NPRFs and their Application to MLS

Chris Brzuska[1], Jan Winkelmann

Aalto University

Abstract. Noise, Signal and TLS 1.3 use key derivation functions (KDF) and pseudorandom functions (PRF) to combine key material to obtain a secure key whenever at least one of the input keys is secure. Security analyses of the aforementioned protocols either analyze the protocol in the random oracle model or, if only two keys are combined at a time, assume that in the KDF/PRF, the role of the key and the message are interchangeable. Thus, protocols typically rely on the ad hoc assumption of a *dual* KDF or a *dual* PRF (DPRF).

To combine more than two secrets, state-of-the-art protocols apply the DPRF multiple times sequentially, each time evaluating it on one new secret as well as piping in the result of the previous DPRF call. The resulting number of sequential DPRF evaluations is thus linear in the number of keys to be combined.

We propose n-pseudorandom functions (NPRFs) as a new primitive which combines an *arbitrary* number of keys. We provide a security model for our multi-instance, multi-key primitive. We then provide a practical construction of NPRFs which is parallelizable and, in each of the parallel branches (one for each secret), requires only 1 PRF evaluation, independently of the number of keys to be combined. It is based on the standard assumption that HMAC is a PRF.

We compare the security of our NPRF construction with the aforementioned piping construction under collision attacks, i.e., colliding key values, colliding key names and colliding context values. By adding one or two more PRF evaluations in each of the parallel branches, we improve the security of our basic NPRF to be on the same or higher level of collision-resistance than the aforementioned piping construction. These extended constructions additionally assume that salted HMAC is a collision-resistant hash-function.

As a case study, we explore key derivation in the current draft of the Message Layer Security (MLS) IETF standard, and provide justification for pull request 337 [1] which is a change to the MLS standard that we suggest, based on our NPRF construction.

[1] https://github.com/mlswg/mls-protocol/pull/337

1 Introduction

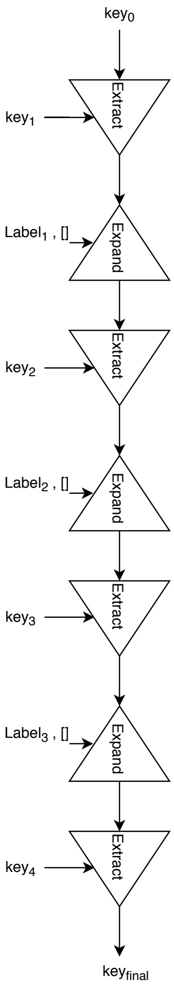


Fig. 1: NPRF in practice.

Aiming for security under strong adversarial corruption capabilities, novel key exchange protocols combine key material in complex ways and aim for security as long as at least one of the input keys is secure, or *honest* in the language of the recent TLS 1.3 analysis by Brzuska, Delignat-Lavaud, Fournet, Kohbrok and Kohlweiss [BDF$^+$]. For example, Signal [CCD$^+$16] and the Noise Framework [DRS20] are based on static and ephemeral Diffie-Hellman (DH) keys for each party and provide security if at least one out of the static-static, static-ephemeral, ephemeral-ephemeral combinations is honest. TLS 1.3 [Res18,BDF$^+$] implements a sophisticated combiner: Taking session resumption into account, the mixed mode of TLS-PSK and TLS-DH provides security if either the DH secret is honest or the current PSK is honest, where the honesty of the PSK is computed recursively, based on whether the previous DH secret or previous PSK was honest. MLS allows to combine an ephemeral secret with a pre-shared key.

The aforementioned protocols require a *pseudorandomness combiner* which returns a pseudorandom key whenever at least one of the input keys was (pseudo-)random and secret from the adversary. Additionally, many protocol designs also aim to avoid collisions on the resulting key, e.g., to prevent unknown key-share attacks (page 234 in [JKSS10]).

The de-facto state-of-the-art methodology for real-life protocols is to use a variant of the `Extract`-then-`Expand` approach, suggested by Krawczyk in his design of HKDF [Kra10]. Namely, `Extract` is modeled as a computational extractor which takes as input a public, uniformly distributed salt and initial key material with high computational average min-entropy and returns a pseudorandom value. Protocols such as TLS, Noise and the current draft of MLS combine two keys by making a call to the `Extract` function of HKDF, using one key as a salt and the other key as initial key material. When combining several keys, the current MLS draft makes sequential calls to `Extract`, each time piping in the previous key value as one of the inputs. Consistently with the `Extract`-

then-Expand design, the calls to Extract are interleaved with calls to Expand. See Figure 1 for this construction principle.

TLS 1.3 also makes three sequential calls[2] to Extract and interleaves the sequential calls with calls to Expand in order to safely derive further keys from intermediate secrets.

Comparison of real and originally intended use There is a discrepancy between using Extract as a dual PRF and the original design rationale underlying the Extract-then-Expand design [Kra10]. Namely, the original design rationale considers the following scenario for Extract:

(E1) Key material is used a *single* time.
(E2) The salt is chosen *uniformly* at random and *independently* of the key material.
(E3) The salt is public.

In practice, two (or more) parties in a protocol perform operations on key material and, in case of an active attack, might use the same key material with several, different salts the latter of which are partially under adversarial control and might depend on the secret. Thus, in the aforementioned use cases of the Extract function, any of the following can occur:

(P1) Key material is used twice or more.
(P2) The salt is partially under adversarial control and can depend on the key material.
(P3) Rather than on the secrecy of the key material, we rely on the secrecy of a uniformly random salt S and treat Extract$(S, .)$ as a PRF.

Extract is typically implemented by HMAC, and similarly, Expand is typically implemented by HMAC. For Expand, HMAC is assumed to be a PRF [Kra10], and, in fact, this assumption is equivalent to (P3). Thus, the Extract-then-Expand already relies on (P3) being true, and it is not an additional assumption[3]. In their article on OPTLS [KW15], Krawczyk and Wee also explain that (P3) is a reasoneable assumption for HMAC in addition to the assumption that HMAC is a strong extractor. Interestingly, in their proof of OPTLS, they rely on (P3) and on the *single-use, random salt* security of the extractor (unlike what is mentioned in (P1)

[2] The first call extracts from a pre-shared key, the second call feeds in the Diffie-Hellman secret and the third call to Extract —currently feeding a constant— ostensibly supports later integration of an additional post-quantum secure secret.

[3] Note that this assumption is usually only made for Expand and not for Extract, blurring the lines between the purpose of the two.

and (P2)). I.e., their argument works provided that there are other cryptographic mechanisms which ensure that the salt is pseudorandom despite adversarial inference and prevent the adversary from modifying the transcript (or at least the part of the transcript which affects the computation of the key and the salt). In the absence of such mechanisms (or if such mechanisms fail or if authentication is performed only later), one needs to provide security in a setting where key material is used multiple times. Here, the assumption of a single-use extractor does not suffice. Which other assumptions could be used instead?

To answer this question, an important note on the OPTLS design is that the two keys which are input to the `Extract` function in OPTLS already come from calls to `Extract`-then-`Expand`. Barak, Dodis, Krawczyk, Pereira, Pietrzak, Standaert, and Yu Yu [BDK$^+$11] point out that extractor assumptions might break down (at least in theory) when the adversary knows the pre-image of a salt which was generated by an `Expand` call. However, when the initial keying material is already a secret and pseudorandom value, then one might simply rely on the pseudorandomness of the initial keying material. I.e., one might want assume that `Extract` is a good pseudorandom function in a *dual* sense, i.e., regardless of whether the salt or the keying material is treated as a key of the function.

In TLS 1.3 [Res18] (different from its predecessor OPTLS), one needs to additionally assume that security even holds when the key material is not uniform [BDF$^+$], since Diffie-Hellman keys are not extracted before they are used in a dual PRF. This slight weakness in the design, potentially, is due to the misnomer of calling the dual pseudorandom function `Extract`. It might be useful to rename such calls to HMAC `DualExpand` rather than `Extract`, since this seems to more accurately reflect the assumptions.

Dual pseudorandomness of HMAC for uniformly random keys might be a plausible assumption, but it is not yet widely studied. The suggestions made in this paper, make this assumption unnecessary for newly designed protocols and, in addition, allows us to efficiently combine more than two keys, as is needed in MLS, Signal and Noise.

Goal We design an n-input pseudorandom function for any $n \in \mathbb{N}$ (NPRF) for the use-case of combining multiple keys. In particular, our NPRF

(N1) takes an arbitrary number of keys.

(N2) returns a pseudo-random key whenever at least one input key was (pseudo-)random and secret.

(N3) is built on the (standard) pseudorandomness and collision-resistance properties of HMAC.

(N4) has equal or higher parallel and overall efficiency than the current, popular NPRF construction (Fig. 1).

We present our NPRF construction in Figure 2. We suggest to use a unique context value *ctx* to derive a key value from each of the input keys and to xor the result. Here, *ctx* needs to be *the same* value in all of the `Expand` calls, thereby foregoing the multiple use of the same key material with different keys, since changing *ctx* in one position changes it in all other positions as well, thus leading to new key material in all calls. *Uniqueness* of the *ctx* means that the NPRF shall not be called with the same *ctx* twice.

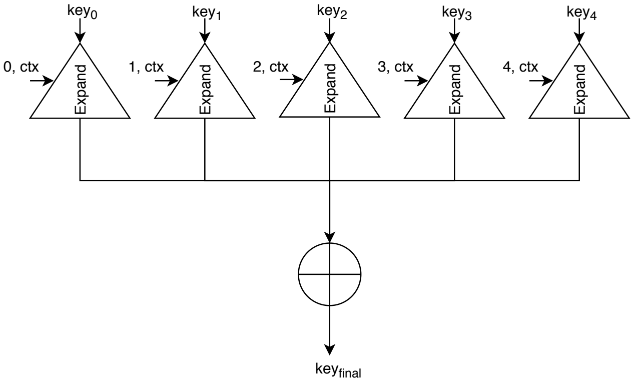


Fig. 2: New NPRF construction.

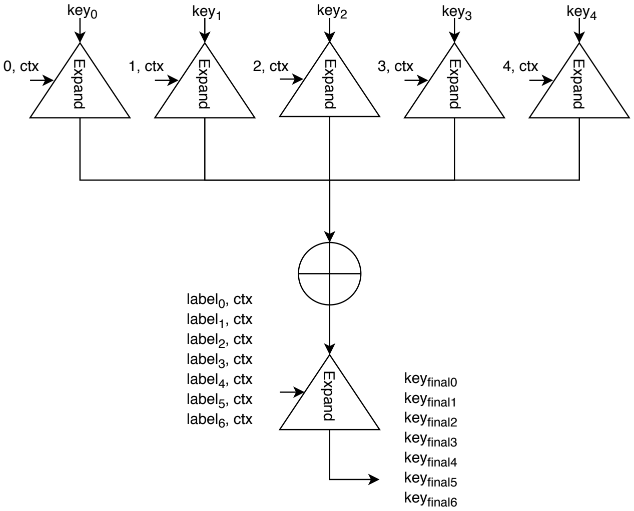


Fig. 3: crNPRF deriving multiple keys.

