\text{out},m_2-\iota} \text{ and } \boldsymbol{s}^{(1)}_{\text{in},m_1-\iota} = \boldsymbol{s}^{(2)}_{\text{in},m_2-\iota}\,. \tag{44}$$

  By definition of $\mathcal{S}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s})$, we have that the output capacity segment matches the next input capacity segment, that is, $\boldsymbol{s}^{(1)}_{\text{C},\text{out},m_1-\iota} = \boldsymbol{s}^{(1)}_{\text{C},\text{in},m_1-\iota+1} = \boldsymbol{s}^{(2)}_{\text{C},\text{in},m_2-\iota+1} = \boldsymbol{s}^{(2)}_{\text{C},\text{out},m_2-\iota}$.

$E(\mathsf{tr}, \boldsymbol{s}) = 0$ implies that $E_{\text{fork},p}(\mathsf{tr}, \boldsymbol{s}) = 0$ (that is, there are no collisions in the capacity segment, cf. Equation 40). This implies that the inputs must have been identical, i.e. $\boldsymbol{s}^{(1)}_{\text{in},m_1-\iota} = \boldsymbol{s}^{(2)}_{\text{in},m_2-\iota}$. If the inputs are identical, since $E_{\text{prp}}(\mathsf{tr}) = 0$, then the outputs must have been identical, i.e. $\boldsymbol{s}^{(1)}_{\text{out},m_1-\iota} = \boldsymbol{s}^{(2)}_{\text{out},m_2-\iota}$.

- The two sequences have the same length, i.e. $m_1 = m_2$.
  In fact, since $E_{\text{fork},h,p}(\mathsf{tr}) = 0$ (Equation 41), $\boldsymbol{s}^{(1)}_{\text{C},\text{in},0} \neq \boldsymbol{s}^{(2)}_{\text{C},\text{out},\iota}$ for any $\iota > 0$. In particular, we conclude that
  $$\boldsymbol{s}^{(1)}_{\text{in},0} = \boldsymbol{s}^{(2)}_{\text{in},0}\,. \tag{45}$$

- $\mathbb{x}^{(1)} = \mathbb{x}^{(2)}$.
  Since $\boldsymbol{s}^{(1)}_{\text{in},0} = \boldsymbol{s}^{(2)}_{\text{in},0}$, we have that $(`h', \mathbb{x}^{(1)}, \boldsymbol{s}^{(1)}_{\text{C},\text{in},0}), (`h', \mathbb{x}^{(2)}, \boldsymbol{s}^{(1)}_{\text{C},\text{in},0}) \in \mathsf{tr}$, which by Equation 39 implies that $\mathbb{x}^{(1)} = \mathbb{x}^{(2)}$.

□

Note that the proof of Claim 5.19 does not rely on E_{time} (one of the events in E). Later on we use E_{time} to argue that if $\mathsf{BackTrack}(\mathsf{tr}, \mathsf{tr}_{\triangledown}, \boldsymbol{s})$ outputs a tuple $(i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i))$ then this output remains unchanged even if additional elements are added to tr_h and tr_p.

Claim 5.20. *Let* tr *be a* (h, p, p^{-1})*-trace, and* $\mathsf{tr}_{\triangledown}$ *its corresponding data structure (see Section 5.1). If* $E_{\text{prp}}(\mathsf{tr}) = 0$ *then, for every* $i \in [\mathsf{k}]$ *and* $\boldsymbol{s} \in \Sigma^{r+c}$, $\mathsf{LookAhead}(\mathsf{tr}_{\triangledown}.\mathsf{p}, \boldsymbol{s}, i) \neq \mathsf{err}$.

Proof. We prove that, for any $S^{(1)}_{\mathsf{LA}}, S^{(2)}_{\mathsf{LA}} \in \mathcal{S}_{\mathsf{LA}}$ (cf. Equation 14), $S^{(1)}_{\mathsf{LA}} = S^{(2)}_{\mathsf{LA}}$. We have that:

$$\begin{aligned} S^{(1)}_{\mathsf{LA}} &= \left(\boldsymbol{s}^{(1)}_{\text{in},0}, \boldsymbol{s}^{(1)}_{\text{out},0}, \boldsymbol{s}^{(1)}_{\text{in},1}, \boldsymbol{s}^{(1)}_{\text{out},1}, \ldots, \boldsymbol{s}^{(1)}_{\text{in},m_1-1}, \boldsymbol{s}^{(1)}_{\text{out},m_1-1}\right)\,, \\ S^{(2)}_{\mathsf{LA}} &= \left(\boldsymbol{s}^{(2)}_{\text{in},0}, \boldsymbol{s}^{(2)}_{\text{out},0}, \boldsymbol{s}^{(2)}_{\text{in},1}, \boldsymbol{s}^{(2)}_{\text{out},1}, \ldots, \boldsymbol{s}^{(2)}_{\text{in},m_2-1}, \boldsymbol{s}^{(2)}_{\text{out},m_2-1}\right)\,. \end{aligned}$$

Then:

- $\boldsymbol{s}^{(1)}_{\text{in},0} = \boldsymbol{s}^{(2)}_{\text{in},0} = \boldsymbol{s}$. That is, the two sequences share the same prefix. This is by construction of $\mathcal{S}_{\mathsf{LA}}$.

- For every $\iota \in [0, \min\{m_1, m_2\}-1]$, if $\boldsymbol{s}^{(1)}_{\text{in},\iota} = \boldsymbol{s}^{(2)}_{\text{in},\iota}$ then $\boldsymbol{s}^{(1)}_{\text{out},\iota} = \boldsymbol{s}^{(2)}_{\text{out},\iota}$.

  Since $E_{\text{prp}}(\mathsf{tr}) = 0$, then for all elements $(\boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}), (\boldsymbol{s}'_{\text{in}}, \boldsymbol{s}'_{\text{out}}) \in \mathsf{tr}_p$, if $\boldsymbol{s}_{\text{in}} = \boldsymbol{s}'_{\text{in}}$, then $\boldsymbol{s}_{\text{out}} = \boldsymbol{s}'_{\text{out}}$. In fact:

  - For every $(`p', \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}), (`p', \boldsymbol{s}'_{\text{in}}, \boldsymbol{s}'_{\text{out}}) \in \mathsf{tr}$, if $\boldsymbol{s}_{\text{in}} = \boldsymbol{s}'_{\text{in}}$ then $\boldsymbol{s}_{\text{out}} = \boldsymbol{s}'_{\text{out}}$ (cf. Item 1).
  - For every $(`p^{-1}', \boldsymbol{s}_{\text{out}}, \boldsymbol{s}_{\text{in}}), (`p^{-1}', \boldsymbol{s}'_{\text{out}}, \boldsymbol{s}_{\text{in}}) \in \mathsf{tr}$, if $\boldsymbol{s}_{\text{in}} = \boldsymbol{s}'_{\text{in}}$ then $\boldsymbol{s}_{\text{out}} = \boldsymbol{s}'_{\text{out}}$ (cf. Item 2).
  - For every $(`p^{-1}', \boldsymbol{s}_{\text{out}}, \boldsymbol{s}_{\text{in}}), (`p', \boldsymbol{s}_{\text{in}}, \boldsymbol{s}'_{\text{out}}) \in \mathsf{tr}$, if $\boldsymbol{s}_{\text{in}} = \boldsymbol{s}'_{\text{in}}$ then $\boldsymbol{s}_{\text{out}} = \boldsymbol{s}'_{\text{out}}$ (cf. Item 4).[12]

Since, by construction of $\mathcal{S}_{\mathsf{LA}}$, for all $\iota > 0$, $\boldsymbol{s}^{(1)}_{\text{out},\iota-1} = \boldsymbol{s}^{(1)}_{\text{in},\iota}$ and $\boldsymbol{s}^{(2)}_{\text{out},\iota-1} = \boldsymbol{s}^{(2)}_{\text{in},\iota}$, it must be that $S^{(1)}_{\mathsf{LA}} \subseteq S^{(2)}_{\mathsf{LA}}$ or $S^{(2)}_{\mathsf{LA}} \subseteq S^{(1)}_{\mathsf{LA}}$. Since the sequences have maximal length (Item 1e), then $S^{(1)}_{\mathsf{LA}} = S^{(2)}_{\mathsf{LA}}$. Therefore, if $\mathcal{S}_{\mathsf{LA}}$ is not empty, it contains at most one element. In particular, it means that $\mathsf{LookAhead}$ does not output err. □

[12]Note that Item 3 is concerned with different input states, which are not relevant here.

5.8 Proof of Lemma 5.1

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

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 $\mathcal{D}_{\mathfrak{S}}$ (see Definition 4.2).

Hyb_1. In this hybrid we make the following changes:

(A) in line 1 we sample oracles as

$$\boldsymbol{g} := (g_i)_{i \in [\mathsf{k}]} \leftarrow \mathcal{D}_{\Sigma}(\lambda, n) \tag{46}$$

instead of the oracles $(h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n)$ from Hyb_0;

(B) 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;

(C) in line 3 the argument verifier is $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}$ instead of $\mathcal{V}^{h,p}$ from Hyb_0;

(D) 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 5.21. *The statistical distance between Hyb_0 and Hyb_1 is at most*

$$\frac{7(t_h + 1 + t_p + L + t_{p^{-1}})^2 - 3(t_h + 1 + t_p + L + t_{p^{-1}})}{2\,|\Sigma|^c}\,.$$

Proof. We consider the predicate E in Definition 5.7 in both hybrids:

- E_0 denotes the event that $E(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$ holds at the end of Hyb_0;
- E_1 denotes the event that $E(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$ holds at the end of Hyb_1.

We argue that

$$\Delta\left(\mathsf{Hyb}_0 \;\middle|\; \overline{E_0}, \mathsf{Hyb}_1 \;\middle|\; \overline{E_1}\right) = 0\,. \tag{47}$$

We discuss the hybrid changes in Items (A) to (C), and then Item (D).

(i) In Hyb_0, the oracle $h \colon \{0,1\}^{\leq n} \to \Sigma^c$ is a random function, $p \colon \Sigma^{r+c} \to \Sigma^{r+c}$ a random permutation, and p^{-1} the inverse of p. We argue that, in Hyb_1, $\mathsf{D2SQuery}^{\boldsymbol{g}}$ answers identically (if E does not hold). $\mathsf{D2SQuery}^{\boldsymbol{g}}$ answers queries to h randomly and consistently, as in Hyb_0. The crux of the argument is to show that $\mathsf{D2SQuery}^{\boldsymbol{g}}$ answers queries to p, p^{-1} as in Hyb_0. For this it suffices to prove the following.

(a) The (emulated) oracles p, p^{-1} behave as functions (the same inputs yield the same outputs):

$$\forall\, (\text{`}p\text{'}, \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}), (\text{`}p\text{'}, \boldsymbol{s}'_{\text{in}}, \boldsymbol{s}'_{\text{out}}) \in \mathsf{tr} : \boldsymbol{s}_{\text{in}} = \boldsymbol{s}'_{\text{in}} \implies \boldsymbol{s}_{\text{out}} = \boldsymbol{s}'_{\text{out}}\,, \tag{48}$$

$$\forall\, (\text{`}p^{-1}\text{'}, \boldsymbol{s}_{\text{out}}, \boldsymbol{s}_{\text{in}}), (\text{`}p^{-1}\text{'}, \boldsymbol{s}'_{\text{out}}, \boldsymbol{s}'_{\text{in}}) \in \mathsf{tr} : \boldsymbol{s}_{\text{out}} = \boldsymbol{s}'_{\text{out}} \implies \boldsymbol{s}_{\text{in}} = \boldsymbol{s}'_{\text{in}}\,. \tag{49}$$

Equation 48 holds by inspection of $\mathsf{D2SQuery}$: new queries to p are stored in $\mathsf{tr}_{\nabla}.\mathsf{p}$; duplicate queries are answered consistently via $\mathsf{tr}_{\nabla}.\mathsf{p.inlu}$ (cf. Items 3a, 4(c)ii, 4(d)i and 4(e)ii).

Equation 49 holds by inspection of $\mathsf{D2SQuery}$: new queries to p^{-1} are stored in $\mathsf{tr}_{\nabla}.\mathsf{p}$ (cf. Item 3b); duplicate queries are answered consistently via $\mathsf{tr}_{\nabla}.\mathsf{p.outlu}$ (cf. Item 3). Since E does not hold, answers in Cache_p are not in conflict with answers stored in $\mathsf{tr}_{\nabla}.\mathsf{p}$ (cf. Equation 27).

(b) The (emulated) oracles p, p^{-1} are injective functions:

$$\forall\, (\text{`}p\text{'}, \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}), (\text{`}p\text{'}, \boldsymbol{s}'_{\text{in}}, \boldsymbol{s}'_{\text{out}}) \in \mathsf{tr} : \boldsymbol{s}_{\text{out}} = \boldsymbol{s}_{\text{out}'} \implies \boldsymbol{s}_{\text{in}} = \boldsymbol{s}'_{\text{in}}\,, \tag{50}$$

$$\forall\, (\text{`}p^{-1}\text{'}, \boldsymbol{s}_{\text{out}}, \boldsymbol{s}_{\text{in}}), (\text{`}p^{-1}\text{'}, \boldsymbol{s}'_{\text{out}}, \boldsymbol{s}'_{\text{in}}) \in \mathsf{tr} : \boldsymbol{s}_{\text{in}} = \boldsymbol{s}_{\text{in}'} \implies \boldsymbol{s}_{\text{out}} = \boldsymbol{s}'_{\text{out}}\,. \tag{51}$$

$E(\mathsf{tr})$ does not hold so $E_{\text{prp}}(\mathsf{tr})$ does not hold (Lemma 5.10), so:

- $E_{\text{col},p}(\mathsf{tr}) = 0$ (cf. Item 1), which implies Equation 50; and
- $E_{\text{col},p^{-1}}(\mathsf{tr}) = 0$ (cf. Item 2), which implies Equation 51.

(c) The (emulated) oracle p^{-1} behaves as the inverse of the oracle p:

$$\forall\, (\text{`}p\text{'}, \boldsymbol{s}_{\text{in}}, \boldsymbol{s}_{\text{out}}), (\text{`}p^{-1}\text{'}, \boldsymbol{s}'_{\text{out}}, \boldsymbol{s}'_{\text{in}}) \in \mathsf{tr}\colon\ \boldsymbol{s}_{\text{out}} = \boldsymbol{s}'_{\text{out}} \iff \boldsymbol{s}_{\text{in}} = \boldsymbol{s}'_{\text{in}}\,. \tag{52}$$

$E(\mathsf{tr})$ does not hold so $E_{\text{prp}}(\mathsf{tr})$ does not hold (Lemma 5.10), so $E_{\text{col},p,p^{-1}}(\mathsf{tr}) = 0$ (cf. Item 3) and $E_{\text{col},p^{-1},p}(\mathsf{tr}) = 0$ (cf. Item 4), which implies Equation 52.