This construction combines an arbitrary number of keys, has low parallel complexity, and its security follows from the PRF-security of the `Expand` function. In TLS 1.3, the piping construction in Figure 1 expands some of the intermediate keys to export early traffic keys (which might justify the piping to some extent). As the key schedule in MLS does not export early keys, the intermediate keys in MLS might not be needed. The construction in Figure 2 is secure against re-ordering attacks since the Expand function takes the *position* as part of the input. Such re-ordering attacks of the xor-combiner in a similar dual PRF construction were previously observed by Bellare and Lysanskaya [BL15] and are prevented by including the position. As both Expand and Extract are typically implemented via an HMAC evaluation, let us compare the complexity of the construction in Figure 2 and the construction in Figure 1 by their number of HMAC evaluations. The construction in Figure 1 has 9 *sequential* HMAC evaluations while our construction only has 5 *parallel* HMAC evaluations.

Security against colliding outputs and derivation of multiple keys Xor is not a collision-resistant operation. When all input keys are known to the adversary, the adversary might be able to trigger a collision on the output key even when the input keys to the construction are unique. Thus, one can apply an additional Expand operation on the output key which, again, uses the *same* unique context value *ctx* as the one which was used in the Expand step before the xor. In summary, the first time, the uniqueness of the context is used to obtain a pseudorandom key, and the second time, the uniqueness of the context is used to obtain uniqueness of the output. We refer to this primitive as *collision-resistant* NPRF (crNPRF). See Figure 3 for a construction. TLS 1.3 and MLS each derive multiple keys from the same secret, in line with the intended purpose of the Expand function. Thus, the final Expand operation simultaneously achieves uniqueness of outputs *and* allows us to derive multiple keys. One should use the pair $(Label, ctx)$ as context value for the final

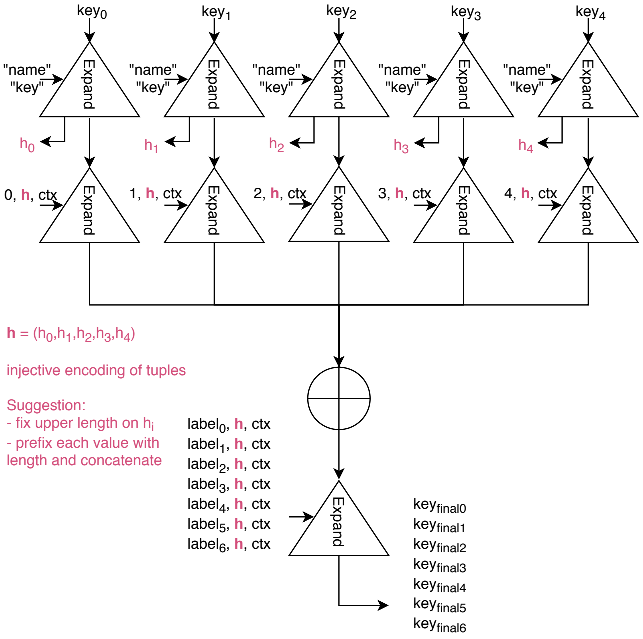


Fig. 4: crNameNPRF.

`Expand` call with different values *Label* and the same *ctx* value, see bottom of Figure 3.

Security against colliding context values and inputs The security of the construction in Figure 2 relies crucially on the uniqueness of the context value *ctx*. In turn, standard DPRF assumptions typically do not have the requirement of such a unique *ctx* value and instead rely on the uniqueness of the input keys. To which extent is it hard, in practice, to find such unique values *ctx*? In the case that we have unique keys, can we perhaps harvest those to obtain unique *ctx* values?

Interestingly, if we have a *unique public name* for each key (Here, uniqueness of the name refers to each name only pointing to a single key.), the ordered list of such public names of keys can serve as *ctx*. If one derives keys from asymmetric cryptography such as a Diffie-Hellman secret, public-keys naturally give rise to names of secrets. E.g., a suitable public name for the secret $\texttt{Hash}(salt, g^{xy})$ would be $(salt, X, Y)$ with $X = g^x$ and $Y = g^y$. These public names collide if and only if the secret keys collides. The same holds if one derives keys only based on symmetric cryptography: Namely, one can preceed the crNPRF computation by an additional `Expand` to derive a public name for each key. This name is unique if and only if the input keys are unique, assuming collision-resistance of `Expand`. See Figure 4 for the resulting construction which we refer to as crNameNPRF.

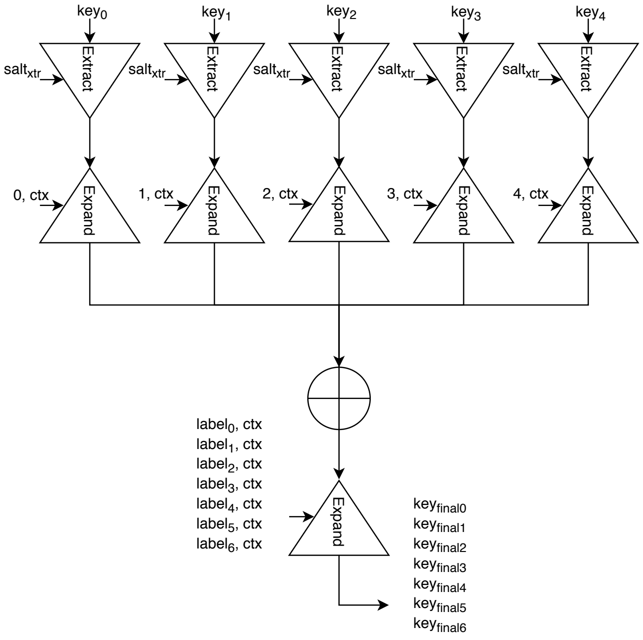


Fig. 5: crNKDF.

Generalization to arbitrary key material We define composable security notions for our primitives using state separation, as suggested in [BDF+18].

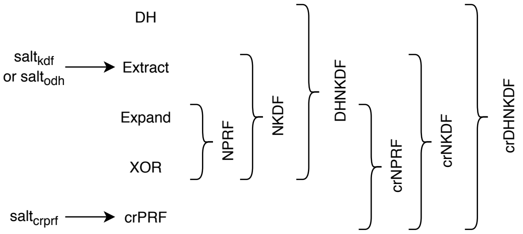


Fig. 6: Overview over constructions

We show that NPRFs, crNPRFs and their named variant crNameNPRFs are composable with arbitrary key material. I.e., if the key material has high entropy, but is not pseudorandom, then one can first use an `Extract` step to obtain a pseudorandom key, thus obtaining *key derivation function* (KDF) variants of the NPRF concept. We call the resulting notions NKDF, crNKDF and crNameNKDF, respectively. We illustrate the crNKDF in Figure 5.

Alternatively, the keys might also be obtained from a Diffie-Hellman secret; we call the resulting construction a DHNKDF or crDHNKDF. We chose to include the discussion of Diffie-Hellman keys, since they are a common source of key material and require the protocols to rely on the Oracle Diffie-Hellman (ODH) assumption [ABR01,BFGJ17] which leads to a slightly different analysis than the (cr)NKDF case. We return to this matter in Section 3.3.

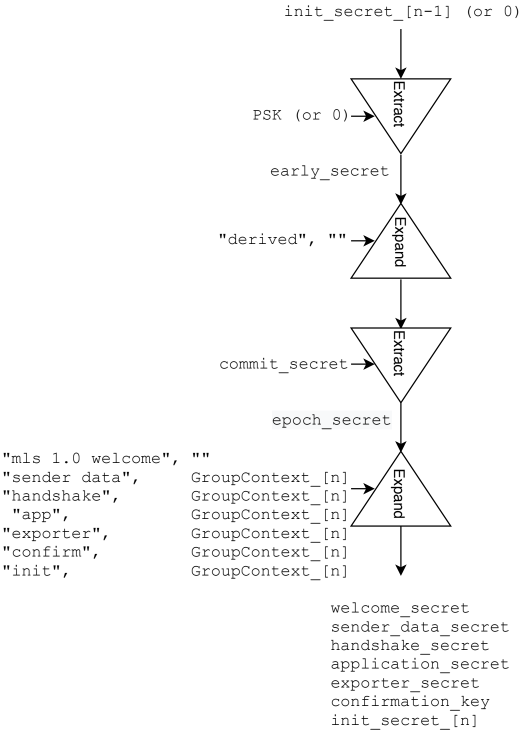


Fig. 7: Current MLS key derivation.

Note that the Name transformation is not needed for (cr)DHNKDF, as the canonically ordered pair of public Diffie-Hellman shares always constitutes a useful public name for a key and thus, derivation of a public name from a symmetric key is not needed. Figure 6 provides an overview over the notions and constructions which we suggest. The figure omits the NameNKDF and NameNPRF variants which add a simple pre-processing step applied to the symmetric key material. Observe in Figure 6 the analogy between extraction from Diffie-Hellman key material and non-uniform key material. We here continue to follow the KDF principle of *universal applicability*.

Mixed key material Our analysis also extends to the case that the key material is mixed, i.e., some might be drawn uniformly at random, some might be extracted from a high-entropy secret and some might be derived from a Diffie-Hellman secret. We return to mixed material in our case study of the MLS key derivation which follows next.

Application to MLS In Pull Request 337[4], we suggest to replace the current piping construction by our crNPRF construction. Figure 7 depicts the current construction and Figure 8 depicts our suggestion according to Pull Request 337, a straightforward adoption of the previously discussed crNPRF approach. MacMillion raised the concern that key material might not be uniformly distributed[5]. We agree that this applies to the PSK, because we do not have control over its generation. We suggest to add an `Extract` step for pre-processing of the PSK. We here suggest to sample a random *fixed* salt and hardcode it into the protocol description of MLS at the point of protocol standardization. This avoids multiple use the PSK key material, as this would violate the single-use security of the computational extractor. Of course, a fixed salt could potentially allow the adversary to influence the PSK generation in a way which depends on the salt, but this seems less plausible/risky than the dangers of multiple extraction from the same key material. It does not seem necessary to add an `Extract` step for pre-processing of the other key material, since the other key

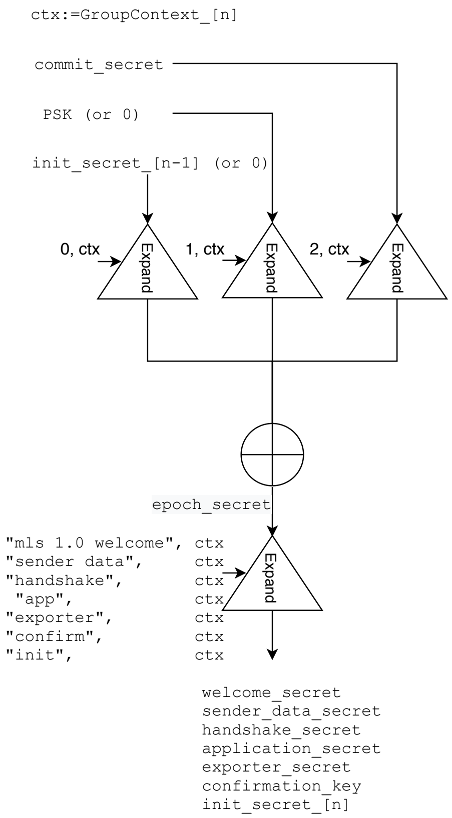


Fig. 8: MLS crNPRF.

[4] https://github.com/mlswg/mls-protocol/pull/337

[5] https://github.com/mlswg/mls-protocol/pull/337#pullrequestreview-422971601

material is either (1) derived from other key material via HMAC and thus already pseudorandom or (2) under adversarial control and thus would not become pseudorandom even when `Extract` is applied to it. We depict this suggestion in Figure 9.

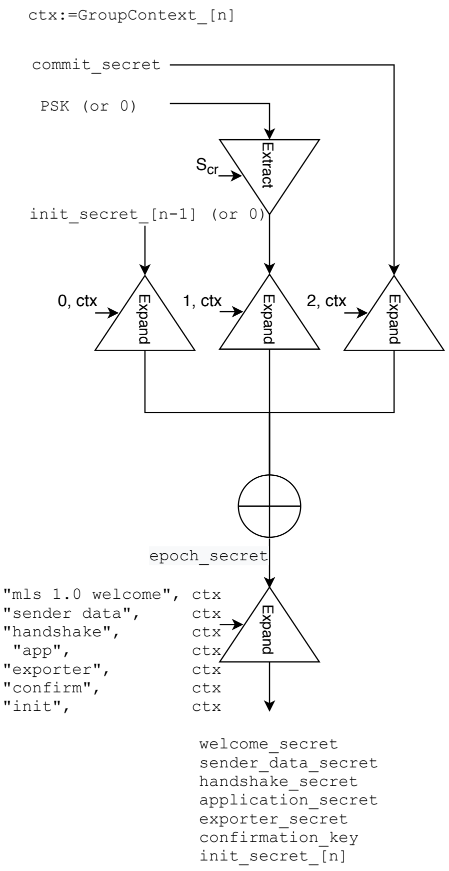


Fig. 9: MLS crNPRF.

Barnes and MacMillion[6] both suggested the use of a protocol variable called the *group context* as unique context value. None of the other working group members raised concern regarding the additional requirement of a unique value. Thus, at the current point (June 9, 2020), we would conclude that the additional `Expand` step of the NameNPRF construction is not needed. In conclusion, the construction in Figure 9 as represents a suggestion which addresses all concerns voiced so far. (Further comments pointed out typos and suggested a different encoding of tuples. From our perspective, any injective encoding of tuples is suitable. We do not have a preference, as it does not concern the cryptographic core of our proposal.)

Efficiency Comparison The crNPRF/KDF construction depicted in Figure 9 has depth 2 or 3 in terms of HMAC evaluations, and this number is independent of the number of keys which are combined. Additionally, a log number of sequential xor operations need to be performed, i.e., logarithmic in the number of keys to be combined. The total number of HMAC evaluations is between $n + m$ and $2n + m$, depending on the number n of keys to be combined and whether or not they need an extraction step as well as the number m of keys which are derived. In comparison, the construction in Figure 7 uses $2(n - 1) + 1$ sequential HMAC invocations and $2(n - 1) + m$ HMAC evaluations overall. For the specific cases we chose to depict, $n = 3$, $m = 7$, our construction requires 11 HMAC evaluations,

[6] https://github.com/mlswg/mls-protocol/pull/337#discussion_r427620764

and the previous construction required 10 HMAC evaluations, increasing the total number of HMAC evaluations by 1.

Assumption Comparison Our construction requires a unique context value which the current construction does not require. The current construction assumes that `Extract` is a DPRF and, in the case the key material is not uniform, also assumes that the DPRF works in that case. In turn, our construction only assumes that HMAC is a PRF, and our extraction step uses the key material only once, provided that the group context value does not repeat. To be able to use a standard extraction assumption and to avoid double-extraction, we can fix a uniformly random salt value at protocol standardization time. We then rely on the fact that the PSK is generated independently from the salt. If collision-resistance of the key derivation shall be proved, both constructions equally rely on the collision-resistance of HMAC.

Outline of the paper We use modular code-writing and state-separating proofs by Brzuska, Delignat-Lavaud, Fournet, Kohbrok and Kohlweiss (BDFKK [BDF$^+$]). We provide the relevant background in Section 2. Section 3 defines the assumptions on which our constructions rely in a modular way. Section 4 contains the security notions for (cr)NPRF, (cr)NKDF and (cr)DHNKDF. Section 5 specifies the corresponding 6 constructions, and Section 6 provides the security proofs for our constructions.

2 Composed Indistinguishability Games

We use pseudocode to specify security games. We use $x \leftarrow a$ to assign value a to variable x and $x \leftarrow\!\!\$\, S$ to sample x uniformly from set S. Sets are initialized to the empty set and integers are initialized to 0, unless otherwise specified. Sets and tables are denoted by capital letters.. We use **bold** font to refer to lists or vectors of values, with $\mathbf{bold}_i$ denoting the i-th element. Indexing starts at 0. Lowercase `monospace` refers to functions, and capitalized `Monospace` to oracles of games. Sometimes we evaluate a functions `f` at every element of a list **input** and store the result in a new list **result**, comparable to the `map` function in programming. We write this as $\mathbf{result} \overset{\text{vec}}{\leftarrow} \mathtt{f}(\mathbf{input})$.

We define the two helper functions $\mathtt{pick}(\mathbf{skX}, \mathbf{pkX}, \mathbf{skY}, \mathbf{pkY})$ and $\mathtt{sort}(a, b)$. `pick` is a helper function for dealing with two pairs of key pairs, where some secret keys are not set, i.e. bot. In this case, we need to select the pairs of public and secret keys such that all secret keys are set (and make a default choice in case both are set). `pick` proceeds as follows: It returns an error of the passed lists are not of the same size. Otherwise, it returns a pair of lists $(\mathbf{sk}, \mathbf{pk})$. $\mathbf{sk}$ contains elements of $\mathbf{skX}$ and $\mathbf{skY}$ while $\mathbf{pk}$ consists of items in $\mathbf{pkX}$ and $\mathbf{pkY}$. If there is an i s.t. $\mathbf{skX}_i = \mathbf{skY} = \bot$, `pick` returns an error. If $\mathbf{skX}_i \neq \bot$, $\mathbf{sk}_i = \mathbf{skX}$ and $\mathbf{pk}_i = \mathbf{pkY}$. Else, $\mathbf{sk}_i = \mathbf{skY}$ and $\mathbf{pk}_i = \mathbf{pkX}$. `sort` returns (a, b) if $a < b$, else it returns (b, a). $\mathsf{assert}\mathit{cond}$ checks condition *cond* and returns an exception when the condition is not true. When a call causes an exception, then we implicitly assume that the game/package also returns an exception. The adversary may catch exception.

We capture our assumptions and security notions as indistinguishability of a real game and an ideal game, i.e., the real game describes the real behaviour of a system whereas the ideal game describes the ideal behaviour of a system. As mentioned before, we slice the pseudocode which describes our games into several pieces of stateful code which can call one another but otherwise cannot access each others state. We call such a piece of stateful code a *package* P, we refer to the set of functions (oracles) that a package P exposes as the *output interface* $\mathsf{out}(\mathsf{P})$, and to the set of functions (oracles) that a package P calls as the *input interface* $\mathsf{in}(\mathsf{P})$. We refer to a function that is called or provided by a package as an *oracle* $\mathtt{O}$. As previously mentioned, one of the advantages of presenting modular pseudocode is that we can iteratively construct larger games. We start by defining games for basic primitives as compositions of packages, represented as a graph. Given these, we can then merge two graphs if they both contain one or more identical packages as nodes. These identical packages represent shared state between two primitives such as a key.

Package Parameters of Key

b_{Ind}:	Indistinguishability Bit
b_{uniq}:	Collision-resistance Bit

Package State

K:	**table** $[\mathsf{Index} \to \mathsf{Key}]$
Q:	**table** $[\mathsf{Index} \to \mathsf{Key}]$
H:	**table** $[\mathsf{Index} \to \mathsf{bool}]$

```
Set(idx, k, hon) → ()
if $Q[idx], H[idx] = k, hon$ :
  return ()
assert $Q[idx] = \bot$
$Q[idx] \leftarrow k; H[idx] \leftarrow hon$
if $b_{\mathsf{Ind}} \wedge H[idx] : K[idx] \leftarrow\$ \{0,1\}^{|k|}$
else : $K[idx] \leftarrow k$
if $b_{\mathsf{uniq}}$ :
  assert $\neg\exists h \neq h' : K[h] = K[h']$
return ()
```

```
Get(idx) → (Key, bool)
assert $Q[idx] \neq \bot$
return $K[idx], H[idx]$
```

Fig. 10: Key stores keys and honesty values in a table. `CSET` is short for `SET`$(.,.,0)$.

2.1 Key Packages

As (shared) keys play a central role in our work, we start by defining a $\mathsf{Key}^{b_{\mathsf{uniq}},b_{\mathsf{Ind}}}$ package which maintains a table K which stores keys and is indexed by a handle. The handle is a public value which the adversary can use to refer to the key. I.e., it can instruct a game to perform computations on the key that corresponds to the handle, e.g., perform a key derivation.

```
Package Parameters of $\mathsf{PKey}$
$G$:    Cyclic group of order $p$
$g$:    Generator

Package State
$K$:  table [PublicKey $\rightarrow$ SecretKey]
$H$:  table [PublicKey $\rightarrow$ bool]

$\mathtt{Get}(pk) \rightarrow (\mathsf{SecretKey}, \mathsf{bool})$
return $(K[pk], H[pk])$

$\mathtt{Set}(sk, hon) \rightarrow \mathsf{PublicKey}$
if $hon$ : $sk \leftarrow\$ \mathbb{Z}_p$
$pk \leftarrow g^{sk}$
$K[pk] \leftarrow sk$
$H[pk] \leftarrow hon$
return $pk$
```

Fig. 11: PKey stores secret keys in a table indexed by public keys.

The $\mathtt{SET}(h, k, hon)$ query allows to store key k under handle h, i.e., $K[h] \leftarrow k$. The boolean value hon allows to mark the key as *honest* ($hon = 1$) and *dishonest* ($hon = 0$). Honesty is stored in the honesty table H via $H[h] \leftarrow hon$ and is useful to treat honest and corrupt keys differently in the game. The $\mathtt{GET}(h)$ query allows the caller to obtain the key k which is stored under handle h, i.e., $k \leftarrow K[h]$, together with its honesty $hon \leftarrow H[h]$. If not specified otherwise, for simplicity, we assume that *keys* and output values of cryptographic primitives have the same length λ, throughout the paper.