(d) The (emulated) oracles p and p^{-1} answer random values (given the above constraints).

- Answers for p^{-1} are sampled in Item 3b uniformly at random.
- Answers for p are sampled in the following cases.
  - If $\mathsf{BackTrack}$ outputs a tuple in the image of $\boldsymbol{\varphi}$ (Item 4e), then the answer comes from the output of $\boldsymbol{g}$ (Item 4(e)i) and possibly extra padding (Item 4(e)iiiA).
  - If $\mathsf{BackTrack}$ outputs a tuple not in the image of $\boldsymbol{\varphi}$ (Item 4d), then the answer is sampled at random (Item 4(d)ii).
  - If $\mathsf{BackTrack}$ outputs none and the query is in Cache_p (Item 4(c)i), then the answer was sampled at random in Item 4(e)iiiD for a previous query.
  - If $\mathsf{BackTrack}$ outputs none (Item 4(c)ii) and the query is not in Cache_p, then the answer is sampled at random (Item 4(c)iii).
  - Since E does not happen, then $\mathsf{BackTrack}$ does not return err (Claim 5.19).

(ii) We address the change in Item (D) by arguing that $\mathsf{D2STrace}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$ in Hyb_0 is identically distributed to $(\boldsymbol{\varphi}^{-1}, \boldsymbol{\psi})(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$ in Hyb_1. Since $\mathsf{D2STrace} = (\boldsymbol{\varphi}^{-1}, \boldsymbol{\psi}) \circ \mathsf{StdTrace}$, it suffices to argue that these two traces are identically distributed:

(a) $\mathsf{tr}_{\mathsf{std},0}$, the output of $\mathsf{StdTrace}$ invoked on $\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}$ in Hyb_0; and

(b) $\mathsf{tr}_{\mathsf{std},1}$, the query-answer trace of $(\tilde{\mathcal{P}} \| \mathcal{V})^{\mathsf{D2SQuery}}$ given oracle access to $\boldsymbol{g}$ in Hyb_1.

We argue this by induction on the queries to (h, p, p^{-1}) made by $\tilde{\mathcal{P}}$ and $\mathcal{V}$. The base case, where no oracle query is made, is trivial. For the inductive case, we assume that $\mathsf{tr}_{\mathsf{std},0}$ and $\mathsf{tr}_{\mathsf{std},1}$ are identically distributed in the prefix resulting from the first $j-1$ oracle queries to (h, p, p^{-1}), and argue that $\mathsf{tr}_{\mathsf{std},0}$ and $\mathsf{tr}_{\mathsf{std},1}$ are identically distributed in the prefix resulting from an additional query to (h, p, p^{-1}).

If the j-th query is to h or p^{-1}, then no entries are added to $\mathsf{tr}_{\mathsf{std},0}$ and $\mathsf{tr}_{\mathsf{std},1}$:

- $\mathsf{StdTrace}$ does not add any entries to its output $\mathsf{tr}_{\mathsf{std}}$ (cf. Item 4b); and
- $\mathsf{D2SQuery}$ does not perform any query to $\boldsymbol{g}$ (cf. Items 2 and 3).

In this case, the claim follows by the inductive assumption.

If the j-th query is to p then a new entry may be added to $\mathsf{tr}_{\mathsf{std},0}$ and $\mathsf{tr}_{\mathsf{std},1}$, and we have to carefully establish that the distribution of this entry is identical in the two hybrids.

Denote by $\boldsymbol{s}_{\text{in}} \in \Sigma^{r+c}$ the j-th query (which is to p). Both hybrids invoke the deterministic procedure $\mathsf{BackTrack}$ (defined in Section 5.2), as follows.

- In Hyb_0, $\mathsf{StdTrace}$ runs $\mathsf{BackTrack}$ in Item 4(a)i on (i) a query-answer trace (and its corresponding data structure) resulting from the (entire) execution of $\tilde{\mathcal{P}} \| \mathcal{V}$, and (ii) the permutation state $\boldsymbol{s}_{\text{in}}$ (the j-th query in this trace).

- In Hyb_1, $\mathsf{D2SQuery}$ runs $\mathsf{BackTrack}$ in Item 4a on (i) a query-answer trace (and its corresponding data structure) that contains the first $j-1$ query-answer pairs made by $\tilde{\mathcal{P}}\|\mathcal{V}$, and (ii) the permutation state $\boldsymbol{s}_{\mathrm{in}}$.

Since previous oracle queries and answers are identically distributed, $\boldsymbol{s}_{\mathrm{in}}$ is identically distributed.

Despite the different traces provided given as input to $\mathsf{BackTrack}$ in the two hybrids, the output of $\mathsf{BackTrack}$ is identically distributed: neither E_0 nor E_1 happens, so $E(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}}) = 0$ and, in particular, $E_{\mathrm{time}}(\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}}, \boldsymbol{s}_{\mathrm{in}}) = 0$ by Lemma 5.16. Hence, by Equations 42 and 43, each index list $J^{(k)} \in \mathcal{J}_{\mathsf{BT}}(\mathsf{tr}, \boldsymbol{s})$ associated to the sequence of permutation states $S^{(k)} \in \mathcal{S}_{\mathsf{BT}}$ computed by $\mathsf{BackTrack}$ in Equation 11 is monotonically increasing, with the last index being j. This means that all permutation states used to compute $\mathcal{S}_{\mathsf{BT}}$ appear in the first j-th entries of the query-answer trace $\mathsf{tr}_{\tilde{\mathcal{P}}}\|\mathsf{tr}_{\mathcal{V}}$.

The traces $\mathsf{tr}_{\mathsf{std},0}, \mathsf{tr}_{\mathsf{std},1}$ are left unchanged unless $\mathsf{BackTrack}$ outputs a tuple $(i, \mathbb{x}, \boldsymbol{\tau}, (\widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i))$ such that $\forall\, \iota \in [i]$, $\widehat{\boldsymbol{\alpha}}_\iota \in \mathrm{Im}(\varphi_\iota)$ (see Item 4(a)iv in $\mathsf{StdTrace}$ and Item 4e in $\mathsf{D2SQuery}$). In this case, the tuple $(\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i)$ is appended to $\mathsf{tr}_{\mathsf{std},0}, \mathsf{tr}_{\mathsf{std},1}$ as a "query" for the i-th oracle in $\mathsf{Hyb}_0, \mathsf{Hyb}_1$ respectively. We have argued that the output of $\mathsf{BackTrack}$ is identically distributed, so we are left to argue that the answer added to $\mathsf{tr}_{\mathsf{std},0}$ is identically distributed to the answer added to $\mathsf{tr}_{\mathsf{std},1}$.

If there is a query of the form $(i, (\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i), \cdot)$ already present in $\mathsf{tr}_{\mathsf{std},0}, \mathsf{tr}_{\mathsf{std},1}$, the claim follows by inductive hypothesis:

- in Hyb_0, $\mathsf{StdTrace}$ appends to $\mathsf{tr}_{\mathsf{std},0}$ the tuple $(i, (\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i), \widehat{\boldsymbol{\rho}}_i)$, where $\widehat{\boldsymbol{\rho}}_i$ is the same answer as in a previous query (see Item 4(a)ivA);
- in Hyb_1, $\mathsf{D2SQuery}$ appends to $\mathsf{tr}_{\mathsf{std},1}$ the tuple $(i, (\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i), \widehat{\boldsymbol{\rho}}_i)$, where $\widehat{\boldsymbol{\rho}}_i$ is the output of g_i, which responds consistently to queries with the same input (see Item 4(e)i).

If the query is fresh:

- $\mathsf{StdTrace}$ appends to $\mathsf{tr}_{\mathsf{std},0}$ the tuple $(i, (\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i), \widehat{\boldsymbol{\rho}}_i)$, where $\widehat{\boldsymbol{\rho}}_i$ is the output of $\mathsf{LookAhead}$;
- $\mathsf{D2SQuery}$ appends to $\mathsf{tr}_{\mathsf{std},1}$ the tuple $(i, (\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i), \widehat{\boldsymbol{\rho}}_i)$ where $\widehat{\boldsymbol{\rho}}_i = g_i(\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i)$.

We claim that $\widehat{\boldsymbol{\rho}}_i \in \Sigma^{\ell_{\mathbf{V}}(i)}$ is identically distributed in the two hybrids, by arguing, in each case, that $\widehat{\boldsymbol{\rho}}_i$ is sampled at random and that $\widehat{\boldsymbol{\rho}}_i$ agrees with the query-answer trace.

(a) $\widehat{\boldsymbol{\rho}}_i$ is sampled at random in Hyb_0 and Hyb_1.
In Hyb_1, $\boldsymbol{g}$ is sampled from $\mathcal{D}_\Sigma(\lambda, n)$. In Hyb_0, $\widehat{\boldsymbol{\rho}}_i$ is the output of $\mathsf{LookAhead}$ (cf. Item 2). Note that, since E does not hold, the output is not err (Claim 5.20), and, since $(\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}) \in \mathsf{tr}_p$, $S^{(k)}_{\mathsf{LA}}$ is not empty and thus the output is not none. Hence, $\mathsf{LookAhead}$'s output is indeed a tuple of elements in $\Sigma^{\ell_{\mathbf{V}}(i)}$ which is uniformly distributed (by construction of $\mathsf{LookAhead}$ in Item 2c, and because p is a random permutation sampled from $\mathcal{D}_{\mathfrak{S}}(\lambda, n)$).

(b) $\widehat{\boldsymbol{\rho}}_i$ agrees with the query-answer trace (queries to p that follow $\mathsf{DS.Squeeze}$ are r-"chunks" of $\widehat{\boldsymbol{\rho}}_i$).
Consider all queries to p whose input is $\boldsymbol{s}_{\mathrm{in}}$ and output is of the form $\boldsymbol{s}^\star_{\mathrm{out}} = (\widehat{\boldsymbol{\rho}}_i[0\colon r], \boldsymbol{s}^{(0)}_{\mathrm{C,out}})$ (for some $\boldsymbol{s}^{(0)}_{\mathrm{C,out}} \in \Sigma^c$) made during the execution of $\tilde{\mathcal{P}}\|\mathcal{V}$. There is at least one such query: the j-th query with input $\boldsymbol{s}_{\mathrm{in}}$ has associated answer rate segment the first r elements of $\widehat{\boldsymbol{\rho}}_i$ in both hybrids (see Item 2c in Hyb_0 and Item 4(e)iiiB in Hyb_1). Since E does not hold in either hybrid, then E_{prp} does not hold either (Lemma 5.10), and all queries with input $\boldsymbol{s}_{\mathrm{in}}$ have output $\boldsymbol{s}^\star_{\mathrm{out}}$.

The remaining $L_{\mathbf{V}}(i) - 1$ chunks may be present in the query-answer trace. In Hyb_0, $\widehat{\boldsymbol{\rho}}_i$ is constructed to agree with them (cf. Item 1c in LookAhead). In Hyb_1, D2SQuery stores all chunks in a list Cache_p in Item 4(e)iiiD, and answers them throughout the execution of $\tilde{\mathcal{P}} \| \mathcal{V}$ in Item 4(c)i. Since E does not hold, then by Lemma 5.10 E_{prp} does not hold either (in particular, this means that Equation 33 does not hold), and all query-answer pairs stored in Cache_p consistent with the query-answer trace.

Hence, after the j-th query, the element (if any) added to $\mathsf{tr}_{\mathsf{std},0}$ and $\mathsf{tr}_{\mathsf{std},1}$ is identically distributed.

We conclude that

$$\begin{aligned}
&\Delta\left(\mathsf{Hyb}_0, \mathsf{Hyb}_1\right) \\
&\leq \Delta\left(\mathsf{Hyb}_0 \mid \overline{E_0}, \mathsf{Hyb}_1 \mid \overline{E_1}\right) + \max\{\Pr[E_0], \Pr[E_1]\} && \text{(by [CY24, Claim 1.2.10])} \\
&\leq 0 + \max\{\Pr[E_0], \Pr[E_1]\} && \text{(by Equation 47)} \\
&\leq \frac{7(t_h + 1 + t_p + L + t_{p^{-1}})^2 - 3(t_h + 1 + t_p + L + t_{p^{-1}})}{2\,|\Sigma|^c} && \text{(by Lemma 5.8)}\,.
\end{aligned}$$

□

Hyb_2. In this hybrid we make the following changes:

- in line 1 we sample oracles as

$$\boldsymbol{e} := (e_i)_{i \in [\mathsf{k}]} \leftarrow \mathcal{U}\left((\{0,1\}^{\leq n} \times \Sigma^{\delta} \times \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)} \to \mathcal{M}_{\mathbf{V},i})_{i \in [\mathsf{k}]}\right) \tag{53}$$

  instead of the oracles $\boldsymbol{g}$ from Hyb_1 (see Equation 46);
- 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 $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}$ 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 5.22. *The statistical distance between Hyb_1 and Hyb_2 is at most*

$$\theta_{\star}(t_h, t_p, t_{p^{-1}}) \cdot \max_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n)\,.$$

Proof. The hybrids differ in that in Hyb_1 the oracle g_i outputs an element in $\Sigma^{\ell_{\mathbf{V}}(i)}$, whereas in Hyb_2 the oracle e_i outputs an element in $\mathcal{M}_{\mathbf{V},i}$. The input domain is the same across hybrids, since both g_i and e_i take inputs in $\{0,1\}^{\leq n} \times \Sigma^{\delta} \times \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)}$. The output domain difference is reconciled in two ways.

- In lines 2 and 3, D2SQuery receives oracle access to $\boldsymbol{\psi}^{-1} \circ \boldsymbol{e}$, which means that, for each $i \in [\mathsf{k}]$, outputs of e_i are mapped by the pre-image sampling procedure ψ_i^{-1} from $\mathcal{M}_{\mathbf{V},i}$ to $\Sigma^{\ell_{\mathbf{V}}(i)}$.