In addition to their function as key tables, we use $\mathsf{Key}^{b_{\mathsf{uniq}},b_{\mathsf{Ind}}}$ packages to express security properties. This is useful when the key values k in the $\mathtt{SET}(h, k, hon)$ query are not available to the adversary since the $\mathtt{SET}(h, k, hon)$ query is made by a part of the security game such as the key derivation function. Then, we want to state that the adversary cannot distinguish whether the key derivation function writes real or uniformly random values into the key package. When the bit b_{Ind} equals 1, then $\mathtt{SET}(h, k, 1)$ stores a uniformly random value instead of a real value. Importantly, this should only happen for honest keys, namely those that were derived from honest keys unknown to the adversary.

Finally, we express uniqueness properties such as collision resistance with the help of a bit b_{uniq}. If $b_{\mathsf{uniq}} = 1$ then $\mathtt{SET}(h, k, hon)$ aborts whenever there already exists another handle $h \neq h'$ with the same key k. See Figure 10 for the code of $\mathsf{Key}^{b_{\mathsf{uniq}},b_{\mathsf{Ind}}}$. Note that the Q table is used to ensure that calling the package twice with the same inputs yields the same output.

We also define a public-key variant PKey which stores secret keys in a table that is indexed by the corresponding public keys. Similar to the symmetric-key case, $\texttt{SET}(h, sk, 0)$ allows to store a dishonest value sk. To model the generation of honest key pairs, $\texttt{SET}(h, sk, 1)$ instructs PKey to ignore sk, to generate a fresh honest key pair instead and to return the public-key to the adversary. See Figure 11 for the code of PKey. PKey is specialized to Diffie-Hellman keys and always generates honest keys uniformly at random. In our definitions and proofs, we only need this version of PKey and thus, unlike Key do not index PKey with a bit b.

3 Assumptions

In this section, we state the assumptions that our different key derivation primitives rely on. We state all assumptions as *multi-key* assumptions. Except for the Oracle Diffie-Hellman (ODH) assumption and collision-resistance (CR), the stated multi-key assumptions reduce to their single-key counterpart via slight variants of the Multi-Instance Lemma by BDFKK. Note that collision-resistance is naturally multi-key, while ODH could be defined with only a single challenge key pair and a reduction that loses a square function in the number of honest shares in the system. We omit this reduction.

Section 3.1 introduces computational extractors. Section 3.3 presents the Oracle Diffie-Hellman Assumption (ODH). Section 3.4 presents our assumption for pseudorandomness of a pseudorandom function (PRF). Section 3.4 presents our assumption of collision-resistance of a PRF. Finally, Section 3.2 states the combiner properties of XOR. Note that XOR is an *information-theoretically* secure combiner for randomness and thus not a computational assumption.

3.1 Computational Extractor

Let χ be a distribution over source key material which has high min-entropy. Given a sample k from distribution χ, an extractor $\texttt{xtr}$ takes an independent, uniformly random, publicly known salt value S and returns a value $y := \texttt{xtr}(S, k)$. An extractor $\texttt{xtr}$ is a good extractor for a distribution χ if the induced distribution over (y, S) is statistically close to the distribution over (z, S), where z is a strings of the same length as y, drawn uniformly at random and independently of S. In other words, $\texttt{xtr}(S, k)$

looks almost uniformly random from the perspective of an observer who knows S, but does not know k.

For a *computational* extractor, introduced by Krawczyk [Kra10], the distribution χ additionally returns some leakage *leak* to the observer, and the requirement on χ is that k has high computational min-entropy given *leak*. In this case, `xtr` is a good computational extractor for χ, if $\texttt{xtr}(S, k)$ is computationally indistinguishable from a uniformly random string of the same length, even when given *leak* and S. See Reyzin [Rey11] for a discussion of computational min-entropy.

```
Package Param.                 Package State
$\chi$:  Distribution          $ctr$:  Counter

Sample() → (Handle, Leakage)
$(k, leak) \leftarrow\!\$\ \chi$
Set$(h, k, ctr)$
$ctr \leftarrow ctr + 1$
return $(ctr, leak)$
```

Fig. 12: Sample pseudocode

We encode the computational security properties of an extractor `xtr` for distribution χ as a real-or-ideal security game GXTR^b (see Figure 13b). The Sample package (see Figure 12 for its code) models the high-entropy distribution and allows the adversary to generate a sample from χ which is then stored in the upper Key package under handle *ctr* which is a counter value maintained by Sample and returned to the adversary.

The `XTR` oracle of the XTR package allows the adversary to trigger the computation of $\texttt{xtr}(S, k)$ which is stored in the lower Key package and can be retrieved by the adversary via a `Get` query to the lower Key package. Additionally, the adversary can store its own values in the upper Key package (This does not change the strength of the security model, but will be convenient in bigger compositions when one is interested in the secure interactions between honest and adversarially chosen key values.). Pseudorandomness is captured by the bit b in the lower Key^{b0} package. If the bit is 0, concrete key values are stored. If the bit is 1, then for honest handles, uniformly random key value of the same length are stored.

Note that we sample the salt once and for all. However, since the honest samples are still drawn independently from the salt S, the reasoning about extractor security remains valid also when the *same* salt is used throughout.

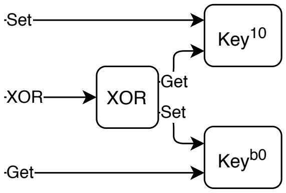
Definition 1 (Computational Extractor). *For a distribution χ, an extractor* `xtr` *and an adversary $\mathcal{A}$ we define the extractor advantage as*

$$\epsilon_{\mathsf{GXTR}}(\mathcal{A}) := \left|\Pr\left[1 = \mathcal{A} \to \mathsf{GXTR}^0\right] - \Pr\left[1 = \mathcal{A} \to \mathsf{GXTR}^1\right]\right|.$$

Package State of XOR

T: **table** [Context → Shard]

```
$\texttt{XOR}(ctx, \mathbf{shards}) \to ()$

$\mathbf{if}\ T[ctx] = \bot :$
    $T[ctx] \leftarrow \texttt{sort}(\mathbf{shards})$
$\mathbf{assert}\ T[ctx] = \texttt{sort}(\mathbf{shards})$
$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}((ctx, \mathbf{shards}))$
$k' \leftarrow \bigoplus \mathbf{k}$
$hon' \leftarrow \bigvee \mathbf{hon}$
$\texttt{Set}(ctx, k', hon')$
```



(a) Game GXOR^b.

Package Parameters of XTR

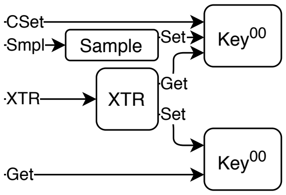
$\texttt{xtr}$: Computational Extractor
l: Length of salt value

Package State

S: salt

```
$\texttt{Init}() \to (\{0,1\}^l)$

$\mathbf{assert}\ S = \bot$
$S \leftarrow\$ \{0,1\}^l$
$\mathbf{return}\ S$
```

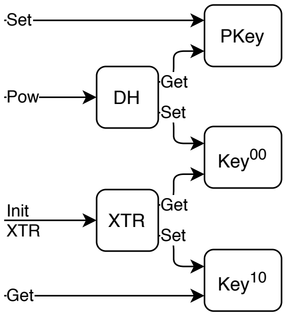
```
$\texttt{XTR}(h) \to ()$

$\mathbf{assert}\ S \neq \bot$
$k, hon \leftarrow \texttt{Get}(h)$
$k' \leftarrow \texttt{xtr}(S, k)$
$\texttt{Set}(h, k', hon)$
```



(b) Game GXTR^b.

Package Parameters of DH

G: Cyclic group of order p
g: Generator
$\texttt{xtr}$: Computational Extractor
l: Length of salt value



Package State

S: salt

```
$\texttt{Init}() \to (\{0,1\}^l)$

$\mathbf{assert}\ S = \bot$
$S \leftarrow\$ \{0,1\}^l$
$\mathbf{return}\ S$
```

```
$\texttt{Pow}(pkX, pkY) \to ()$

$\mathbf{assert}\ S \neq \bot$
$x, xHon \leftarrow \texttt{Get}(pkX)$
$y, yHon \leftarrow \texttt{Get}(pkY)$
$\mathbf{assert}\ x \neq \bot \vee y \neq \bot$
$\mathbf{if}\ x \neq \bot :$
    $k \leftarrow \texttt{xtr}(S, pkY^x)$
$\mathbf{else}\ :$
    $k \leftarrow \texttt{xtr}(S, pkX^Y)$
$\texttt{Set}(\texttt{sort}(pkX, pkY), k, xHon \wedge yHon)$
```

(c) Game GODH^b.

Fig. 13: Games for xor, extractors, and Oracle Diffie-Hellman.

3.2 XOR

Exclusive-OR (XOR) has the statistical property that the XOR of an arbitrary number of strings is distributed as the uniform distribution as long as one of the strings was drawn uniformly and independently at random. The game GXOR^b encodes this statement (See Figure 13a). The top Key^{10} package allows the adversary to trigger the sampling of random values and to store its own values. The `XOR` oracle of the XOR package allows the adversary to indicate a set of handles for keys which the XOR package retrieves from the upper Key^{10} package, then computes their XOR and stores the resulting value in the lower Key^{b0} package.

It is important that the XOR package ensures that each input key is only consumed once, as else, one could run into attacks where a one-time-pad is used twice. The XOR package ensures this by taking as input a set of partial handles **shard** and a context value *ctx*. For each $\mathsf{shards} \in$ **shards**, the XOR package retrieves the key corresponding for the handle $h(\mathit{ctx}, \mathit{shard})$, and the value *ctx* can never be used again, unless it is used exactly with the same **shards** (making the call useless). Thereby, the same value *ctx* can not be used for two calls to `XOR` and thus, handles in different calls have different *ctx* value and are distinct. See Figure 13a.

Definition 2 (XOR). *For an adversary $\mathcal{A}$ we define the* XOR *advantage as*

$$\epsilon_{\mathsf{GXOR}}(\mathcal{A}) := \left|\Pr\left[1 = \mathcal{A} \to \mathsf{GXOR}^0\right] - \Pr\left[1 = \mathcal{A} \to \mathsf{GXOR}^1\right]\right|.$$

Note that regardless of the computation time of $\mathcal{A}$, we have $\epsilon_{\mathsf{GXOR}}(\mathcal{A}) = 0$.

3.3 Oracle Diffie-Hellman (ODH)

The decisional Diffie-Hellman assumption (DDH) captures that given two honestly generated values $X = g^x$ and $Y = g^y$, the Diffie-Hellman secret g^{xy} is computationally indistinguishable from a uniformly random group element Z, even when given X and Y [Bon98].