  Let $i \in [\mathsf{k}]$ be a round index. For $\boldsymbol{x} \in \Sigma^{\ell_{\mathbf{V}}(i)}$, let $\bar{\boldsymbol{x}} := \left\{\boldsymbol{y} \in \Sigma^{\ell_{\mathbf{V}}(i)} \;\middle|\; \psi(\boldsymbol{y}) = \psi(\boldsymbol{x})\right\}$ be the corresponding elements in the quotient space in $\Sigma^{\ell_{\mathbf{V}}(i)}$. The statistical distance between the distributions of the answer to a query to the i-th oracle in Hyb_1 and Hyb_2 is the following:

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

$$= \frac{1}{2} \sum_{\boldsymbol{x} \in \Sigma^{\ell_{\mathbf{V}}(i)}} \left| \frac{1}{|\Sigma|^{\ell_{\mathbf{V}}(i)}} - \Pr\Big[\boldsymbol{x} = \psi_i^{-1}(\boldsymbol{y}) \;\Big|\; \boldsymbol{y} \leftarrow \mathcal{U}(\mathcal{M}_{\mathbf{V},i})\Big] \right|$$

$$= \frac{1}{2} \sum_{\boldsymbol{x} \in \Sigma^{\ell_{\mathbf{V}}(i)}} \left| \frac{1}{|\Sigma|^{\ell_{\mathbf{V}}(i)}} - \Pr\big[\psi_i(\boldsymbol{x}) = \boldsymbol{y} \;\big|\; \boldsymbol{y} \leftarrow \mathcal{U}(\mathcal{M}_{\mathbf{V},i})\big] \cdot \Pr\big[\boldsymbol{x} = \boldsymbol{x}' \;\big|\; \boldsymbol{x}' \leftarrow \mathcal{U}(\bar{\boldsymbol{x}})\big] \right|$$

$$= \frac{1}{2} \sum_{\boldsymbol{x} \in \Sigma^{\ell_{\mathbf{V}}(i)}} \left| \frac{1}{|\Sigma|^{\ell_{\mathbf{V}}(i)}} - \frac{|\bar{\boldsymbol{x}}|}{|\Sigma|^{\ell_{\mathbf{V}}(i)}} \cdot \frac{1}{|\bar{\boldsymbol{x}}|} \right| = 0\,.$$

- In line 4, answers in the query-answer trace $\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}$ are not translated: they are left as the answers of the oracles $\boldsymbol{e} = (e_i)_{i \in [\mathsf{k}]}$. In contrast, in Hyb_1, answers of the oracles $\boldsymbol{g} = (g_i)_{i \in [\mathsf{k}]}$ in the query-answer trace $\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}$ are mapped via $\boldsymbol{\psi} = (\psi_i \colon \Sigma^{\ell_{\mathbf{V}}(i)} \to \mathcal{M}_{\mathbf{V},i})_{i \in [\mathsf{k}]}$.

  Fix a round index $i \in [\mathsf{k}]$. By Definition 4.1, for every $i \in [\mathsf{k}]$, 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). Hence the statistical distance between the distributions of an answer corresponding to the i-th oracle appearing in the query-answer trace in Hyb_1 and Hyb_2 is the following:

$$\Delta_{\psi_i} :=\Delta \left(\psi_i \circ \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)}), \mathcal{U}(\mathcal{M}_{\mathbf{V},i})\right) = \varepsilon_{\mathsf{cdc},i}(\lambda, n)\,.$$

Each of the $\theta_\star(t_h, t_p, t_{p^{-1}})$ queries made by $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}}$ incurs a statistical error that is at most $\max_{i \in [\mathsf{k}]} \{\Delta_{\psi_i} + \Delta_{\psi_i^{-1}}\}$ (a query can be to any of the k oracles). Moreover, the argument verifier $\mathcal{V}$ makes one query to each oracle, incurring a statistical error that is at most $\sum_{i \in [\mathsf{k}]} \max\{\Delta_{\psi_i} + \Delta_{\psi_i^{-1}}\}$. We conclude that the statistical distance between Hyb_1 and Hyb_2 is at most

$$\begin{aligned} &\Delta\left(\mathsf{Hyb}_1, \mathsf{Hyb}_2\right) \\ &\le \theta_\star(t_h, t_p, t_{p^{-1}}) \cdot \max_{i \in [\mathsf{k}]} \left\{\Delta_{\psi_i} + \Delta_{\psi_i^{-1}}\right\} + \sum_{i \in [\mathsf{k}]} \max\left\{\Delta_{\psi_i} + \Delta_{\psi_i^{-1}}\right\} \\ &= \theta_\star(t_h, t_p, t_{p^{-1}}) \cdot \max_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n)\,. \end{aligned} \tag{54}$$

□

Hyb_3. In this hybrid we make the following changes:

- in line 1 we sample oracles as

$$\boldsymbol{f} := (f_i)_{i \in [\mathsf{k}]} \leftarrow \mathcal{U}\left((\{0,1\}^{\le n} \times \{0,1\}^{\delta_\star} \times \mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i} \to \mathcal{M}_{\mathbf{V},i})_{i \in [\mathsf{k}]}\right) \tag{55}$$

  instead of the oracles $\boldsymbol{e}$ from Hyb_2 (see Equation 53);
- 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 $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}$ instead of $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{e}}}$ 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 5.23. *Hyb_2 and Hyb_3 are identically distributed.*

Proof. The hybrids differ in that in Hyb_2 the prover messages remain encoded (the oracle e_i takes inputs in $\{0,1\}^{\leq n} \times \Sigma^{\delta} \times \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)}$) whereas in Hyb_3 the prover messages are not encoded (the oracle f_i takes inputs in $\{0,1\}^{\leq n} \times \{0,1\}^{\delta_\star} \times \mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i}$). Note that verifier messages are not encoded in both hybrids (the oracles e_i and f_i both output elements in $\mathcal{M}_{\mathbf{V},i}$). The difference is reconciled in two ways.

- In lines 2 and 3, D2SQuery receives oracle access to $\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}$ (compared to $\boldsymbol{\psi}^{-1} \circ \boldsymbol{e}$ in Hyb_2), which means that, for each $i \in [\mathsf{k}]$, inputs of f_i are mapped by the inverse φ_i^{-1} of the encoding map $\varphi_i \colon \mathcal{M}_{\mathbf{P},i} \to \Sigma^{\ell_{\mathbf{P}}(i)}$ to elements in $\mathcal{M}_{\mathbf{P},i}$. (Moreover, salt strings in Σ^{δ} are mapped to their binary representation in $\{0,1\}^{\delta_\star}$.)
- In line 4, the query-answer trace $\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}$ is left unchanged. In contrast, in Hyb_2, queries to the oracles $\boldsymbol{e} = (e_i)_{i \in [\mathsf{k}]}$ in the query-answer trace $\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}$ are mapped via the inverse(s) $\boldsymbol{\varphi}^{-1}$ of $\boldsymbol{\varphi} = (\varphi_i \colon \mathcal{M}_{\mathbf{P},i} \to \Sigma^{\ell_{\mathbf{P}}(i)})_{i \in [\mathsf{k}]}$, and to (unencoded) prover messages (and salt strings in Σ^{δ} are mapped to their binary representation in $\{0,1\}^{\delta_\star}$).

The maps $\boldsymbol{\varphi} = (\varphi_i \colon \mathcal{M}_{\mathbf{P},i} \to \Sigma^{\ell_{\mathbf{P}}(i)})_{i \in [\mathsf{k}]}$ are injective. Similarly, the binary encoding map $\mathsf{bin}(\cdot)$, which is used to encode the salt, is injective. Hence for any input query in

$$\{0,1\}^{\leq n} \times \{0,1\}^{\delta_\star} \times \mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i} ,$$

there exists at most one element in

$$\{0,1\}^{\leq n} \times \Sigma^{\delta} \times \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)}$$

that is associated with it. Therefore, while oracles in Hyb_3 store input queries in another format (using $\boldsymbol{\varphi}^{-1}$ and bin^{-1}), the malicious prover $\tilde{\mathcal{P}}$ cannot distinguish the two hybrids, since the input-output behavior is identical across Hyb_2 and Hyb_3. The same holds for the verifier $\mathcal{V}$ in line 3. We conclude that

$$\Delta\left(\mathsf{Hyb}_2, \mathsf{Hyb}_3\right) = 0 .$$

□

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 55);
- in line 2 the argument prover is run as $\mathsf{D2SAlgo}^{\boldsymbol{f}}(\tilde{\mathcal{P}})$.
- in line 3 the argument verifier is run as $\mathcal{V}_{\mathsf{std}}^{\boldsymbol{f}}$ instead of as $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}$ in Hyb_3; and
- in line 4 the query-answer trace is not translated as in Hyb_3.

Claim 5.24. *The statistical distance between Hyb_3 and Hyb_4 is at most*

$$\frac{7(L+1)(2t_h + 1 + 2t_p + L + 2t_{p^{-1}})}{2\,|\Sigma|^c} - \frac{5(L+1)}{|\Sigma|^c} .$$

Proof. The argument prover $\mathsf{D2SAlgo}^{\boldsymbol{f}}(\tilde{\mathcal{P}})$ equals the argument prover $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}$ in Hyb_3 (see Equation 17). Hence the only potential differences between the hybrids are the argument verifier (line 3) and the query-answer trace (line 4).

Let $E_{\mathcal{V}}$ denote the event in Hyb_3 where $\mathcal{V}^{\mathsf{D2SQuery}}$ aborts but $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}}$ does not. We upper bound the probability of $E_{\mathcal{V}}$ as follows:

$$
\begin{aligned}
&\Pr[E_{\mathcal{V}}] && (56)\\
&= \Pr\left[E(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \;\wedge\; \overline{E(\mathsf{tr}_{\tilde{\mathcal{P}}})} \;\middle|\; \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}_{\mathsf{IP}}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}} \end{array} \right] \\
&\leq \sum_{j = t_h + t_p + t_{p^{-1}} + 1}^{t_h + 1 + t_p + L + t_{p^{-1}}} \left(\frac{2(j-1)}{|\Sigma|^c} + \frac{2j-1}{|\Sigma|^c} + \frac{2j-1}{|\Sigma|^c} + \frac{j-1}{|\Sigma|^c} \right) && \text{(by Equation 35)} \\
&= \frac{7(L+1)(2t_h + 1 + 2t_p + L + 2t_{p^{-1}})}{2\,|\Sigma|^c} - \frac{5(L+1)}{|\Sigma|^c}\,.
\end{aligned}
$$

Next, we argue that the two hybrids are identically distributed if $E_{\mathcal{V}}$ does not happen:

$$
\Delta\left(\mathsf{Hyb}_3 \;\middle|\; \overline{E_{\mathcal{V}}}, \mathsf{Hyb}_4\right) = 0\,. \tag{57}
$$

We discuss changes in lines 1 to 3 in Hyb_4 in turn. Observe that:

- the decision bit of $\mathcal{V}^{\boldsymbol{f}}_{\mathsf{std}}(\mathbb{x}, \pi)$, where $\pi = \big(\tau, (\alpha_i)_{i \in [\mathsf{k}]}\big)$, equals $\mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i \in [\mathsf{k}]}, (\rho_i)_{i \in [\mathsf{k}]}\big)$ with $\rho_i := f_i\big(\mathbb{x}, \tau, \alpha_1, \dots, \alpha_i\big)$ for every $i \in [\mathsf{k}]$;
- the decision bit of $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}}(\mathbb{x}, \pi)$, where $\pi = (\boldsymbol{\tau}, (\alpha_i)_{i \in [\mathsf{k}]})$, equals $\mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i \in [\mathsf{k}]}, (\rho_i)_{i \in [\mathsf{k}]}\big)$ with $\rho_1, \dots, \rho_{\mathsf{k}}$ answered by $\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1} \circ \boldsymbol{f} \circ \boldsymbol{\varphi}^{-1}}$.

We show that, if $E_{\mathcal{V}}$ does not happen, then $\rho_1, \dots, \rho_{\mathsf{k}}$ are identically distributed (when also given all the other quantities in the experiment). We argue this by induction on the round number $i \in [\mathsf{k}]$.

- *Base case:* $i = 1$. The verifier $\mathcal{V}^{\psi_1^{-1} \circ \mathsf{D2SQuery} \circ \varphi_1^{-1}}$ receives as input an instance $\mathbb{x} \in \{0,1\}^{\leq n}$ and a NARG string $\pi = (\boldsymbol{\tau}, (\alpha_i)_{i \in [\mathsf{k}]})$ internally invokes $\mathbf{V}\big(\mathbb{x}, (\alpha_i)_{i \in [\mathsf{k}]}, (\rho_i)_{i \in [\mathsf{k}]}\big)$ where ρ_1 is generated as follows.
  - Invoke $\mathsf{DS.Start}(\mathbb{x})$, which in turn makes an oracle query to h which will internally run $\mathsf{D2SQuery}$, sample a uniformly-random capacity state in Item 2, and produce the initial state $\mathsf{st}_0 := ((0^r, \boldsymbol{s}_{\mathrm{C}}), 0, r)$.
  - Run $\mathsf{DS.Absorb}(\mathsf{st}_0, \boldsymbol{\tau} \| \widehat{\boldsymbol{\alpha}}_1)$ using the encoded prover message $\widehat{\boldsymbol{\alpha}}_1 := \varphi_1(\alpha_1)$. Split the salt and prover message into $L_\delta + L_{\mathbf{P}}(1)$ blocks (all in Σ^r except for the last one, which is in $\Sigma^{\leq r}$). Each query to the permutation function will have use a capacity segment that was not previously input (since $E_{\mathcal{V}}$ does not happen), and $\mathsf{BackTrack}$ will always return none since Item 4 will satisfy the length of candidates in $\mathcal{S}_{\mathsf{BT}}$. By definition of DS, at the end of $\mathsf{DS.Absorb}$, $\mathcal{V}$ produced capacity states $\boldsymbol{s}_{\mathrm{in}}, \boldsymbol{s}_{\mathrm{out}}$ where:
    1. $(\mathbb{x}, \boldsymbol{s}_{\mathrm{C},0})$ is the only element in tr_h;
    2. $\boldsymbol{s}_{\mathrm{in},0} = \boldsymbol{s}_{\mathrm{C},0}$ (i.e., the first capacity segment is the input query), by construction of DS;
    3. $\forall\, \iota \in [0, L_\delta + L_{\mathbf{P}}(1)]$, $(\boldsymbol{s}_{\mathrm{in},\iota}, \boldsymbol{s}_{\mathrm{out},\iota} \in \mathsf{tr}_p$ (i.e., $\boldsymbol{s}_{\mathrm{out},\iota} = p(\boldsymbol{s}_{\mathrm{in},\iota})$ and $\boldsymbol{s}_{\mathrm{C,out},\iota} = \boldsymbol{s}_{\mathrm{C,in},\iota+1}$)

    Since $E_{\mathcal{V}}$ does not happen, E_{fork} does not hold, and thus $\mathcal{S}_{\mathsf{BT}}$ contains exactly one element. Denote with st'_1 the state at the end of the Absorb procedure.

  - Run $\mathsf{DS.Squeeze}^{\psi_1^{-1}\circ\mathsf{D2SQuery}\circ\varphi_1^{-1}}(\mathsf{st}_1', \ell_{\mathbf{V}}(i))$ which will internally run $\mathsf{BackTrack}$ in Item 4a. The backtracking algorithm will recover a unique sequence of permutation states in $\mathcal{S}_{\mathsf{BT}}$ (see Equation 11), since tr contains $L_\delta + L_{\mathbf{P}}(1)$ distinct elements (as E does not happen).

    $\mathsf{BackTrack}$ outputs $(\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1)$ and query $f_1(\mathbb{x}, \mathsf{bin}(\boldsymbol{\tau}), \varphi_1^{-1}(\widehat{\boldsymbol{\alpha}}_1))$ which returns the verifier message ρ_1. Then, $\mathsf{D2SQuery}$ will split $\widehat{\boldsymbol{\rho}}_1 := \psi_1(\rho_1)$ into chunks of r elements (except the last one, which is $\leq r$) and store them in a temporary query-answer trace list Cache_p using capacity states sampled uniformly at random (Item 4(e)iiiD). By construction of DS, the squeeze operation will query those exact locations which are then placed in tr (see Item 4(e)i) and $\mathcal{V}$ reconstructs the encoded verifier message $\widehat{\boldsymbol{\rho}}_1$ (since $\varphi_1^{-1} \circ \varphi_1$ is the identity).

  Therefore, the verifier message is generated as

$$\rho_1 = \psi_1^{-1} \circ \psi_1 \circ f_1(\mathbb{x}, \mathsf{bin}(\boldsymbol{\tau}), \varphi_1^{-1} \circ \varphi_1(\alpha_1)) = \psi_1^{-1} \circ \psi_1 \circ f_1(\mathbb{x}, \tau, \alpha_1)$$

  which is identically distributed to $f_1(\mathbb{x}, \tau, \alpha_1)$ by Lemma 3.2.

- *Inductive case.* Assume that $\rho_1, \ldots, \rho_{i-1}$ are identically distributed; we show that also ρ_i is identically distributed across Hyb_3 and Hyb_4. Let st_i be the duplex object resulting after squeezing the $(i-1)$-th challenge (cf. Item 4d). By inductive hypothesis, the duplex object st_i' after squeezing ρ_{i-1} is identically distributed across hybrids.

  The verifier absorbs the i-th prover message via $\mathsf{DS.Absorb}^{\boldsymbol{\psi}^{-1}\circ\mathsf{D2SQuery}\circ\boldsymbol{\varphi}^{-1}}(\mathsf{st}_{i-1}, \widehat{\boldsymbol{\alpha}}_i)$ which in turn will make $L_{\mathbf{P}}(i)$ queries to $\mathsf{D2SQuery}$, following **DSFS**. Since $E_{\mathcal{V}}$ does not happen, for each of the queries to p with input $\boldsymbol{s}$, $E_{\text{fork},p}(\mathsf{tr}, \boldsymbol{s}) = 0$ and thus, during the ι-th query to the permutation oracle p, $\mathsf{BackTrack}$ will return none since $\mathcal{S}_{\mathsf{BT}}$ will have only one candidate of length $L_\delta + \sum_{k<i} L_{\mathbf{P}}(k) + \iota$, and Item 4 will not satisfy the length requirements.

  Afterwards, $\mathcal{V}$ invokes then $\mathsf{DS.Squeeze}^{\boldsymbol{\psi}^{-1}\circ\mathsf{D2SQuery}\circ\boldsymbol{\varphi}^{-1}}(\mathsf{st}_i', \ell_{\mathbf{V}}(i))$ which will internally run $\mathsf{BackTrack}$. As before, the backtracking algorithm will recover a unique sequence of permutation states in $\mathcal{S}_{\mathsf{BT}}$ (see Equation 11), since E does not happen, and output $(\mathbb{x}, \boldsymbol{\tau}, \widehat{\boldsymbol{\alpha}}_1, \ldots, \widehat{\boldsymbol{\alpha}}_i)$ and set:

$$\rho_i := \psi_i^{-1} \circ \psi_i \circ f_i(\mathbb{x}, \mathsf{bin}(\boldsymbol{\tau}), \varphi_1^{-1} \circ \varphi_1(\alpha_1), \ldots, \varphi_i^{-1} \circ \varphi_i(\alpha_i)) = \psi_i^{-1} \circ \psi_i \circ f_i(\mathbb{x}, \tau, \alpha_1, \ldots, \alpha_i)\,.$$

  which is identically distributed to $f_i(\mathbb{x}, \tau, \alpha_1, \ldots, \alpha_i)$ by Lemma 3.2.

We conclude that

$$\begin{aligned}
&\Delta\left(\mathsf{Hyb}_3, \mathsf{Hyb}_4\right) \\
&\leq \Delta\left(\mathsf{Hyb}_3 \,\middle|\, \overline{E_{\mathcal{V}}}, \mathsf{Hyb}_4\right) + \Pr[E_{\mathcal{V}}] \\
&= 0 + \frac{7(L+1)(2t_h + 1 + 2t_p + L + 2t_{p^{-1}})}{2\,|\Sigma|^c} - \frac{5(L+1)}{|\Sigma|^c} \quad \text{(by Equation 57)} \\
&= \frac{7(L+1)(2t_h + 1 + 2t_p + L + 2t_{p^{-1}})}{2\,|\Sigma|^c} - \frac{5(L+1)}{|\Sigma|^c}\,. \quad \text{(by Equation 56)}
\end{aligned}$$

□

Adding up all intermediate error terms, we obtain:

$$
\begin{aligned}
&\Delta\left(\left(b,\mathbb{x},\pi,\mathsf{tr}\ \middle|\ \begin{array}{l} (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ (\mathbb{x},\pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}} \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{h,p}(\mathbb{x},\pi) \\ \mathsf{tr} := \mathsf{D2STrace}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}) \end{array}\right), \left(b,\mathbb{x},\pi,\mathsf{tr}\ \middle|\ \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}_{\mathsf{IP}}(\lambda,n) \\ (\mathbb{x},\pi_{\tilde{\mathcal{P}}_\star}) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}_\star}} \mathsf{D2SAlgo}^{\boldsymbol{f}}(\tilde{\mathcal{P}}) \\ b \xleftarrow{\mathsf{tr}_{\mathcal{V}_{\mathsf{std}}}} \mathcal{V}^{\boldsymbol{f}}_{\mathsf{std}}(\mathbb{x},\pi) \\ \mathsf{tr} := \mathsf{tr}_{\tilde{\mathcal{P}}_\star} \| \mathsf{tr}_{\mathcal{V}_{\mathsf{std}}} \end{array}\right)\right) \\
&\leq \frac{7(t_h+1+t_p+L+t_{p^{-1}})^2 - 3(t_h+1+t_p+L+t_{p^{-1}})}{2\,|\Sigma|^c} && \text{(by Claim 5.21)} \\
&\quad + \theta_\star(t_h,t_p,t_{p^{-1}}) \cdot \max_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda,n) + \sum_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda,n) && \text{(by Claim 5.22)} \\
&\quad + 0 && \text{(by Claim 5.23)} \\
&\quad + \frac{7(L+1)(2t_h+1+2t_p+L+2t_{p^{-1}})}{2\,|\Sigma|^c} - \frac{5(L+1)}{|\Sigma|^c} && \text{(by Claim 5.24)} \\
&\leq \frac{7(t_h+t_p+t_{p^{-1}})^2 + 28(L+1)(t_h+t_p+t_{p^{-1}}) + 14(L+1)^2 - 3(t_h+t_p+t_{p^{-1}}) - 13(L+1)}{2\,|\Sigma|^c} + \\
&\quad \theta_\star(t_h,t_p,t_{p^{-1}}) \cdot \max_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda,n) + \sum_{i\in[\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda,n)\,.
\end{aligned}
$$

```
1 :  $(h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n)$
       $\boldsymbol{g} \leftarrow \mathcal{D}_{\Sigma}(\lambda,n)$
       $\boldsymbol{e} \leftarrow \mathcal{U}\left(((\{0,1\}^{\leq n} \times \Sigma^{\delta} \times \Sigma^{\ell_{\mathbf{P}}(1)} \times \cdots \times \Sigma^{\ell_{\mathbf{P}}(i)} \to \mathcal{M}_{\mathbf{V},i})_{i\in[\mathsf{k}]}\right)$
       $\boldsymbol{f} \leftarrow \mathcal{U}\left(((\{0,1\}^{\leq n} \times \{0,1\}^{\delta_\star} \times \mathcal{M}_{\mathbf{P},1} \times \cdots \times \mathcal{M}_{\mathbf{P},i} \to \mathcal{M}_{\mathbf{V},i})_{i\in[\mathsf{k}]}\right)$
       $\boldsymbol{f} \leftarrow \mathcal{D}_{\mathsf{IP}}(\lambda,n)$
2 :  $(\mathbb{x},\pi) \xleftarrow{\mathsf{tr}_{\tilde{\mathcal{P}}}\ \boxed{\mathsf{tr}_{\tilde{\mathcal{P}}_\star}}} \tilde{\mathcal{P}}^{h,p,p^{-1}}$   $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}$   $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1}\circ\boldsymbol{e}}}$   $\tilde{\mathcal{P}}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1}\circ\boldsymbol{f}\circ\boldsymbol{\varphi}^{-1}}}$   $\mathsf{D2SAlgo}^{\boldsymbol{f}}(\tilde{\mathcal{P}})$
3 :  $b \xleftarrow{\mathsf{tr}_{\mathcal{V}}\ \boxed{\mathsf{tr}_{\mathcal{V}_{\mathsf{std}}}}} \mathcal{V}^{h,p}(\mathbb{x},\pi)$   $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{g}}}(\mathbb{x},\pi)$   $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1}\circ\boldsymbol{e}}}(\mathbb{x},\pi)$   $\mathcal{V}^{\mathsf{D2SQuery}^{\boldsymbol{\psi}^{-1}\circ\boldsymbol{f}\circ\boldsymbol{\varphi}^{-1}}}(\mathbb{x},\pi)$   $\mathcal{V}^{\boldsymbol{f}}_{\mathsf{std}}(\mathbb{x},\pi)$
4 :  $\mathsf{tr} := \mathsf{D2STrace}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$   $(\boldsymbol{\varphi}^{-1},\boldsymbol{\psi})(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$   $\boldsymbol{\varphi}^{-1}(\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}})$   $\mathsf{tr}_{\tilde{\mathcal{P}}} \| \mathsf{tr}_{\mathcal{V}}$   $\mathsf{tr}_{\tilde{\mathcal{P}}_\star} \| \mathsf{tr}_{\mathcal{V}_{\mathsf{std}}}$
5 :  return $(b,\mathbb{x},\pi,\mathsf{tr})$
```

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 $\delta_\star$:

$$\begin{aligned} \theta_\star(t) &:= t\,, \\ \eta_\star(\lambda, t) &:= \frac{25t^2}{|\Sigma|^c} + t \cdot \max_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i} + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}\,, \\ \delta_\star &:= \delta \log |\Sigma|\ . \end{aligned} \tag{58}$$

Theorem 6.1 (soundness)**.** *If* IP *has state-restoration soundness error* $\varepsilon^{\mathrm{sr}}_{\mathsf{IP}}$ *(see Definition 3.14) then* NARG *has soundness error* $\varepsilon_{\mathsf{NARG}}$ *(see Definition 3.5) 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 3.16) 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 3.8) 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*
- $\mathbf{et}_{\mathsf{NARG}}(\lambda, t, n, \delta_{\tilde{\mathcal{P}}}(\lambda, n), \tau_{\tilde{\mathcal{P}}}(\lambda, n)) \leq \mathbf{et}^{\mathrm{sr}}_{\mathsf{IP}}\left(\delta_\star, \theta_\star(t), n, \delta'_{\tilde{\mathcal{P}}}(\lambda, n), \tau'_{\tilde{\mathcal{P}}}(\lambda, n)\right) + t^2 \cdot \log t \cdot (n + (r + c) \cdot \log |\Sigma|) + t \cdot (t_{\boldsymbol{\varphi}^{-1}} + t_{\boldsymbol{\psi}} + \delta_\star)$.

Above,

$$\begin{aligned} \delta'_{\tilde{\mathcal{P}}}(\lambda, n) &:= \delta_{\tilde{\mathcal{P}}}(\lambda, n) + \eta_\star(\lambda, t) \\ \tau'_{\tilde{\mathcal{P}}}(\lambda, n) &:= \tau_{\tilde{\mathcal{P}}}(\lambda, n) + t^2 \cdot \log t \cdot (n + (r + c) \cdot \log |\Sigma|) + t \cdot (t_{\boldsymbol{\varphi}^{-1}} + t_{\boldsymbol{\psi}} + \delta_\star)\,. \end{aligned}$$

Moreover, if the IP state-restoration extractor is straightline (see Definition 3.14) then the NARG extractor is also straightline (see Definition 3.6). 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* $\mathbf{et}_{\mathsf{NARG}}(\lambda, t, n) \leq \mathbf{et}^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t), n) + t^2 \cdot \log t \cdot (n + (r + c) \cdot \log |\Sigma|) + t \cdot (t_{\boldsymbol{\varphi}^{-1}} + t_{\boldsymbol{\psi}} + \delta_\star)$.