The oracle Diffie-Hellman assumption (ODH) captures that the values extracted from an honest Diffie-Hellman secret via hashing are pseudorandom, even if the adversary is given an oracle where it can submit values Z and see extractions from, e.g., from Z^x [ABR01,BFGJ17]. Our encoding of the ODH assumption uses the extractor package XTR as a salted extractor. Note that for ODH, it is crucial that the salt is sampled once and for all, since else the adversary would be able to see extractions from the same DH secret with different salts, leading to the *salted* oracle Diffie-Hellman assumption [BDF$^+$] which is stronger and less well-understood. One can see DH as a special kind of distribution with leakage and as making an extractor assumption for this particular distribution. Note however, that the leakage is interactively computed, based on the adversary's inputs, making it a stronger assumption. See Figure 13c for the code of the ODH^b game. As before, security is encoded by the bit b in the lowest Key package which is b in the real game (stores real keys) and 1 in the ideal key. Note that in the case of DH, we need to compute the AND of the honesty values of the DH secrets (since knowing one of the two allows the adversary to derive the secret) whereas in all other games in this paper, we compute the OR of the honesty values of the keys we combine.

Definition 3 (ODH). *For an adversary $\mathcal{A}$ we define the* ODH *advantage as*

$$\epsilon_{\mathsf{GODH}}(\mathcal{A}) := \left|\Pr\left[1 = \mathcal{A} \to \mathsf{GODH}^0\right] - \Pr\left[1 = \mathcal{A} \to \mathsf{GODH}^1\right]\right|.$$

Package Parameters

$\mathtt{f}$:	Coll.-res. prf
ol:	Length of output
sl:	Length of salt value

Package State

T:	**table** [Cont. → Handle]
S:	$\{0,1\}^l$

```
CRInit() → ({0,1}^l)
assert $S = \bot$
$S \leftarrow_\$ \{0,1\}^{sl}$
return $S$
```

```
CREval($h$, $ctx1$, $ctx2$)
assert $S \neq \bot$
if $T[ctx1] = \bot$ :
    $T[ctx1] \leftarrow h$
assert $T[ctx1] = h$
$k, hon \leftarrow$ Get($h$)
$k' \leftarrow \mathtt{f}(k, (ctx1, ctx2), ol, S)$
Set(($ctx1$, $ctx2$), $k'$, $hon$)
```

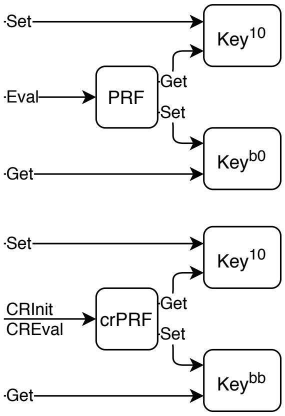


Fig. 14: Games GPRF^b and GcrPRF^b for the pseudorandom functions.

3.4 Pseudorandomness and Collision-Resistance of PRFs

The game GPRF^b encodes a multi-key version of the standard pseudorandomness assumption for pseudorandom functions. For convenience of proof, we actually make a slightly weaker assumption where we restrict the adversary to never use the same input *ctx* with two different keys. This allows us to use *ctx* as a handle for the output value of the PRF which is a technicality which will be useful later. In particular, it means that handles do not become very long.

See Figure 14 for the pseudo-code of GPRF^b. Analogously to previous games, the top Key^{10} package allows the adversary to store dishonest values and trigger the sampling of honest keys, the PRF package allows the adversary to trigger prf evaluations on inputs and keys of the adversary's choice, and the lower Key^b package allows the adversary to retrieve outputs which are concrete if $b = 0$ and uniformly random if $b = 1$ and if the key's handle is honest.

Definition 4 (PRF). *For an adversary $\mathcal{A}$ we define the* PRF *advantage as*

$$\epsilon_{\mathsf{GPRF}}(\mathcal{A}) := \left|\Pr\left[1 = \mathcal{A} \to \mathsf{GPRF}^0\right] - \Pr\left[1 = \mathcal{A} \to \mathsf{GPRF}^1\right]\right|.$$

Pseudorandomness and Collision-resistance The GcrPRF^b game encodes the security of a pseudorandom function which is, additionally, collision-resistance. E.g., HKDF is designed to be both collision-resistant and a pseudorandom function, see Krawczyk [Kra10,Kra18]. In Figure 14, the code lines in bright purple are code lines which are part of GcrPRF^b, but not part of GPRF^b. Throughout this paper, bright purple denotes code relating to collision-resistance. Pseudorandomness is encoded as before by the first bit of the lower Key^{bb} package. Recall from Section 2.1 that if the bit in the second position is 1, then this means that an error message is returned when twice the same value is stored. Indistinguishability between plain storage in Key^{00} and unique storage in Key^{11} encodes collision-resistance, since the adversary can distinguish whenever it can cause a collision.

Recall that in in GPRF^b, we restricted the adversary to use unique contexts merely for convenience. In turn, for GcrPRF^b, restricting the adversary to unique contexts is crucial, since the adversary can register the same dishonest key values under two different handles in the upper Key^{10} package, and if the adversary were allows to use the same context with both of them, the adversary would cause a trivial collision (and thus a trivial win) on the output key.

Salting In concrete security, there always *exists* an adversary against the collision-resistance of a function—Rogaway thus duped our standard collision-resistance notion *human ignorance collision-resistance*, since it relies on the collision not being known. Using a salt in the hash-function makes collision-resistance statements meaningful again, and

Definition 5 (crPRF). *For an adversary $\mathcal{A}$ the* crPRF *advantage is defined as*

$$\epsilon_{\mathsf{GcrPRF}}(\mathcal{A}) := \left|\Pr\left[1 = \mathcal{A} \to \mathsf{GcrPRF}^0\right] - \Pr\left[1 = \mathcal{A} \to \mathsf{GcrPRF}^1\right]\right|.$$

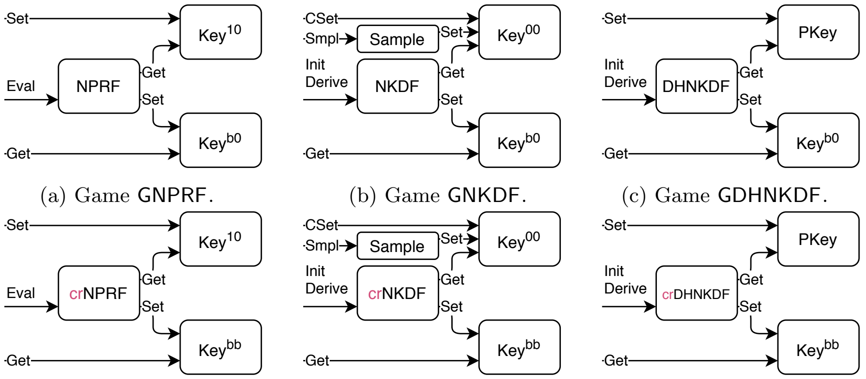
4 Security Notions

Figure 15 describes our security notions. The security properties provided by the six primitives are encoded in the lower Key package. Namely, in the real games of all six games, the bits of the lower Key^{00} package are 00. In turn, in the ideal games in Figure 15 (a)-(c), the bits of the lower Key^{10} package are 10, whereas in the ideal games in Figure 15 (d)-(e), the bits of the lower Key^{11} package are 11. Recall from Section 2.1 that Key^{00} stores real keys while Key^{10} and Key^{11} sample keys at random when the handle is honest. Thereby, for all six games, the indistinguishability between the real and the ideal variant of the games encodes pseudorandomness. In addition, Key^{11} sends a special abort message to the adversary when two key values collide. Therefore, for the games in Figure 15 (d)-(e), the indistinguishability of the real and ideal game additionally encodes collision-resistance of *all* derived keys, including dishonest ones. The table T in all security games ensures that the *context* values used for derivation are unique. The variable S in all games stores the salt which is sampled once and for all.

To summarize, all six primitives combine an arbitrary number of keys into a pseudorandom key if at least one input key is honest and if they are called with unique context values. In each column in Figure 15, the main difference between the top and the bottom game is the second bit of the lower Key package so that the lower game also encodes collision-resistance. The difference between each of the columns is how the original key material is generated from which the final key is derived. We explain each column in turn.

In the GNPRF^{b0} and $\mathsf{GcrNPRF}^{bb}$ games in Figure 15 (a) and Figure 15 (d), the top Key^{10} package samples honest keys uniformly at random and keeps them secret from the adversary. It also allows the adversary to register its own dishonest keys, possibly even multiple times.

The GNKDF and $\mathsf{GcrNKDF}$ games in Figure 15 (b) and Figure 15 (e) allow the same registration capabilities to the adversary for dishonest



(a) Game GNPRF.

(b) Game GNKDF.

(c) Game GDHNKDF.

(d) Game GcrNPRF.

(e) Game GcrNKDF.

(f) Game GcrDHNKDF.

Package Parameters	
f:	cr-NPRF
sl:	Salt length
ol:	Output key length

```
Init()
assert $S = \bot$
$S \leftarrow\!\!\$\ \{0,1\}^{sl}$
return $S$

Eval($\mathbf{h}$, $ctx1$, $ctx2$)
if $T[ctx1] = \mathbf{h}$ :
    return
assert $T[ctx1] = \bot$
$T[ctx1] \leftarrow \mathbf{h}$
assert $S \neq \bot$
$\mathbf{k}, \mathbf{hon} \overset{vec}{\leftarrow}$ Get($\mathbf{h}$)
$k' \leftarrow \mathtt{f}(\mathbf{k}, ctx1, ctx2, S)$
$hon' \leftarrow \bigvee \mathbf{hon}$
Set($(ctx1, ctx2), k', hon'$)
```

(g) NPRF package code.

Package Parameters	
f:	cr-NKDF
l:	Salt length

```
Init()
assert $S_{Ext} = \bot$
$S_{Ext} \leftarrow\!\!\$\ \{0,1\}^{l}$
$S_{cr} \leftarrow\!\!\$\ \{0,1\}^{l}$
return $S_{Ext}, S_{cr}$

Derive($\mathbf{h}$, $ctx1$, $ctx2$)
if $T[ctx] = \mathbf{h}$ :
    return
assert $T[ctx] = \bot$
$T[ctx] \leftarrow \mathbf{h}$
assert $S_{Ext} \neq \bot, S_{cr} \neq \bot$
$\mathbf{k}, \mathbf{hon} \overset{vec}{\leftarrow}$ Get($\mathbf{h}$)
$k' \leftarrow \mathtt{f}(\mathbf{k}, ctx1, S_{Ext}, ctx2, S_{cr})$
$hon' \leftarrow \bigvee \mathbf{hon}$
Set($(ctx1, ctx2), k', hon'$)
```

(h) NKDF package code.