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 $(\mathcal{P}, \mathcal{V}) := \mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ (Construction 4.3) to the soundness of $(\mathcal{P}_{\mathsf{std}}, \mathcal{V}_{\mathsf{std}}) := \mathsf{FS}[\mathsf{IP}, \delta_\star]$ (Construction 3.17). Specifically, by Lemma 5.1, every $(t_h, t_p, t_{p^{-1}})$-query malicious argument prover $\tilde{\mathcal{P}}$ for $\textbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ 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 soundness of $\mathsf{FS}[\mathsf{IP}, \delta_\star]$ is determined by the state-restoration soundness of IP. Specifically:

$$\begin{aligned}
&\varepsilon_{\mathsf{NARG}}(\lambda, (t_h, t_p, t_{p^{-1}}), n) \\
&= \Pr\left[\begin{array}{l|l} |\mathbb{x}| \le n & (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ \wedge\ \mathbb{x} \notin \mathcal{L}(\mathcal{R}) & (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}^{h,p,p^{-1}} \\ \wedge\ \mathcal{V}^{h,p}(\mathbb{x}, \pi) = 1 & \end{array}\right] && \text{(by Definition 3.5)} \\
&\le \Pr\left[\begin{array}{l|l} |\mathbb{x}| \le n & \boldsymbol{f} \leftarrow \mathcal{D}_{\mathsf{IP}}(\lambda, n) \\ \wedge\ \mathbb{x} \notin \mathcal{L}(\mathcal{R}) & (\mathbb{x}, \pi) \leftarrow \mathsf{D2SAlgo}^{\boldsymbol{f}}(\tilde{\mathcal{P}}) \\ \wedge\ \mathcal{V}^{\boldsymbol{f}}_{\mathsf{std}}(\mathbb{x}, \pi) = 1 & \end{array}\right] + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) && \text{(by Lemma 5.1)} \\
&\le \varepsilon_{\mathsf{FS}[\mathsf{IP},\delta_\star]}(\lambda, \theta_\star(t_h, t_p, t_{p^{-1}}), n) + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) && \text{(by Definition 3.5)} \\
&\le \varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n) + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}}))\,. && \text{(by Theorem 3.18)}
\end{aligned}$$

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:

$$\begin{aligned}
&\varepsilon_{\mathsf{NARG}}(\lambda, (t_h, t_p, t_{p^{-1}}), n) \\
&\le \varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n) + \eta_\star(\lambda, (t_h, t_p, t_{p^{-1}})) && \text{(by the above derivation)} \\
&= \varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t_h, t_p, t_{p^{-1}}), n) \\
&\quad + \frac{7(t_h + t_p + t_{p^{-1}})^2 + 28(L+1)(t_h + t_p + t_{p^{-1}}) + 14(L+1)^2 - 3(t_h + t_p + t_{p^{-1}}) - 13(L+1)}{2\,|\Sigma|^c} + \\
&\quad \theta_\star(t_h, t_p, t_{p^{-1}}) \cdot \max_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) && \text{(by Equation 5)} \\
&\le \varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t), n) + \frac{25t^2}{|\Sigma|^c} + \theta_\star(t) \cdot \max_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i}(\lambda, n) && \text{(since } t_h + t_p + t_{p^{-1}} \le t\text{)} \\
&= \varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t), n) + \frac{25t^2}{|\Sigma|^c} + t \cdot \max_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i} + \sum_{i \in [\mathsf{k}]} \varepsilon_{\mathsf{cdc},i} \\
&= \varepsilon^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t), n) + \eta_\star(\lambda, t)\,. && \text{(by definition of } \eta_\star(\lambda, t)\text{)}
\end{aligned}$$

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* $\mathsf{tr}_{\mathsf{std}} := \mathsf{D2STrace}(\mathsf{tr} \| \mathsf{tr}_{\mathcal{V}})$.
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 3.17). 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 3.19, 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 3.8)} \\
&= \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}}^{\boldsymbol{f}}_{\mathsf{std}} \\ \wedge\, b = 1 & b \xleftarrow{\mathsf{tr}_{\mathcal{V}}} \mathcal{V}^{\boldsymbol{f}}_{\mathsf{std}}(\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{FS}[\mathsf{IP},\delta_\star]}(\lambda, \theta_\star(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 3.8)} \\
&\le \kappa_{\mathsf{FS}[\mathsf{IP},\delta_\star]}\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^{\mathrm{sr}}_{\mathsf{IP}}\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 3.19)}
\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 (a bound on the runtime is given in Equation 18), 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

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

We simplify the expression by using:

$$t_h + t_p + t_{p^{-1}} \le t\,,$$
$$\delta_{\tilde{\mathcal{P}}_{\mathsf{std}}} \le \delta_{\tilde{\mathcal{P}}} + \eta_\star(\lambda, t)\,,$$
$$\tau_{\tilde{\mathcal{P}}_{\mathsf{std}}} \le \tau_{\tilde{\mathcal{P}}} + t^2 \cdot \log t \cdot (n + (r+c) \cdot \log|\Sigma|) + t \cdot (t_{\boldsymbol{\varphi}^{-1}} + t_{\boldsymbol{\psi}} + \delta_\star)\,,$$

and using Theorem 3.19:

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

The straightline case. The above analysis directly specializes to the straightline case. Suppose that $\mathbf{E}^{\mathrm{sr}}$ is a straightline extractor (Definition 3.14): it is a deterministic algorithm that does not need access to the IP state-restoration prover $\tilde{\mathbf{P}}^{\mathrm{sr}}$. Then, by Theorem 3.19, $\mathcal{E}_{\mathsf{std}}$ is a straightline extractor (Definition 3.6): 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 3.6): 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

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

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

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

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

$$\mathbf{et}_{\mathsf{NARG}}(\lambda, t, n) \le \mathbf{et}^{\mathrm{sr}}_{\mathsf{IP}}(\delta_\star, \theta_\star(t), n) + t^2 \cdot \log t \cdot (n + (r+c) \cdot \log|\Sigma|) + t \cdot (t_{\boldsymbol{\varphi}^{-1}} + t_{\boldsymbol{\psi}} + \delta_\star)\,.$$

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* $n \in \mathbb{N}$*,*

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

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 $i = 1, \dots, \mathsf{k}$:
    (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 $\mu_h$ be the empty list. (The oracle $h$ is not programmed.)
8. Set $\mu_p$ 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 $(\pi, \mu)$.
```

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 $\mu_{p,i}$ for the permutation oracle p, a corresponding query-answer list $\mu_{p^{-1},i}$ 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 rage segments $\boldsymbol{s}_{\mathrm{R},1}, \boldsymbol{s}_{\mathrm{R},2}, \dots, \boldsymbol{s}_{\mathrm{R},L_{\mathbf{V}}(i)} \in \Sigma^r$.
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

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

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 IP 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 $\widehat{\boldsymbol{\rho}}_1 \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(1)}), \ldots, \widehat{\boldsymbol{\rho}}_{\mathsf{k}} \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(\mathsf{k})})$ 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

$$E_{\text{start}}(\mathsf{tr}, \boldsymbol{s}_0) := \text{“}\, \exists\, \boldsymbol{s}_1' \in \Sigma^{r+c} \colon (\text{‘}p\text{’}, \boldsymbol{s}_0, \boldsymbol{s}_1') \in \mathsf{tr} \,\vee\, (\text{‘}p^{-1}\text{’}, \boldsymbol{s}_1', \boldsymbol{s}_0) \in \mathsf{tr}\,\text{”}\,. \tag{59}$$

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

$$E_{\text{squeeze}}(\mathsf{tr}, (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{L_{\mathbf{V}}})) := \text{“}\, \exists\, \iota \in [L_{\mathbf{V}} - 1]\,,\, \boldsymbol{s}_{\text{out}} \in \Sigma^{r+c} \colon \boldsymbol{s}_{\text{out}} \neq \boldsymbol{s}_{\iota+1} \wedge (\text{‘}p\text{’}, \boldsymbol{s}_\iota, \boldsymbol{s}_{\text{out}}) \in \mathsf{tr}\,\text{”}\,. \tag{60}$$

Next, consider the following two events.

- In Hyb_0, E_0 is the (bad) event that $E_{\text{start}}(\mathsf{tr}, \boldsymbol{s}_0) \vee E_{\text{squeeze}}(\mathsf{tr}, (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{L_{\mathbf{V}}}))$ 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
  - $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{L_{\mathbf{V}}}$ are the permutations states in the encoded verifier messages $\widehat{\boldsymbol{\rho}}_1, \ldots, \widehat{\boldsymbol{\rho}}_{\mathsf{k}}$ produced by $\mathsf{DS.Squeeze}^p$ in line 11 (across all k rounds).
- In Hyb_1, E_1 is the (bad) event that $E_{\text{start}}(\mathsf{tr}, \boldsymbol{s}_0) \vee E_{\text{squeeze}}(\mathsf{tr}, (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{L_{\mathbf{V}}}))$ 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);
  - $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{L_{\mathbf{V}}}$ 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 $\Pr[E_0] \leq \Pr[E_1]$. Observe that:

- E_{start} is identical across Hyb_0 and Hyb_1; moreover,
- E_{squeeze} never holds in Hyb_0 because there is no programming (p is a function so for every $(\text{`}p\text{'}, \boldsymbol{s}_\iota, \boldsymbol{s}_{\text{out}})$ and $(\text{`}p\text{'}, \boldsymbol{s}'_\iota, \boldsymbol{s}'_{\text{out}})$ in tr it holds that $\boldsymbol{s}_\iota = \boldsymbol{s}'_\iota \implies \boldsymbol{s}_{\text{out}} = \boldsymbol{s}'_{\text{out}}$).

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}) \overset{\mathsf{tr}}{\leftarrow} \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] \\
&\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}) \overset{\mathsf{tr}}{\leftarrow} \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}) \\ \textbf{for } i = 1, \dots, \mathsf{k} \\ \quad (\alpha_i, \mathsf{aux}_i) := \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 := \varphi_i(\alpha_i) \\ \quad \mathsf{st}'_i := \mathsf{DS.Absorb}^p(\mathsf{st}_{i-1}, \widehat{\boldsymbol{\alpha}}_i) \\ \quad (\widehat{\boldsymbol{\rho}}_i, \mathsf{st}'_i) := \mathsf{DS.Squeeze}^p(\mathsf{st}'_i, \ell_{\mathbf{V}}(i)); \quad \rho_i := \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}) \overset{\mathsf{tr}}{\leftarrow} \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] + 0 \\
&\leq \Pr[E_1] \,.
\end{aligned}$$

Next, we argue that $\mathsf{Hyb}_0 \mid \overline{E_0}$ and $\mathsf{Hyb}_1 \mid \overline{E_1}$ have the same distribution:

- In Hyb_0, there is no programming, and the permutation p and its inverse p^{-1} answer random and consistent elements in Σ^{r+c}.
- In Hyb_1, the permutation states $\boldsymbol{s}_1, \dots, \boldsymbol{s}_{L_{\mathbf{V}}}$ are uniformly and independently sampled in $\mathsf{ProgramBlocks}$. The event E_1 does not hold, so for every $\iota \in [0, L_{\mathbf{V}} - 1]$ it holds that $(\text{`}p\text{'}, \boldsymbol{s}_\iota, \boldsymbol{s}_{\text{out}}) \in \mathsf{tr} \implies \boldsymbol{s}_{\text{out}} = \boldsymbol{s}_{\iota+1}$, which implies that p's answers are consistent.

We deduce that

$$\begin{aligned}
\Delta\left(\mathsf{Hyb}_0, \mathsf{Hyb}_1\right) &\leq \Delta\left(\mathsf{Hyb}_0 \mid \overline{E_0}, \mathsf{Hyb}_1 \mid \overline{E_1}\right) + \max\{\Pr[E_0], \Pr[E_1]\} && \text{(by [CY24, Claim 1.2.10])} \\
&= 0 + \Pr[E_1] && \text{(argued above)} \\
&= \Pr[E_1] \,.
\end{aligned}$$

We are left to upper bound the probability of E_1. We do this by separately upper bounding the probability of E_{start} in Claim 7.7 and of E_{squeeze} in Claim 7.8, which implies the claimed upper bound:

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

Claim 7.7. *The following holds:*

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

Proof. Let $L_\delta := \lceil \delta/r \rceil$ be the number of blocks in the salt string $\boldsymbol{\tau}$. We distinguish two cases.

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

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

$$\le \Pr\left[(\text{‘}p\text{’}, \boldsymbol{s}_0, \cdot) \in \mathsf{tr} \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^{\delta}) \\ \boldsymbol{s}_0 := (\boldsymbol{\tau} \| 0^{r-\delta}, h(\mathbb{x})) \end{array} \right] + \Pr\left[(\text{‘}p^{-1}\text{’}, \cdot, \boldsymbol{s}_0) \in \mathsf{tr} \;\middle|\; \begin{array}{l} (h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n) \\ (\mathbb{x}, \mathbb{w}, \mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^{\delta}) \\ \boldsymbol{s}_0 := (\boldsymbol{\tau} \| 0^{r-\delta}, h(\mathbb{x})) \end{array} \right]$$

$$\le \frac{t_p}{|\Sigma|^{\delta}} + \frac{t_{p^{-1}}}{|\Sigma|^{\delta}} \, .$$

- *Case 2: $L_\delta > 1$ (i.e. $\delta > r$).* We split the analysis of this case into two parts: there exists a sequence of query-answers to p that match an absorb of the salt $\boldsymbol{\tau}$, or no sequence matching the salt can be found.

  Let $\boldsymbol{s}_{\mathrm{C},-L_\delta+1} := h(\mathbb{x}) \in \Sigma^c$. Parse $\boldsymbol{\tau} \| 0^{r-(\delta \bmod r)}$ into the sequence of L_δ rate segments $(\boldsymbol{s}_{\mathrm{R},-L_\delta+1} \| \cdots \| \boldsymbol{s}_{\mathrm{R},0})$. We consider the following statement: there exist capacity segments $\boldsymbol{s}_{\mathrm{C},-L_\delta+2}, \ldots, \boldsymbol{s}_{\mathrm{C},0} \in \Sigma^c$ such that

$$\forall\, j \in \{-L_\delta + 1, \ldots, 0\} \colon \quad \begin{array}{ll} & (\text{‘}p\text{’}, (\boldsymbol{s}_{\mathrm{R},j-1}, \boldsymbol{s}_{\mathrm{C},i-1}), (\boldsymbol{s}_{\mathrm{R},j}, \boldsymbol{s}_{\mathrm{C},j})) \in \mathsf{tr} \\ \vee & (\text{‘}p^{-1}\text{’}, (\boldsymbol{s}_{\mathrm{R},j}, \boldsymbol{s}_{\mathrm{C},t}), (\boldsymbol{s}_{\mathrm{R},j-1}, \boldsymbol{s}_{\mathrm{C},j-1})) \in \mathsf{tr} \, . \end{array} \qquad (61)$$

  Then:

  (a) If Equation 61 holds:

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

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

$$+ \Pr\left[\begin{array}{l|l} \forall\, j \in \{-L_\delta+1,\dots,0\}: & (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ (\text{`}p^{-1}\text{'}, (\boldsymbol{s}_{\mathrm{R},j},\cdot),(\boldsymbol{s}_{\mathrm{R},j-1},\cdot)) \in \mathsf{tr} & (\mathbb{x},\mathbb{w},\mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ & \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta) \\ & \text{parse } \boldsymbol{\tau}\|0^{r-(\delta \bmod r)} \text{ into blocks } \boldsymbol{s}_{\mathrm{R},-L_\delta+1},\dots,\boldsymbol{s}_{\mathrm{R},0} \end{array}\right]$$

$$= \left(1 - \left(1 - \frac{1}{|\Sigma|^\delta}\right)^{t_p}\right) + \left(1 - \left(1 - \frac{1}{|\Sigma|^\delta}\right)^{t_{p^{-1}}}\right)$$

$$\leq \frac{t_p}{|\Sigma|^\delta} + \frac{t_{p^{-1}}}{|\Sigma|^\delta}\,. \qquad \text{(by Bernoulli's inequality)}$$

(b) Else (Equation 61 does not hold), let

$$j^* := \operatorname*{arg\,max}_{j \in \{-L_\delta+1,\dots,0\}} : \begin{array}{l} (\text{`}p\text{'}, (\boldsymbol{s}_{\mathrm{R},j-1},\boldsymbol{s}_{\mathrm{C},j-1}),(\boldsymbol{s}_{\mathrm{R},j},\boldsymbol{s}_{\mathrm{C},j})) \notin \mathsf{tr} \\ \wedge\, (\text{`}p^{-1}\text{'}, (\boldsymbol{s}_{\mathrm{R},j},\boldsymbol{s}_{\mathrm{C},j}),(\boldsymbol{s}_{\mathrm{R},j-1},\boldsymbol{s}_{\mathrm{C},j-1})) \notin \mathsf{tr}\,. \end{array} \tag{62}$$

In this case, the capacity state $\boldsymbol{s}_{\mathrm{C},j^*+1}$ is sampled uniformly at random from Σ^c (see line 5), and has been correctly guessed from the adversary. That is,

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

$$\leq \Pr\left[\begin{array}{l|l} & (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ & (\mathbb{x},\mathbb{w},\mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \exists\, j^* \text{ as per Equation 62} & \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta) \\ \wedge\, (\text{`}p\text{'},\cdot,(\cdot,\boldsymbol{s}_{\mathrm{C},j^*})) \in \mathsf{tr} & \boldsymbol{s}_{\mathrm{C},-L_\delta+1} := h(\mathbb{x}) \\ & \boldsymbol{s}_{\mathrm{C},-L_\delta+2},\dots,\boldsymbol{s}_{\mathrm{C},0} \leftarrow \mathcal{U}(\Sigma^c) \\ & (\boldsymbol{s}_{\mathrm{R},-L_\delta+1}\|\cdots\|\boldsymbol{s}_{\mathrm{R},0}) := (\boldsymbol{\tau}\|0^{r-(\delta \bmod r)}) \end{array}\right]$$

$$+ \Pr\left[\begin{array}{l|l} & (h,p,p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda,n) \\ & (\mathbb{x},\mathbb{w},\mathsf{aux}) \xleftarrow{\mathsf{tr}} \mathcal{A}^{h,p,p^{-1}} \\ \exists\, j^* \text{ as per Equation 62} & \boldsymbol{\tau} \leftarrow \mathcal{U}(\Sigma^\delta) \\ \wedge\, (\text{`}p\text{'},\cdot,(\cdot,\boldsymbol{s}_{\mathrm{C},j^*})) \in \mathsf{tr} & \boldsymbol{s}_{\mathrm{C},-L_\delta+1} := h(\mathbb{x}) \\ & \boldsymbol{s}_{\mathrm{C},-L_\delta+2},\dots,\boldsymbol{s}_{\mathrm{C},0} \leftarrow \mathcal{U}(\Sigma^c) \\ & (\boldsymbol{s}_{\mathrm{R},-L_\delta+1}\|\cdots\|\boldsymbol{s}_{\mathrm{R},0}) := (\boldsymbol{\tau}\|0^{r-(\delta \bmod r)}) \end{array}\right]$$

$$\leq \frac{t_p}{|\Sigma|^c} + \frac{t_{p^{-1}}}{|\Sigma|^c}\,.$$

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

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

□

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:

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

(by Equation 60)

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

□

Hyb_2. In this hybrid we modify line 11 (relative to Hyb_1). 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 $\rho_i := \psi_i(\widehat{\boldsymbol{\rho}}_i')$, 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:

$$\begin{aligned}
&\Delta\left(\mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)}), (\psi_i^{-1} \circ \psi_i \circ \mathcal{U})(\Sigma^{\ell_{\mathbf{V}}(i)})\right) \\
&= \sum_{\boldsymbol{x} \in \Sigma^{\ell_{\mathbf{V}}(i)}} \left| \Pr\left[\boldsymbol{x}' = \boldsymbol{x} \;\middle|\; \boldsymbol{x}' \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)})\right] - \Pr\left[\boldsymbol{x}'' = \boldsymbol{x} \;\middle|\; \begin{array}{c} \boldsymbol{x}' \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)}) \\ y := \psi_i(\boldsymbol{x}') \\ \boldsymbol{x}'' \leftarrow \psi_i^{-1}(y) \end{array}\right] \right| \\
&= 0\,. \qquad \text{(by Lemma 3.2)}
\end{aligned}$$

□

Hyb₃. In this hybrid we modify line 11 (relative to Hyb_2). In each round $i \in [\mathsf{k}]$, we sample $\widehat{\boldsymbol{\rho}}_i \in \Sigma^{\ell_{\mathbf{V}}(i)}$ in a different way: 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. □

Hyb₄. In this hybrid we replace the IP prover $\mathbf{P}$ with the IP simulator $\mathbf{S}$. Instead of computing the prover messages $(\alpha_i)_{i\in[\mathsf{k}]}$ using the IP prover $\mathbf{P}$ as in hybrid Hyb_3 (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 :  $(h, p, p^{-1}) \leftarrow \mathcal{D}_{\mathfrak{S}}(\lambda, n)$
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 :  $\mathsf{st}'_0 := \mathsf{DS.Start}^h(\mathbb{x})$
5 :  $\mathsf{st}_0 := \mathsf{DS.Absorb}^p(\mathsf{st}'_0, \boldsymbol{\tau})$
6 :  $\boxed{(\mathbb{x}, (\rho_i)_{i \in [\mathsf{k}]}, (\alpha_i)_{i \in [\mathsf{k}]}) \leftarrow \mathbf{S}(\mathbb{x})}$
7 :  $\mathbf{for}\ i = 1, \ldots, \mathsf{k}$
8 :    $(\alpha_i, \mathsf{aux}_i) := \mathbf{P}(\mathbb{x}, \mathbb{w})\ \mathbf{if}\ i = 1\ \mathbf{else}\ \mathbf{P}(\mathsf{aux}_{i-1}, \rho_{i-1})$
9 :    $\widehat{\boldsymbol{\alpha}}_i := \varphi_i(\alpha_i)$
10 :   $\mathsf{st}'_i := \mathsf{DS.Absorb}^p(\mathsf{st}_{i-1}, \widehat{\boldsymbol{\alpha}}_i)$
11 :   $(\widehat{\boldsymbol{\rho}}_i, \mathsf{st}'_i) := \mathsf{DS.Squeeze}^p(\mathsf{st}'_i, \ell_{\mathbf{V}}(i));\quad \rho_i := \psi_i(\widehat{\boldsymbol{\rho}}_i)$
       $\boxed{\widehat{\boldsymbol{\rho}}_i \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)});\ \rho_i := \psi_i(\widehat{\boldsymbol{\rho}}_i);\quad (\mu_{p,i}, \mu_{p^{-1},i}, \mathsf{st}_i) := \mathsf{ProgramBlocks}(i, \mathsf{st}'_i, \widehat{\boldsymbol{\rho}}_i)}$
       $\boxed{\widehat{\boldsymbol{\rho}}'_i \leftarrow \mathcal{U}(\Sigma^{\ell_{\mathbf{V}}(i)});\ \rho_i := \psi_i(\widehat{\boldsymbol{\rho}}'_i);\ \widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i);\ (\mu_{p,i}, \mu_{p^{-1},i}, \mathsf{st}_i) := \mathsf{ProgramBlocks}(i, \mathsf{st}'_i, \widehat{\boldsymbol{\rho}}_i)}$
       $\boxed{\rho_i \leftarrow \mathcal{U}(\mathcal{M}_{\mathbf{V},i});\ \widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i);\quad (\mu_{p,i}, \mu_{p^{-1},i}, \mathsf{st}_i) := \mathsf{ProgramBlocks}(i, \mathsf{st}'_i, \widehat{\boldsymbol{\rho}}_i)}$
       $\boxed{\widehat{\boldsymbol{\rho}}_i \leftarrow \psi_i^{-1}(\rho_i);\quad (\mu_{p,i}, \mu_{p^{-1},i}, \mathsf{st}_i) := \mathsf{ProgramBlocks}(i, \mathsf{st}'_i, \widehat{\boldsymbol{\rho}}_i)}$
12 : $\mu_h := (\,)$
13 : $\mu_p := \mu_{p,1} \| \cdots \| \mu_{p,\mathsf{k}}$
14 : $\mu_{p^{-1}} := \mu_{p^{-1},1} \| \cdots \| \mu_{p^{-1},\mathsf{k}}$
15 : $\pi := (\boldsymbol{\tau}, (\alpha_i)_{i \in [\mathsf{k}]})$
16 : $y \leftarrow \mathcal{A}^{(h,p,p^{-1})}(\mathsf{aux}, \pi)$  $\boxed{\mathcal{A}^{(h,p,p^{-1})[\mu_h, \mu_p, \mu_{p^{-1}}]}(\mathsf{aux}, \pi)}$
```

Figure 5: Hybrid experiments Hyb_0, $\boxed{\mathsf{Hyb}_1}$, $\boxed{\mathsf{Hyb}_2}$, $\boxed{\mathsf{Hyb}_3}$, $\boxed{\mathsf{Hyb}_4}$ 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 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 `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 relies 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 corresponding duplex-sponge Fiat–Shamir transformation).
- A trait `Unit` defines the alphabet Σ for the permutation (and hash function). The only requirements for `Unit` are: constant size known at compile time (`Sized`), copy (`Clone`), secure deletion (via `zeroize::Zeroize`), and serialization/deserialization (via functions `write` and `read`).
- A `DuplexSponge<P: Permutation>` implementation is given, which implements the duplex construction in overwrite mode as described in Construction 3.3, based on a given arbitrary permutation `P`.

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

- A `ProverState<P: Permutation>` structure that contains the NARG prover state. This structure allows absorbing/squeezing units, and internally updating the argument string. It contains: (i) the current sponge state, (ii) the NARG string serialized so far, and (iii) the private coins of the argument prover.
- A `VerifierState<P: Permutation>` structure that 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 derive the IP 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*[13] 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 `FieldToUnit<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 in `ff` (within `zkcrypto/ff`).

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

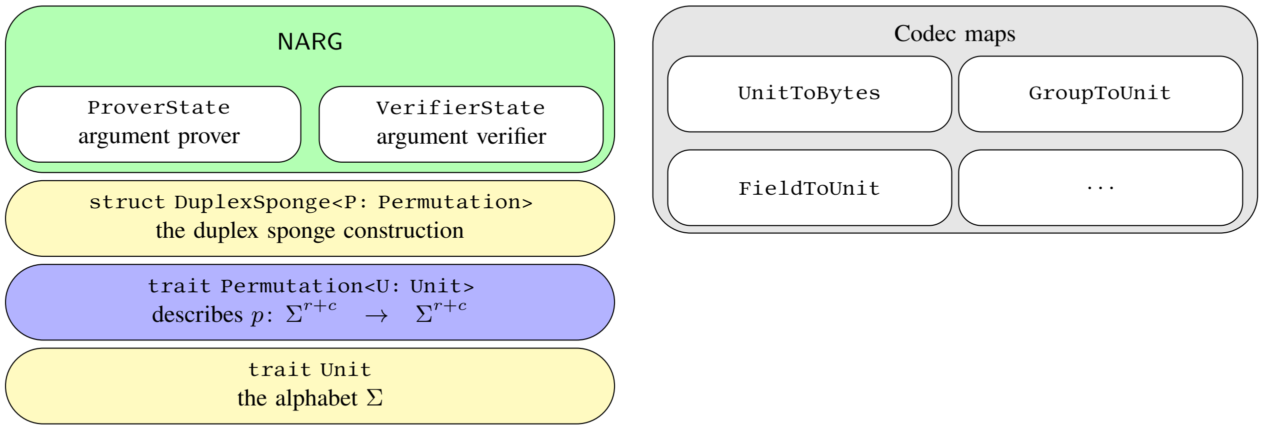

NARG
ProverState
argument prover
VerifierState
argument verifier
struct DuplexSponge<P: Permutation>
the duplex sponge construction
trait Permutation<U: Unit>
describes $p\colon \Sigma^{r+c} \to \Sigma^{r+c}$
trait Unit
the alphabet Σ
Codec maps
UnitToBytes
GroupToUnit
FieldToUnit
...

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

- An extension `UnitToField<F>` that provides a negligibly-biased map ψ that sends uniformly-distributed units into (almost) uniformly-distributed elements of a field `F` following Lemma C.1. The extension supports units that are bytes (`U=u8`) and field elements (`U=F`). The extension is implemented for `arkworks` and `zkcrypto/ff`.
- An extension `GroupToUnit<G>` that provides an encoding map φ that compresses and serializes elliptic curve group elements into units. We support elliptic curves types from `arkworks-rs/algebra` and `zkcrypto/ff`.
- An extension `UnitToBytes` 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 C.1 to ensure that the squeezed IP verifier messages are negligibly close to uniformly distributed ones.

8.2 Concrete security and ergonomics

We discuss some design choices and 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 let `ProverState` build the argument string, without having to worry about serialization/deserialization of the NARG string. On the other end, `VerifierState` reads incrementally the IP prover messages from the NARG string recovering the IP messages and recomputing the IP 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 absorbed and to include those messages in the correct round;[14] (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.

Session identifiers. The focus of this paper is single-theorem (knowledge) soundness and zero-knowledge. Real-world protocols, however, demand more complex security notions, and the gold standard for security is *universal composability* [Can01]. For protocols that rely on random oracles, this means establishing universal composability in (a suitable flavor of) the global random oracle model [CDGLN18]. zkSNARKs based on the

[14]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.

BCS transformation [BCS16] are known to be UC-secure [CF24] and, since the Fiat–Shamir transformation is a building block of the BCS transformation (see Section 2.6), we believe it is plausible that **DSFS** can also be shown to satisfy universal composability (given suitable properties for the underlying IP) in a suitably defined *global random permutation model*. While the formal study of the UC-security of **DSFS** is left to future work (see Section 1.3), our library includes features intended to facilitate composability.

Specifically UC-security in the global random oracle model demands domain separation of the oracle by way of session identifiers. There is a natural place to put such information in **DSFS**: the initialization of the sponge state via the oracle h, which can be pre-processed and chosen separately from the permutation oracle p used during the online phase of the protocol. Specifically, the initialization step $\mathsf{st}'_0 := \mathsf{DS.Start}^h(\mathbb{x})$ in Construction 4.3 is replaced by $\mathsf{st}'_0 := \mathsf{DS.Start}^h(\mathsf{sid}\|\mathbb{x})$ where sid is a fixed-length string that can be viewed as (the hash of) a session identifier. Our library provides an additional structure `DomainSeparator` that wraps the `String` type and exposes helper functions for embedding additional information within the initial oracle call. This information is then hashed using Keccak to produce sid.

The user is responsible for choosing a session identifier sid that is unique, for example, including: (i) a protocol identifier, which uniquely determines the non-interactive argument including information such as the codec and the hash functions used; (ii) a context identifier, providing information such as a timestamp and a URL identifying the resource in which the proof is being used; (iii) a serialization of the relation being proved.