Package Parameters	
f:	cr-DHNKDF
l:	Salt length

```
Init()
assert $S_{Ext} = \bot$
$S_{Ext} \leftarrow\!\!\$\ \{0,1\}^{l}$
$S_{cr} \leftarrow\!\!\$\ \{0,1\}^{l}$
return $S_{Ext}, S_{cr}$

Pow($\mathbf{pkX}, \mathbf{pkY}$, $ctx1$, $ctx2$)
if $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$ :
    return
assert $T[ctx1] = \bot$
$T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $S_{Ext} \neq \bot, S_{cr} \neq \bot$
$\mathbf{skX}, \mathbf{honX} \overset{vec}{\leftarrow}$ Get($\mathbf{pkX}$)
$\mathbf{skY}, \mathbf{honY} \overset{vec}{\leftarrow}$ Get($\mathbf{pkY}$)
$\mathbf{hon} \overset{vec}{\leftarrow} \mathbf{honX} \wedge \mathbf{honY}$
$\mathbf{sk}, \mathbf{pk} \leftarrow$ pick($\mathbf{skX}, \mathbf{pkX}$,
    $\mathbf{skY}, \mathbf{pkY}$)
$k' \leftarrow \mathtt{f}(\mathbf{sk}, \mathbf{pk}, ctx1, S_{Ext}, ctx2, S_{cr})$
$hon' \leftarrow \bigvee \mathbf{hon}$
Set($(ctx1, ctx2), k', hon'$)
```

(i) DHNKDF package code.

Fig. 15: Security Notions. Observe that (c) and (f) use a PKey package. (d)-(f) use the same idealization bit in the lower Key package for pseudorandomness and uniqueness.

keys. The honest input key material is sampled in package Sample from a high-entropy distribution and then stored in the Key^{00} package.

Finally, in the the GDHNKDF and GcrDHNKDF games in Figure 15 (c) and Figure 15 (f), the adversary may register dishonest Diffie-Hellman secrets in Pkey and trigger the sampling of honest Diffie-Hellman shares in Pkey.

5 Constructions

We have three constructions, each of which relies on different key material: The NPRF relies on the honest keys being uniformly random, the NKDF relies on key material with high computational min-entropy, and the DHNKDF relies on key material which stems from DH keys. We turn to each of these constructions shortly. Before, we want to discuss how to turn a version of the construction which does not provide unique keys into a variant which does provide unique keys. Namely, it suffices to take the output of a NPRF, NKDF or DHNKDF and run it through a crPRF, with unique context: The pseudorandomness will be preserved, and additionally, the output will become unique. As we will see later, we have an analogy between the construction and the reduction proof. As a feature of our design, the original proof for each of the three primitives can be augmented by a single additional proof step to add the collision-resistance property. Since the plain and the collision-resistance primitive are so similar, we will treat them both at the same time, adding a bright purple cr to highlight the part that is only relevant for the collision-resistant version of the statement.

```
$\mathtt{f}_{\mathtt{crNPRF}}(\mathbf{k}, ctx1, ctx2, S_{cr})$
for $i \in \{0, ..., |\mathbf{k}| - 1\}$ :
    $\mathbf{k}_i \leftarrow \mathtt{f}_{\mathtt{PRF}}(\mathbf{k}_i, (ctx1, i))$
$k' \leftarrow \bigoplus \mathbf{k}$
$k' \leftarrow \mathtt{f}_{\mathtt{crPRF}}(k', S_{cr}, (ctx1, ctx2))$
return $k'$
```

```
$\mathtt{f}_{\mathtt{crNKDF}}(\mathbf{k}, ctx1, S_{Ext}, ctx2, S_{cr})$
$\mathbf{k} \overset{\text{vec}}{\leftarrow} \mathtt{xtr}(S_{Ext}, \mathbf{k})$
for $i \in \{0, ..., |\mathbf{k}| - 1\}$ :
    $\mathbf{k}_i \leftarrow \mathtt{f}_{\mathtt{PRF}}(\mathbf{k}_i, (ctx1, i))$
$k' \leftarrow \bigoplus \mathbf{k}$
$k' \leftarrow \mathtt{f}_{\mathtt{crPRF}}(k', S_{cr}, (ctx1, ctx2))$
return $k'$
```

```
$\mathtt{f}_{\mathtt{crDHNKDF}}(\mathbf{pk}, \mathbf{sk}, ctx1, S_{Ext}, ctx2, S_{cr})$
assert $|\mathbf{pk}| = |\mathbf{sk}|$
$\mathbf{k} \overset{\text{vec}}{\leftarrow} \mathbf{pk}^{\mathbf{sk}}$
$\mathbf{k} \overset{\text{vec}}{\leftarrow} \mathtt{xtr}(S_{Ext}, \mathbf{k})$
for $i \in \{0, ..., |\mathbf{pk}| - 1\}$ :
    $\mathbf{k}_i \leftarrow \mathtt{f}_{\mathtt{PRF}}(\mathbf{k}_i, (ctx1, i))$
$k' \leftarrow \bigoplus \mathbf{k}$
$k' \leftarrow \mathtt{f}_{\mathtt{crPRF}}(k', S_{cr}, (ctx1, ctx2))$
return $k'$
```

Fig. 16: Pseudocode of the constructions $\mathtt{f}_{\mathtt{crNPRF}}$, $\mathtt{f}_{\mathtt{crNKDF}}$ and $\mathtt{f}_{\mathtt{crDHNKDF}}$.

5.1 NPRF and crNPRF

Figure 16 provides the code of our crNPRF construction $\mathtt{f}_{\mathsf{crNPRF}}$. It takes a list of keys $\mathbf{k}$ and a context values *ctx* and a salt S_{cr} as input. Then, for the i-th entry in $\mathbf{k}$, denoted, $\mathbf{k}_i$, $\mathtt{f}_{\mathsf{crNPRF}}$ evaluates a pseudorandom function $\mathtt{f}_{\mathsf{PRF}}$, keyed with $\mathbf{k}_i$, on the input (ctx, i) to obtain an output which then overwrites the $\mathbf{k}_i$ in the state of the function evaluation. After performing this operation for all entries i, the resulting $\mathbf{k}_i$ values are then xored into k', then a collision-resistant PRF $\mathtt{f}_{\mathtt{crPRF}}$ is evaluated on k', S_{cr} and *ctx* and the result is assigned back to k' and then k' is returned.

Theorem 1 (NPRF). *For all adversaries $\mathcal{A}$, it holds that*

$$\begin{aligned}
& \epsilon_{\mathsf{NPRF}}(\mathcal{A}) \\
= \; & \epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{A} \to \mathsf{MOD\text{-}NPRF}) && (1) \\
\leq \; & \epsilon_{\mathsf{PRF}}(\mathcal{A} \to \mathsf{MOD\text{-}NPRF} \to \mathsf{R}^1{}_{\mathsf{NPRF}}) \\
+ \; & \epsilon_{\mathsf{PRF}}(\mathcal{A} \to \mathsf{MOD\text{-}NPRF} \to \mathsf{R}^3{}_{\mathsf{NPRF}}) && (2)
\end{aligned}$$

where the reductions $\mathsf{R}^1{}_{\mathsf{NPRF}}$ and $\mathsf{R}^3{}_{\mathsf{NPRF}}$ are specified in Figures 20b and 20d, respectively.

Theorem 2 (crNPRF). *For all adversaries $\mathcal{A}$, it holds that*

$$\begin{aligned}
& \epsilon_{\mathsf{crNPRF}}(\mathcal{A}) \\
= \; & \epsilon_{\mathsf{CORE\text{-}crNPRF}}(\mathcal{A} \to \mathsf{MOD\text{-}crNPRF}) && (3) \\
\leq \; & \epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{A} \to \mathsf{MOD\text{-}crNPRF} \to \mathsf{R}^1{}_{\mathsf{crNPRF}}) \\
+ \; & \epsilon_{\mathsf{crPRF}}(\mathcal{A} \to \mathsf{MOD\text{-}crNPRF} \to \mathsf{R}^2{}_{\mathsf{crNPRF}}) \\
+ \; & \epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{A} \to \mathsf{MOD\text{-}crNPRF} \to \mathsf{R}^3{}_{\mathsf{crNPRF}}) && (4)
\end{aligned}$$

where the reductions $\mathsf{R}^1{}_{\mathsf{NPRF}}$, $\mathsf{R}^3{}_{\mathsf{NPRF}}$, $\mathsf{R}^1{}_{\mathsf{crNPRF}}$, $\mathsf{R}^2{}_{\mathsf{crNPRF}}$ and $\mathsf{R}^3{}_{\mathsf{crNPRF}}$ are specified in Figures 20b, 20d, 21b, 21c and 21d, respectively.

5.2 NKDF and crNKDF

Figure 16 provides the code of our crNKDF construction $\mathtt{f}_{\mathsf{crNKDF}}$. It is analogous to the crNPRF construction, except that it extracts the initial key material from a high-entropy source rather than assuming that it is sampled uniformly at random.

Theorem 3 (NKDF)**.** *For all adversaries $\mathcal{A}$, it holds that*

$$\begin{aligned} & \epsilon_{\mathsf{NKDF}}(\mathcal{A}) \\ = \; & \epsilon_{\mathsf{CORE\text{-}NKDF}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}NKDF}) \qquad (5) \\ \leq \; & \epsilon_{\mathsf{XTR}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}NKDF} \rightarrow \mathsf{R}^1{}_{\mathsf{NKDF}}) \\ + \; & \epsilon_{\mathsf{NPRF}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}NKDF} \rightarrow \mathsf{R}^2{}_{\mathsf{NKDF}}) \\ + \; & \epsilon_{\mathsf{XTR}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}NKDF} \rightarrow \mathsf{R}^3{}_{\mathsf{NKDF}}) \qquad (6) \end{aligned}$$

and reductions $\mathsf{R}^1{}_{\mathsf{NKDF}}$ *and* $\mathsf{R}^3{}_{\mathsf{NKDF}}$ *are specified in Figures 20f and 20h, respectively.*

Theorem 4 (crNKDF)**.** *For all adversaries $\mathcal{A}$, it holds that*

$$\begin{aligned} & \epsilon_{\mathsf{crNKDF}}(\mathcal{A}) \\ = \; & \epsilon_{\mathsf{CORE\text{-}crNKDF}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}crNKDF}) \qquad (7) \\ \leq \; & \epsilon_{\mathsf{CORE\text{-}NKDF}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}crNKDF} \rightarrow \mathsf{R}^1{}_{\mathsf{crNKDF}}) \\ + \; & \epsilon_{\mathsf{crPRF}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}crNKDF} \rightarrow \mathsf{R}^2{}_{\mathsf{crNKDF}}) \\ + \; & \epsilon_{\mathsf{CORE\text{-}NKDF}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}crNKDF} \rightarrow \mathsf{R}^3{}_{\mathsf{crNKDF}}) \qquad (8) \end{aligned}$$

where the reductions $\mathsf{R}^1{}_{\mathsf{NKDF}}$, $\mathsf{R}^3{}_{\mathsf{NKDF}}$, $\mathsf{R}^1{}_{\mathsf{crNKDF}}$, $\mathsf{R}^2{}_{\mathsf{crNKDF}}$ *and* $\mathsf{R}^3{}_{\mathsf{crNKDF}}$ *are specified in Figures 20f, 20h, 21f, 21g and 21h, respectively.*

5.3 DHNKDF and crDHNKDF

Figure 16 provides the code of our crDHNKDF construction $\mathtt{f}_{\mathsf{crDHNKDF}}$. Again, the difference to the crNPRF construction resides in the initial key material. In the crDHNKDF construction, the initial key material is extracted from a Diffie-Hellman secret.

Theorem 5 (DHNKDF)**.** *For all adversaries $\mathcal{A}$, it holds that*

$$\begin{aligned} & \epsilon_{\mathsf{DHNKDF}}(\mathcal{A}) \\ = \; & \epsilon_{\mathsf{CORE\text{-}DHNKDF}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}DHNKDF}) \qquad (9) \\ \leq \; & \epsilon_{\mathsf{ODH}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}DHNKDF} \rightarrow \mathsf{R}^1{}_{\mathsf{DHNKDF}}) \\ + \; & \epsilon_{\mathsf{NPRF}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}DHNKDF} \rightarrow \mathsf{R}^2{}_{\mathsf{DHNKDF}}) \\ + \; & \epsilon_{\mathsf{ODH}}(\mathcal{A} \rightarrow \mathsf{MOD\text{-}DHNKDF} \rightarrow \mathsf{R}^3{}_{\mathsf{DHNKDF}}) \qquad (10) \end{aligned}$$

where the reductions $\mathsf{R}^1{}_{\mathsf{DHNKDF}}$ *and* $\mathsf{R}^3{}_{\mathsf{DHNKDF}}$ *are specified in Figures 20j and 20l, respectively.*

Theorem 6 ($\mathsf{crDHNKDF}$)**.** *For all adversaries $\mathcal{A}$, it holds that*

$$\begin{aligned}
& \epsilon_{\mathsf{crDHNKDF}}(\mathcal{A}) \\
= \quad & \epsilon_{\mathsf{CORE\text{-}crDHNKDF}}(\mathcal{A} \to \mathsf{MOD\text{-}crDHNKDF}) && (11) \\
\leq \quad & \epsilon_{\mathsf{CORE\text{-}DHNKDF}}(\mathcal{A} \to \mathsf{MOD\text{-}crDHNKDF} \to \mathsf{R^1}_{\mathsf{crDHNKDF}}) \\
+ \quad & \epsilon_{\mathsf{crPRF}}(\mathcal{A} \to \mathsf{MOD\text{-}crDHNKDF} \to \mathsf{R^2}_{\mathsf{crDHNKDF}}) \\
+ \quad & \epsilon_{\mathsf{CORE\text{-}DHNKDF}}(\mathcal{A} \to \mathsf{MOD\text{-}crDHNKDF} \to \mathsf{R^3}_{\mathsf{crDHNKDF}}) && (12)
\end{aligned}$$

where the reductions $\mathsf{R^1}_{\mathsf{DHNKDF}}$*,* $\mathsf{R^3}_{\mathsf{DHNKDF}}$*,* $\mathsf{R^1}_{\mathsf{crDHNKDF}}$*,* $\mathsf{R^2}_{\mathsf{crDHNKDF}}$ *and* $\mathsf{R^3}_{\mathsf{crDHNKDF}}$ *are specified in Figures 20j, 20l, 21j, 21k and 21l, respectively.*

6 Proofs

We now prove Theorem 1-5. Section 6.1 describes the modularization of the high-level security notions into modular games. The proofs proceed via inlining and Bellare-Rogaway style code-based game-playing [BR06]. While such proofs might be tedious, we deem them crucial for the case of constructing primitives based on uniqueness criteria, since we would catch missing preconditions in such a proof.

Once modularized, in Section 6.2, we can proceed via visual, graph-based reduction arguments which will be striking in their simplicity and clarity. I.e., the modularization in Section 6.1 is the hard part, and the remaining part of the proof then becomes easy.

To shorten the proofs additionally, our modular design allows us to re-use parts of previous proofs. e.g., for each of the collision-resistant primitives, we will re-use lemmas for their non-collision-resistant counterpart. Towards this goal, it is useful to define the following advantages:

$$\begin{aligned}
\epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{A}) :=& \,|\Pr\left[1 = \mathcal{A} \to \mathsf{CORE\text{-}NPRF}^0\right] \\
&- \Pr\left[1 = \mathcal{A} \to \mathsf{CORE\text{-}NPRF}^3\right]| \\
\epsilon_{\mathsf{CORE\text{-}crNPRF}}(\mathcal{A}) :=& \,|\Pr\left[1 = \mathcal{A} \to \mathsf{CORE\text{-}crNPRF}^0\right] \\
&- \Pr\left[1 = \mathcal{A} \to \mathsf{CORE\text{-}crNPRF}^3\right]| \\
\epsilon_{\mathsf{CORE\text{-}NKDF}}(\mathcal{A}) :=& \,|\Pr\left[1 = \mathcal{A} \to \mathsf{CORE\text{-}NKDF}^0\right] \\
&- \Pr\left[1 = \mathcal{A} \to \mathsf{CORE\text{-}NKDF}^3\right]| \\
\epsilon_{\mathsf{CORE\text{-}crNKDF}}(\mathcal{A}) :=& \,|\Pr\left[1 = \mathcal{A} \to \mathsf{CORE\text{-}crNKDF}^0\right] \\
&- \Pr\left[1 = \mathcal{A} \to \mathsf{CORE\text{-}crNKDF}^3\right]|
\end{aligned}$$

$$\begin{aligned}\epsilon_{\textsf{CORE-DHNKDF}}(\mathcal{A}) := |\Pr\left[1 = \mathcal{A} \to \textsf{CORE-DHNKDF}^0\right] \\ - \Pr\left[1 = \mathcal{A} \to \textsf{CORE-DHNKDF}^3\right]| \\ \epsilon_{\textsf{CORE-crDHNKDF}}(\mathcal{A}) := |\Pr\left[1 = \mathcal{A} \to \textsf{CORE-crDHNKDF}^0\right] \\ - \Pr\left[1 = \mathcal{A} \to \textsf{CORE-crDHNKDF}^3\right]|,\end{aligned}$$

where the respective games are defined in Figure 20 and Figure 21.

6.1 Modularization proofs via Inlining

We first decompose all six monolithic games into functionally equivalent modular games. Since the monolithic games and the modular games have the same input-output behaviour, an adversary's advantage does not change. We obtain the following six claims, structured into 3 claims, each of which states the decomposition soundness for the collision-resistant and the non-collision-resistant variant of the respective primitive. The respective modularizations are depicted in Figure 17 and Figure 18.

```
Package State
$T$:   table

Init()
return CRInit()

CREval($\mathbf{h}$, $ctx1$, $ctx2$)
if $T[ctx1] = \bot$ :
   $T[ctx1] \leftarrow \mathbf{h}$
assert $T[ctx1] = \mathbf{h}$
for $i \in \{0, ..., |\mathbf{h}| - 1\}$ :
   Eval($\mathbf{h}_i, (ctx1, i)$)
XOR($ctx1, (0, ..., |\mathbf{h}| - 1)$)
CREval($ctx1, ctx1, ctx2$)
```

(a) Code of MOD-crNPRF

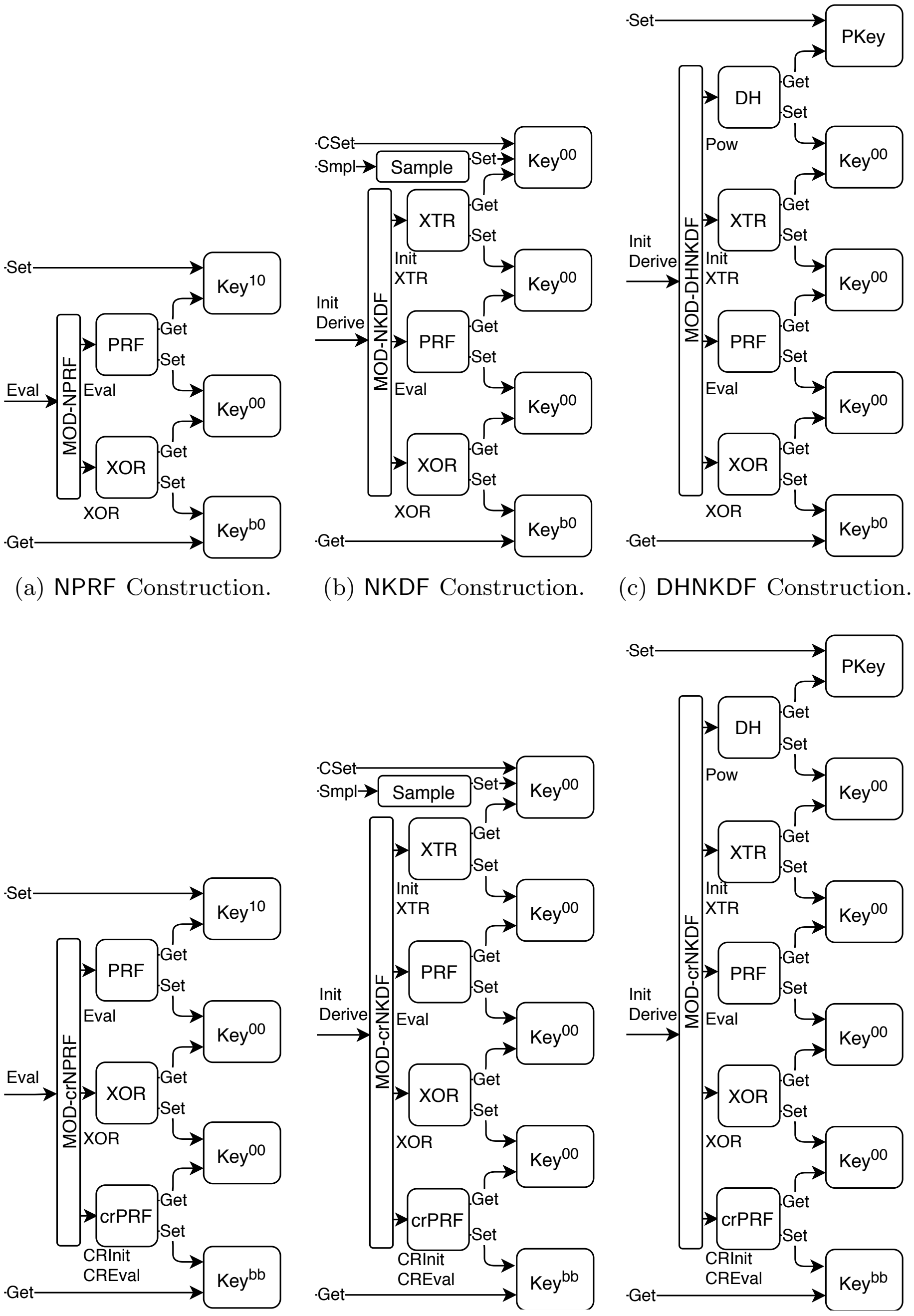
```
Package State
$T$:   table

Init()
return Init(), CRInit()

CRDerive($\mathbf{h}$, $ctx1$, $ctx2$)
if $T[ctx1] = \bot$ :
   $T[ctx1] \leftarrow \mathbf{h}$
assert $T[ctx1] = \mathbf{h}$
for $i \in \{0, ..., |\mathbf{h}| - 1\}$ :
   XTR($\mathbf{h}_i$)
   Eval($\mathbf{h}_i, (ctx1, i)$)
XOR($ctx1, (0, ..., |\mathbf{h}| - 1)$)
CREval($ctx1, ctx1, ctx2$)
```

(b) Code of MOD-crNKDF

```
Package State
$T$:   table

Init()
return Init(), CRInit()

CRDerive($\mathbf{pkX}, \mathbf{pkY}$, $ctx1$, $ctx2$)
assert $|\mathbf{pkX}| = |\mathbf{pkY}|$
$(\mathbf{pkX}, \mathbf{pkY}) \overset{\text{vec}}{\leftarrow}$ sort($\mathbf{pkX}, \mathbf{pkY}$)
if $T[ctx1] = \bot$ :
   $T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$
for $i \in \{0, ..., |\mathbf{pkX}| - 1\}$ :
   Pow($\mathbf{pkX}_i, \mathbf{pkY}_i$)
   XTR($(\mathbf{pkX}_i, \mathbf{pkY}_i)$)
   Eval($(\mathbf{pkX}_i, \mathbf{pkY}_i), (ctx1, i)$)
XOR($ctx1, (0, ..., |\mathbf{h}| - 1)$)
CREval($ctx1, ctx1, ctx2$)
```

(c) Code of MOD-crDHNKDF

Fig. 17: Pseudocode of Modular Constructions in Figure 16.