Private randomness of the argument prover. The argument prover `ProverState` uses `DuplexSponge<U>` as part of **DSFS** in order to derive the IP verifier messages. In addition, `ProverState` uses a separate duplex sponge `DuplexSponge<u8>` for its own randomness: to derive the salt string $\boldsymbol{\tau} \in \Sigma^{\delta}$ (included in the argument string) and to derive the private randomness of the IP prover (if any, and typically used for zero knowledge). This latter duplex sponge is seeded by operating system randomness and, optionally, external randomness provided by the user. (This helps in constrained environments lacking a source of high-quality randomness.)

A On indifferentiability

In Appendix A.1 we discuss oracle implementations and indifferentiability [MRH04]. In Appendix A.2 we discuss how indifferentiability affects the properties of a non-interactive argument (and its limitations for this).

A.1 Oracle constructions and indifferentiability

We define *oracle constructions* (Definition A.1) and *indifferentiability* (Definition A.2). We also define a notion of *strong indifferentiability* (Definition A.3), and show that it is implied via a certain loss (Lemma A.4). In Remark A.5 we show to capture **DSFS** via the abstraction of oracle constructions.

Below $\mathcal{D}$ and $\mathcal{D}_\diamond$ are oracle distributions (families of distributions indexed by a security parameter $\lambda \in \mathbb{N}$ and an instance size bound $n \in \mathbb{N}$).

Definition A.1. *A* $(\mathcal{D}, \mathcal{D}_\diamond)$**-construction** *is a probabilistic stateful algorithm $\mathcal{C}$ that works as follows: $\mathcal{C}$ receives as input a security parameter $\lambda \in \mathbb{N}$ and instance size bound $n \in \mathbb{N}$ and oracle access to functions $\boldsymbol{g} \in \mathcal{D}_\diamond(\lambda, n)$, and answers queries with inputs and outputs consistent with functions $\boldsymbol{f} \in \mathcal{D}(\lambda, n)$.*

Definition A.2. *A $(\mathcal{D}, \mathcal{D}_\diamond)$-construction $\mathcal{C}$ is* **indifferentiable with error** ε_{ind} *if there exists a probabilistic stateful algorithm $\mathcal{S}_\mathcal{C}$ such that for every t-query T-time algorithm D:*

$$\left|\Pr\left[D^{\mathcal{C}^{\boldsymbol{g}},\boldsymbol{g}} = 1 \,\middle|\, \boldsymbol{g} \leftarrow \mathcal{D}_\diamond(\lambda, n)\right] - \Pr\left[D^{\boldsymbol{f},\mathcal{S}_\mathcal{C}^{\boldsymbol{f}}} = 1 \,\middle|\, \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n)\right]\right| \leq \varepsilon_{\text{ind}}(\lambda, n, t, T) .$$

Definition A.3. *A $(\mathcal{D}, \mathcal{D}_\diamond)$-construction $\mathcal{C}$ is* **strongly indifferentiable with error** $\varepsilon_{\text{sind}}$ *if there exists a probabilistic stateful algorithm $\mathcal{S}_\mathcal{C}$ such that for every t-query T-time ℓ-bit-output algorithm A:*

$$\Delta\left(A^{\mathcal{C}^{\boldsymbol{g}},\boldsymbol{g}}, A^{\boldsymbol{f},\mathcal{S}_\mathcal{C}^{\boldsymbol{f}}}\right) \leq \varepsilon_{\text{sind}}(\lambda, n, t, T, \ell) .$$

Lemma A.4. *If $\mathcal{C}$ is indifferentiable with error ε_{ind} then $\mathcal{C}$ is strongly indifferentiable with error $\varepsilon_{\text{sind}}$ such that*

$$\varepsilon_{\text{sind}}(\lambda, n, t, T, \ell) \leq \ell \cdot \varepsilon_{\text{ind}}(\lambda, n, t, T) .$$

Proof. Let $\mathcal{S}_\mathcal{C}$ be the probabilistic stateful algorithm guaranteed by Definition A.2. Fix a t-query T-time ℓ-bit-output A, and let $(A_i)_{i\in[\ell]}$ be the algorithms that each output a bit of A's ℓ-bit output. We can write:

$$\begin{aligned}
&\Delta\left(A^{\mathcal{C}^{\boldsymbol{g}},\boldsymbol{g}}, A^{\boldsymbol{f},\mathcal{S}_\mathcal{C}^{\boldsymbol{f}}}\right) \\
&\leq \sum_{i\in[\ell]} \Delta\left(A_i^{\mathcal{C}^{\boldsymbol{g}},\boldsymbol{g}}, A_i^{\boldsymbol{f},\mathcal{S}_\mathcal{C}^{\boldsymbol{f}}}\right) && \text{(by sub-additivity of statistical distance)} \\
&= \sum_{i\in[\ell]} \left|\Pr\left[A_i^{\mathcal{C}^{\boldsymbol{g}},\boldsymbol{g}} = 1 \,\middle|\, \boldsymbol{g} \leftarrow \mathcal{D}_\diamond(\lambda, n)\right] - \Pr\left[A_i^{\boldsymbol{f},\mathcal{S}_\mathcal{C}^{\boldsymbol{f}}} = 1 \,\middle|\, \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n)\right]\right| \\
& && \text{(by definition of statistical distance)} \\
&\leq \sum_{i\in[\ell]} \varepsilon_{\text{ind}}(\lambda, n, t, T) && \text{(by applying Definition A.2 to each distinguisher } A_i\text{)} \\
&= \ell \cdot \varepsilon_{\text{ind}}(\lambda, n, t, T) .
\end{aligned}$$

□

Remark A.5 (oracle construction for the duplex sponge)**.** Let $\mathsf{IP} = (\mathbf{P}, \mathbf{V})$ be a public-coin IP and let $\mathcal{D}_{\mathsf{IP}}$ be the corresponding oracle distribution for some salt size δ (see Equation 4). Let $\mathcal{D}_{\mathfrak{S}}$ be an ideal permutation oracle distribution (see Definition 4.2). Let cdc a codec for IP over the alphabet Σ of $\mathcal{D}_{\mathfrak{S}}$ (see Definition 4.1). We describe a $(\mathcal{D}_{\mathsf{IP}}, \mathcal{D}_{\mathfrak{S}})$-construction $\mathcal{C}_{\mathrm{DSFS}}$ that realizes $\mathbf{DSFS}[\mathsf{IP}, \mathsf{cdc}, \delta]$ in Construction 4.3. Below we use syntax from the ideal duplex sponge described in Section 3.3.

```
$\mathcal{C}^{h,p}_{\mathrm{DSFS}}$:
1. Initialize k empty lists $(L_i)_{i \in [\mathsf{k}]}$.
2. All queries to $h$ or $p$ below are memoized so any duplicate queries are answered internally.
3. Upon receiving a query $x$ for $f_i$, if there is $y$ such that $(x, y) \in L_i$ then answer with $y$.
   Otherwise (the query is new so) proceed as follows:
   (a) Parse $x$ as $(\mathbb{x}, \tau, \alpha_1, \dots, \alpha_i)$ (abort if this cannot be done).
   (b) If parsing is possible do the following:
       i. Initialize the sponge state with the instance: $\mathsf{st}'_0 := \mathsf{DS.Start}^h(\mathbb{x})$.
       ii. Absorb the salt: $\mathsf{st}_0 := \mathsf{DS.Absorb}^p(\mathsf{st}'_0, \boldsymbol{\tau})$ where $\boldsymbol{\tau}$ is $\tau$ represented over $\Sigma$.
       iii. For $j = 1, \dots, i$:
            A. Encode the prover message: $\widehat{\boldsymbol{\alpha}}_j := \varphi_j(\alpha_j)$.
            B. Absorb the encoded prover message: $\mathsf{st}'_j := \mathsf{DS.Absorb}^p(\mathsf{st}_{j-1}, \widehat{\boldsymbol{\alpha}}_j)$.
            C. Squeeze the encoded verifier message: $(\widehat{\boldsymbol{\rho}}_j, \mathsf{st}_j) := \mathsf{DS.Squeeze}^p(\mathsf{st}'_j, \ell_{\mathbf{V}}(j))$.
            D. Decode the verifier message: $\rho_j := \psi_j(\widehat{\boldsymbol{\rho}}_j)$.
   (c) Add $(x, \rho_i)$ to $L_i$, and answer the query with $\rho_i$.
```

Note that

$$\mathcal{V}^{h,p}(\mathbb{x}, \pi) \equiv \mathcal{V}_{\mathsf{std}}^{\mathcal{C}^{h,p}_{\mathrm{DSFS}}}(\mathbb{x}, \pi)\,.$$

A.2 Limitations of indifferentiability

Let $\mathsf{NARG} = (\mathcal{P}, \mathcal{V})$ be a non-interactive argument for relation $\mathcal{R}$ in the $\mathcal{D}$-oracle model (see Section 3.4) and let $\mathcal{C}$ be a $(\mathcal{D}, \mathcal{D}_\diamond)$-construction (see Definition A.1). Given an oracle construction $\mathcal{C}$, define a new non-interactive argument $\mathsf{NARG}_\diamond := (\mathcal{P}_\diamond, \mathcal{V}_\diamond)$ in the $\mathcal{D}_\diamond$-oracle model where, for every $\boldsymbol{g} \in \mathcal{D}_\diamond(\lambda, n)$,

$$\mathcal{P}^{\boldsymbol{g}}_\diamond(\mathbb{x}, \mathbb{w}) := \mathcal{P}^{\mathcal{C}^{\boldsymbol{g}}}(\mathbb{x}, \mathbb{w})\,,$$
$$\mathcal{V}^{\boldsymbol{g}}_\diamond(\mathbb{x}, \pi) := \mathcal{V}^{\mathcal{C}^{\boldsymbol{g}}}(\mathbb{x}, \pi)\,.$$

We discuss how security properties of NARG impact security properties of $\mathsf{NARG}_\diamond$, by reasoning about the indifferentiability of $\mathcal{C}$ (see Definition A.2).

- In Appendix A.2.1 we see that indifferentiability suffices to establish **soundness**.
- In Appendix A.2.2 we see that indifferentiability *does not suffice* to establish **knowledge soundness**.
- In Appendix A.2.3 we see that indifferentiability *does not suffice* to establish **zero knowledge**.

Throughout, we assume that $\mathcal{S}_\mathcal{C}$ (from Definition A.2) is query-preserving, i.e., $\mathcal{S}^{\boldsymbol{f}}_\mathcal{C}$ makes at most the same number of queries as it answers. (The discussion can be straightforwardly adapted to account for a parameter that accounts for $\mathcal{S}^{\boldsymbol{f}}_\mathcal{C}$'s query complexity in terms of the queries it answers.)

A.2.1 Soundness

The soundness error of $\mathsf{NARG}_\diamond$ can be upper bounded by the soundness error of NARG plus the indifferentiability error ε_{ind}. More precisely, two different arguments yield incomparable upper bounds with different parameters: (i) a direct argument relies on a distinguisher whose running time depends on the time to decide the language $\mathcal{L}(\mathcal{R})$ (possibly exponential time); and (ii) an alternate argument, going through strong indifferentiability (Definition A.3), incurs a multiplicative loss of n (the instance size bound).

Lemma A.6. *If* NARG *has soundness error* $\varepsilon_{\mathsf{NARG}}(\lambda, t, n)$ *then* $\mathsf{NARG}_\diamond$ *has soundness error* $\varepsilon_{\mathsf{NARG}_\diamond}(\lambda, t, n)$ *that satisfies two (incomparable) upper bounds:*

$$\varepsilon_{\mathsf{NARG}_\diamond}(\lambda, t, n) \leq \varepsilon_{\mathsf{NARG}}(\lambda, t, n) + \varepsilon_{\text{ind}}(\lambda, n, t + t_{\mathcal{V}}, T + T_{\mathcal{V}} + T_{\mathcal{L}}) ,$$

and

$$\begin{aligned}\varepsilon_{\mathsf{NARG}_\diamond}(\lambda, t, n) &\leq \varepsilon_{\mathsf{NARG}}(\lambda, t, n) + \varepsilon_{\text{sind}}(\lambda, n, t + t_{\mathcal{V}}, T + T_{\mathcal{V}}, n + |\pi| + 1) \\ &\leq \varepsilon_{\mathsf{NARG}}(\lambda, t, n) + (n + |\pi| + 1) \cdot \varepsilon_{\text{ind}}(\lambda, n, t + t_{\mathcal{V}}, T + T_{\mathcal{V}}) .\end{aligned}$$

Proof. We prove the two bounds separately; the second inequality in the second bound is by Lemma A.4.

Bound via ε_{ind}. The soundness property (Definition 3.5) is a predicate applied to the tuple $(\mathbb{x}, \pi, b)$ sampled according to the experiment. Fix a t-query T-time malicious argument prover $\tilde{\mathcal{P}}_\diamond$ against $\mathcal{V}_\diamond$. Define the adversary D that, given two oracles $\boldsymbol{o}_1, \boldsymbol{o}_2$, works as follows.

$D^{\boldsymbol{o}_1, \boldsymbol{o}_2}$:

1. Compute $(\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}_\diamond^{\boldsymbol{o}_2}$.
2. Compute $b \leftarrow \mathcal{V}^{\boldsymbol{o}_1}(\mathbb{x}, \pi)$.
3. Check that $|\mathbb{x}| \leq n$, $\mathbb{x} \notin \mathcal{L}(\mathcal{R})$, $b = 1$.
4. If so, output 1; else output 0.

Note that D makes $t + t_{\mathcal{V}}$ queries and runs in time $T + T_{\mathcal{V}} + T_{\mathcal{L}}$. Therefore we can write

$$\begin{aligned}
&\Pr\left[\begin{array}{l|l} |\mathbb{x}| \leq n & \boldsymbol{g} \leftarrow \mathcal{D}_\diamond(\lambda, n) \\ \wedge\, \mathbb{x} \notin \mathcal{L}(\mathcal{R}) & (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}_\diamond^{\boldsymbol{g}} \\ \wedge\, b = 1 & b \leftarrow \mathcal{V}_\diamond^{\boldsymbol{g}}(\mathbb{x}, \pi) \end{array}\right] \\
&= \Pr\left[\begin{array}{l|l} |\mathbb{x}| \leq n & \boldsymbol{g} \leftarrow \mathcal{D}_\diamond(\lambda, n) \\ \wedge\, \mathbb{x} \notin \mathcal{L}(\mathcal{R}) & (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}_\diamond^{\boldsymbol{g}} \\ \wedge\, b = 1 & b \leftarrow \mathcal{V}^{\mathcal{C}^{\boldsymbol{g}}}(\mathbb{x}, \pi) \end{array}\right] && \text{(by construction of } \mathcal{V}_\diamond\text{)} \\
&\leq \Pr\left[\begin{array}{l|l} |\mathbb{x}| \leq n & \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n) \\ \wedge\, \mathbb{x} \notin \mathcal{L}(\mathcal{R}) & (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}_\diamond^{\mathcal{S}_{\mathcal{C}}^{\boldsymbol{f}}} \\ \wedge\, b = 1 & b \leftarrow \mathcal{V}^{\boldsymbol{f}}(\mathbb{x}, \pi) \end{array}\right] + \varepsilon_{\text{ind}}(\lambda, n, t + t_{\mathcal{V}}, T + T_{\mathcal{V}} + T_{\mathcal{L}}) \\
& && \text{(by applying Definition A.2 to } D\text{)} \\
&= \varepsilon_{\mathsf{NARG}}(\lambda, t, n) + \varepsilon_{\text{ind}}(\lambda, n, t + t_{\mathcal{V}}, T + T_{\mathcal{V}} + T_{\mathcal{L}}) .
\end{aligned}$$

Bound via $\varepsilon_{\text{sind}}$. The soundness property (Definition 3.5) is a predicate applied to the tuple $(\mathbb{x}, \pi, b)$ sampled according to the experiment. Fix a t-query T-time malicious argument prover $\tilde{\mathcal{P}}_\diamond$ against $\mathcal{V}_\diamond$. Define the adversary A that, given two oracles $\boldsymbol{o}_1, \boldsymbol{o}_2$, works as follows.

$A^{\boldsymbol{o}_1, \boldsymbol{o}_2}$:

1. Compute $(\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}_{\diamond}^{\boldsymbol{o}_2}$.
2. Compute $b \leftarrow \mathcal{V}^{\boldsymbol{o}_1}(\mathbb{x}, \pi)$.
3. Output $(\mathbb{x}, \pi, b)$.

Note that A makes $t + t_{\mathcal{V}}$ queries, runs in time $T + T_{\mathcal{V}}$, and outputs at most $n + |\pi| + 1$ bits. Therefore we can write

$$\begin{aligned}
&\Pr\left[\begin{array}{l|l} |\mathbb{x}| \le n & \boldsymbol{g} \leftarrow \mathcal{D}_{\diamond}(\lambda, n) \\ \wedge\, \mathbb{x} \notin \mathcal{L}(\mathcal{R}) & (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}_{\diamond}^{\boldsymbol{g}} \\ \wedge\, b = 1 & b \leftarrow \mathcal{V}_{\diamond}^{\boldsymbol{g}}(\mathbb{x}, \pi) \end{array}\right] \\
&= \Pr\left[\begin{array}{l|l} |\mathbb{x}| \le n & \boldsymbol{g} \leftarrow \mathcal{D}_{\diamond}(\lambda, n) \\ \wedge\, \mathbb{x} \notin \mathcal{L}(\mathcal{R}) & (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}_{\diamond}^{\boldsymbol{g}} \\ \wedge\, b = 1 & b \leftarrow \mathcal{V}^{\mathcal{C}^{\boldsymbol{g}}}(\mathbb{x}, \pi) \end{array}\right] && \text{(by construction of } \mathcal{V}_{\diamond}\text{)} \\
&\le \Pr\left[\begin{array}{l|l} |\mathbb{x}| \le n & \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n) \\ \wedge\, \mathbb{x} \notin \mathcal{L}(\mathcal{R}) & (\mathbb{x}, \pi) \leftarrow \tilde{\mathcal{P}}_{\diamond}^{\mathcal{S}_{\mathcal{C}}^{\boldsymbol{f}}} \\ \wedge\, b = 1 & b \leftarrow \mathcal{V}^{\boldsymbol{f}}(\mathbb{x}, \pi) \end{array}\right] + \varepsilon_{\mathrm{sind}}(\lambda, n, t + t_{\mathcal{V}}, T + T_{\mathcal{V}}, n + |\pi| + 1) \\
& && \text{(by applying Definition A.3 to } A\text{)} \\
&= \varepsilon_{\mathsf{NARG}}(\lambda, t, n) + \varepsilon_{\mathrm{sind}}(\lambda, n, t + t_{\mathcal{V}}, T + T_{\mathcal{V}}, n + |\pi| + 1)\,.
\end{aligned}$$

□

A.2.2 Knowledge soundness

For simplicity, we focus on straightline knowledge soundness; a similar discussion holds for rewinding knowledge soundness. Indifferentiability (Definition A.2) as well as strong indifferentiability (Definition A.3) do not offer capabilities for translating a query-answer trace from the $\mathcal{D}_{\diamond}$-oracle model to the $\mathcal{D}$-oracle model. This is necessary in order to build a knowledge extractor for $\mathsf{NARG}_{\diamond}$ by relying on a knowledge extractor for NARG. To address this, we propose *double indifferentiability* as a notion that precisely captures what we need; abstractly, this is what we prove in Lemma 5.1 fro deducing the knowledge soundness of **DSFS**.

Definition A.7. *A $(\mathcal{D}, \mathcal{D}_{\diamond})$-construction $\mathcal{C}$ is* **doubly indifferentiable with error** $\varepsilon_{\mathrm{dind}}$ *if there exist a probabilistic stateful algorithm $\mathcal{S}_{\mathcal{C}}$ and a deterministic algorithm $\mathcal{T}_{\mathcal{C}}$ such that for every t-query T-time ℓ-bit-output algorithm A:*

$$\Delta\left(\left\{(z, \mathcal{T}_{\mathcal{C}}(\mathsf{tr}_{\diamond})) \,\middle|\, z \xleftarrow{\mathsf{tr}_{\diamond}} A^{\mathcal{C}^{\boldsymbol{g}}, \boldsymbol{g}}\right\}, \left\{(z, \mathsf{tr}) \,\middle|\, z \xleftarrow{\mathsf{tr}} A^{\boldsymbol{f}, \mathcal{S}_{\mathcal{C}}^{\boldsymbol{f}}}\right\}\right) \le \varepsilon_{\mathrm{dind}}(\lambda, n, t, T, \ell)\,.$$

We prove that the above notion enables upper bounding the knowledge soundness error of $\mathsf{NARG}_{\diamond}$ in terms of the knowledge soundness error of NARG plus the double indefferentiaibility error $\varepsilon_{\mathrm{dind}}$.

Lemma A.8. *If NARG has straightline knowledge soundness error $\kappa_{\mathsf{NARG}}(\lambda, t, n)$ then $\mathsf{NARG}_{\diamond}$ has straightline knowledge soundness error $\varepsilon_{\mathsf{NARG}_{\diamond}}(\lambda, t, n)$ such that*

$$\begin{aligned}
&\kappa_{\mathsf{NARG}_{\diamond}}(\lambda, t, n) \\
&\le \kappa_{\mathsf{NARG}}(\lambda, t, n) + \varepsilon_{\mathrm{dind}}(\lambda, n, t + t_{\mathcal{V}}, T + T_{\mathcal{V}}, n + |\pi| + 1)\,.
\end{aligned}$$

Proof. The second inequality follows from Lemma A.4. We prove the first inequality.

The straightline knowledge soundness property (Definition 3.6) is a predicate applied to the tuple $(\mathbb{x}, \pi, b, \mathsf{tr})$ sampled according to the experiment. Fix a t-query T-time malicious argument prover $\tilde{\mathcal{P}}_\diamond$ against $\mathcal{V}_\diamond$. Letting $\mathcal{E}$ be the straightline extractor for NARG, we define a straightline extractor $\mathcal{E}_\diamond$ for $\mathsf{NARG}_\diamond$ as follows.

```
$\mathcal{E}_\diamond(\mathbb{x}, \pi, \mathsf{tr}_\diamond)$:
1. Compute $\mathsf{tr} \leftarrow \mathcal{T}_\mathcal{C}(\mathsf{tr}_\diamond)$.
2. Compute $\mathbb{w} \leftarrow \mathcal{E}(\mathbb{x}, \pi, \mathsf{tr})$.
3. Output $\mathbb{w}$.
```

Define the adversary A that, given two oracles $\boldsymbol{o}_1, \boldsymbol{o}_2$, works as follows.

```
$A^{\boldsymbol{o}_1, \boldsymbol{o}_2}$:
1. Compute $(\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_\diamond} \tilde{\mathcal{P}}_\diamond^{\boldsymbol{o}_2}$.
2. Compute $b \leftarrow \mathcal{V}^{\boldsymbol{o}_1}(\mathbb{x}, \pi)$.
3. Output $(\mathbb{x}, \pi, b)$.
```

Note that A makes $t + t_\mathcal{V}$ queries, runs in time $T + T_\mathcal{V}$, and outputs at most $n + |\pi| + 1$ bits. Therefore we can write

$$
\Pr\left[\begin{array}{l} |\mathbb{x}| \le n \\ \wedge\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \\ \wedge\ b = 1 \end{array} \middle| \begin{array}{l} \boldsymbol{g} \leftarrow \mathcal{D}_\diamond(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_\diamond} \tilde{\mathcal{P}}_\diamond^{\boldsymbol{g}} \\ \mathbb{w} \leftarrow \mathcal{E}_\diamond(\mathbb{x}, \pi, \mathsf{tr}_\diamond) \\ b \leftarrow \mathcal{V}_\diamond^{\boldsymbol{g}}(\mathbb{x}, \pi) \end{array}\right]
$$

$$
= \Pr\left[\begin{array}{l} |\mathbb{x}| \le n \\ \wedge\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \\ \wedge\ b = 1 \end{array} \middle| \begin{array}{l} \boldsymbol{g} \leftarrow \mathcal{D}_\diamond(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}_\diamond} \tilde{\mathcal{P}}_\diamond^{\boldsymbol{g}} \\ \mathsf{tr} \leftarrow \mathcal{T}_\mathcal{C}(\mathsf{tr}_\diamond) \\ \mathbb{w} \leftarrow \mathcal{E}(\mathbb{x}, \pi, \mathsf{tr}) \\ b \leftarrow \mathcal{V}^{\mathcal{C}^{\boldsymbol{g}}}(\mathbb{x}, \pi) \end{array}\right] \qquad \text{(by construction of } \mathcal{V}_\diamond \text{ and } \mathcal{E}_\diamond\text{)}
$$

$$
\le \Pr\left[\begin{array}{l} |\mathbb{x}| \le n \\ \wedge\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \\ \wedge\ b = 1 \end{array} \middle| \begin{array}{l} \boldsymbol{f} \leftarrow \mathcal{D}(\lambda, n) \\ (\mathbb{x}, \pi) \xleftarrow{\mathsf{tr}} \tilde{\mathcal{P}}_\diamond^{\mathcal{S}_\mathcal{C}^{\boldsymbol{f}}} \\ \mathbb{w} \leftarrow \mathcal{E}(\mathbb{x}, \pi, \mathsf{tr}) \\ b \leftarrow \mathcal{V}^{\boldsymbol{f}}(\mathbb{x}, \pi) \end{array}\right] + \varepsilon_{\mathrm{dind}}(\lambda, n, t + t_\mathcal{V}, T + T_\mathcal{V}, n + |\pi| + 1)\,.
$$

(by applying Definition A.7 to A)

□

A.2.3 Zero knowledge

Let NARG and $\mathsf{NARG}_\diamond$ be as in Appendix A.2. The zero-knowledge property of NARG states that the following two distributions are $z_{\mathsf{NARG}}(\lambda, t, n)$-close in statistical distance (for every t-query admissible adversary $\mathcal{A}$):

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

Analogously, the zero-knowledge property of $\mathsf{NARG}_\diamond$ states that the following two distributions are $z_{\mathsf{NARG}_\diamond}(\lambda, t, n)$-close in statistical distance (for every t-query admissible adversary $\mathcal{A}$):

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

Our goal is to derive an upper bound on $z_{\mathsf{NARG}_\diamond}$ in terms of z_{NARG} plus some additive error associated to the oracle construction $\mathcal{C}$. This would also entail (somehow) constructing the new simulator $\mathcal{S}_\diamond$ for $\mathsf{NARG}_\diamond$ in terms of the old simulator $\mathcal{S}$ for NARG, for example as follows:

$\mathcal{S}_\diamond^{\boldsymbol{g}}(\mathbb{x})$:
1. Compute $(\pi, \mu) \leftarrow \mathcal{S}^{\mathcal{C}^{\boldsymbol{g}}}(\mathbb{x})$.
2. Remap (somehow) the programmed locations μ for $\boldsymbol{f}$ to programmed locations $\mu_\diamond$ for $\boldsymbol{g}$.
3. Output $(\pi, \mu_\diamond)$.

However, indifferentiability does not provide any guarantees about the behavior of an adversary when an oracle is programmed (nor do the strengthenings we considered in Definition A.3 and Definition A.7).

One way to address this would be to augment the notion of indifferentiability to include programming. However this would yield an additive error that grows at least quadratically in t, whereas the "right" additive error is a linear growth in t. We do not know of a property that allows for such a loss while generically transfering the zero knowledge property from NARG to $\mathsf{NARG}_\diamond$. This is why we prove the zero knowledge property of **DSFS** directly.

B 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 $\mathbb{G}$.[15] Let $\mathbb{Z}_p$ be the coordinate field of the elliptic curve. Schnorr's protocol is a 3-message IP for the relation $\mathcal{R} := \big\{(X, x) \in \mathbb{G} \times \mathbb{Z}_p : X = xG\big\}$. 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 C.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 C.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 3.3). 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 $i_{\text{S}} = 0$, and then reads the first element of the rate segment in the permutation state.

[15]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).

C 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 C.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

[AB24] T. Ashur and A. S. Bhati. *Generalized Indifferentiable Sponge and its Application to Polygon Miden VM*. Cryptology ePrint Archive, Paper 2024/911. 2024.

[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`.

[AKMQ23] J. Aumasson, D. Khovratovich, B. Mennink, and P. Quine. *SAFE: Sponge API for Field Elements*. Cryptology ePrint Archive, Paper 2023/522. 2023.

[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.

[BR06] M. Bellare and P. Rogaway. "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs". In: *Proceedings of the 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '06. 2006. DOI: 10.1007/11761679_25.

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

[Can01] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science*. FOCS '01. 2001, pp. 136–145.

[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`.

[dus] dusk-network. *safe: Sponge API for Field Elements*. URL: `https://github.com/dusk-network/safe`.

[Eth24] Ethereum Foundation. *zkEVM Formal Verification Project*. `https://verified-zkevm.org/`. 2024.

[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`.

[KMM23] D. Khovratovich, M. Marhuenda Beltrán, and B. Mennink. “Generic Security of the SAFE API and Its Applications”. In: *Proceedings of the 29th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT 2023. 2023, pp. 301–327.

[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.

[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`.