(a) NPRF Construction.

(b) NKDF Construction.

(c) DHNKDF Construction.

(d) crNPRF Construction.

(e) crNKDF Construction.

(f) crDHNKDF Constr.

Fig. 18: Modular Constructions for the primitives in Figure 15.

Claim 1 (GcrNPRF **Decomposition**) *It holds that*

$$\mathsf{GNPRF}^0 \overset{code}{=} \mathsf{MOD\text{-}NPRF} \to \mathsf{CORE}_{\mathsf{GNPRF}}{}^0$$
$$\mathsf{GNPRF}^1 \overset{code}{=} \mathsf{MOD\text{-}NPRF} \to \mathsf{CORE}_{\mathsf{GNPRF}}{}^3$$
$$\mathsf{GcrNPRF}^0 \overset{code}{=} \mathsf{MOD\text{-}crNPRF} \to \mathsf{CORE}_{\mathsf{GcrNPRF}}{}^0$$
$$\mathsf{GcrNPRF}^1 \overset{code}{=} \mathsf{MOD\text{-}crNPRF} \to \mathsf{CORE}_{\mathsf{GcrNPRF}}{}^3$$

where $\overset{code}{=}$ *denotes that the code of the two packages on the left and right of the equivalence have the same input-output behavior.* MOD-NPRF *and* MOD-crNPRF *are defined in Figure 17a,* $\mathsf{CORE}_{\mathsf{GNPRF}}{}^0$ *is defined in Figure 20a,* $\mathsf{CORE}_{\mathsf{GNPRF}}{}^3$ *is defined in Figure 20d,* $\mathsf{CORE}_{\mathsf{crGNPRF}}{}^0$ *is defined in Figure 21a, and* $\mathsf{CORE}_{\mathsf{crGNPRF}}{}^3$ *is defined in Figure 21d.*

Before turning to the proof of Claim 1, we observe that Claim 1 directly implies Equality 1 and Equality 3 in Theorem 1, because if two pairs of games have the same input-output behavior, then the adversary's advantage in both games is identical.

Proof. The proof of Claim 1 proceeds via inlining $\mathsf{CORE}_{\mathsf{GcrNPRF}}{}^0$ into MOD-crNKDF and comparing with $\mathsf{GcrNPRF}^0$ and, analogously, inlining $\mathsf{CORE}_{\mathsf{GcrNPRF}}{}^3$ into MOD-crNKDF and comparing with $\mathsf{GcrNPRF}^1$. Since $\mathsf{CORE}_{\mathsf{GcrNPRF}}{}^0$ and $\mathsf{CORE}_{\mathsf{GcrNPRF}}{}^3$ only differ w.r.t. their last bit in the lowest Key^{b0} package, we can simply choose to inline all packages except for that last one and obtain the same code. Moreover, also $\mathsf{GcrNPRF}^0$ and $\mathsf{GcrNPRF}^1$ only differ w.r.t. the lowest Key^{b0} package and thus, in the inlining, we can focus on the main packages.

Consider Figure 19. In the left-most column, there is the code of `CREval` of MOD-crNKDF. In the second column, we inline the `EVAL` (defined in package PRF), `XOR` (defined in package XOR) and `CREval` (defined in package `CRPRF`) oracles. Compare with their descriptions in Figure 13 and Figure 14. From the second to third column, we remove redundant checks—these checks are redundant since the first three lines of the oracle code ensure that each value *ctx* is only used once. Once the checks are removed, the assignments to the table can be removed, too. From the third to fourth column, we pull the `Get` call in front of the for-loop and use vector assignment notation instead. From the fourth to fifth column, instead of writing a value into a Key^{00} package and reading it again, we simply assign the value directly to the respective variable. Occasionally, we refer to this tool as a *Set-then-Get* rule. This step is valid whenever the handle in the call to the Key^{00} package is only used once. This is the case here as the first three lines of the oracle code ensure that each value *ctx* is only used once. Additionally, from the fourth to fifth column, we move certain computations up or down in the code which is a valid step as the variables are neither read nor written to in the meanwhile. We then obtain the code of $\mathtt{f}_{\mathtt{crNPRF}}$ and thus recovered the code of $\mathsf{CORE}_{\mathsf{GcrNPRF}}{}^{0}$ in column 6. □

Claim 2 (GcrNKDF **Decomposition)** *It holds that*

$$\mathsf{GNKDF}^{0} \overset{code}{=} \mathsf{MOD\text{-}NKDF} \to \mathsf{CORE}_{\mathsf{GNKDF}}{}^{0}$$
$$\mathsf{GNKDF}^{1} \overset{code}{=} \mathsf{MOD\text{-}NKDF} \to \mathsf{CORE}_{\mathsf{GNKDF}}{}^{3}$$
$$\mathsf{GcrNKDF}^{0} \overset{code}{=} \mathsf{MOD\text{-}crNKDF} \to \mathsf{CORE}_{\mathsf{GcrNKDF}}{}^{0}$$
$$\mathsf{GcrNKDF}^{1} \overset{code}{=} \mathsf{MOD\text{-}crNKDF} \to \mathsf{CORE}_{\mathsf{GcrNKDF}}{}^{3}$$

where MOD-NKDF *and the packages* MOD-crNKDF *are defined in Figure 17b,* $\mathsf{CORE}_{\mathsf{GNKDF}}{}^{0}$ *is defined in Figure 20e,* $\mathsf{CORE}_{\mathsf{GNKDF}}{}^{3}$ *is defined in Figure 20h,* $\mathsf{CORE}_{\mathsf{crGNKDF}}{}^{0}$ *is defined in Figure 21e, and* $\mathsf{CORE}_{\mathsf{crGNKDF}}{}^{3}$ *is defined in Figure 21h.*

Claim 2 directly implies Equality 5 and Equality 7 in Theorem 3.

Proof. Analogously to the proof of Claim 1, the proof proceeds by inlining, see Figure 22 for the step-by-step inlining and the accompanying explanations. As before, the proof starts with a straightforward inlining of the oracles `Eval`, `XOR`, `CREval` defined by the packages PRF, XOR, CRPRF in Figure 13 and Figure 14. Then, redundant checks and assignments are removed based on the *ctx* check in the beginning of the oracle call. Then, vector notation is applied, code lines are moved up and down respectively, and the dropping of *Set-then-Get* is applied until we obtain the code of $\mathtt{f}_{\mathtt{crNKDF}}$, as desired. □

Claim 3 (GcrDHNKDF **Decomposition)** *It holds that*

$$\mathsf{GDHNKDF}^0 \overset{code}{=} \mathsf{MOD\text{-}DHNKDF} \to {\mathsf{CORE}_{\mathsf{GDHNKDF}}}^0$$
$$\mathsf{GDHNKDF}^1 \overset{code}{=} \mathsf{MOD\text{-}DHNKDF} \to {\mathsf{CORE}_{\mathsf{GDHNKDF}}}^3$$
$$\mathsf{GcrDHNKDF}^0 \overset{code}{=} \mathsf{MOD\text{-}crDHNKDF} \to {\mathsf{CORE}_{\mathsf{GcrDHNKDF}}}^0$$
$$\mathsf{GcrDHNKDF}^1 \overset{code}{=} \mathsf{MOD\text{-}crDHNKDF} \to {\mathsf{CORE}_{\mathsf{GcrDHNKDF}}}^3$$

where $\mathsf{MOD\text{-}DHNKDF}$ *is defined in Figure 17c,* ${\mathsf{CORE}_{\mathsf{GDHNKDF}}}^0$ *and the packages* $\mathsf{MOD\text{-}crDHNKDF}$ *are defined in Figure 20i,* ${\mathsf{CORE}_{\mathsf{GDHNKDF}}}^3$ *is defined in Figure 20l,* ${\mathsf{CORE}_{\mathsf{crGDHNKDF}}}^0$ *is defined in Figure 21i, and* ${\mathsf{CORE}_{\mathsf{crGDHNKDF}}}^3$ *is defined in Figure 21l.*

Claim 3 directly implies Equality 9 and Equality 11 in Theorem 5.

Proof. The proof proceeds via inlining, see Figure 23 and is analogous to the previous two claims. One difference is the computation of the honesty function which is slightly more complex than in the previous two cases, since we first compute an *and* over the honesty of the DH shares and then an *or* over the results of that computation. □

6.2 Modular Core Proofs

We now proceed to the modular proofs of the adversary's advantages against the core games. We prove each of the six core lemmas individually.

Lemma 1 ($\mathsf{CORE}_{\mathsf{GNPRF}}$)**.** *For all adversaries* $\mathcal{B}$

$$\begin{aligned}
&\epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{B})\\
\leq&\epsilon_{\mathsf{PRF}}(\mathcal{B} \to {\mathsf{R}^1}_{\mathsf{NPRF}}) + \epsilon_{\mathsf{XOR}}(\mathcal{B} \to {\mathsf{R}^2}_{\mathsf{NPRF}}) + \epsilon_{\mathsf{PRF}}(\mathcal{B} \to {\mathsf{R}^3}_{\mathsf{NPRF}})\\
=&\epsilon_{\mathsf{PRF}}(\mathcal{B} \to {\mathsf{R}^1}_{\mathsf{NPRF}}) + 0 + \epsilon_{\mathsf{PRF}}(\mathcal{B} \to {\mathsf{R}^3}_{\mathsf{NPRF}}),
\end{aligned}$$

where ${\mathsf{R}^1}_{\mathsf{NPRF}}$ *is defined in Figure 20b,* ${\mathsf{R}^2}_{\mathsf{XOR}}$ *is defined in Figure 20c, and* ${\mathsf{R}^3}_{\mathsf{NPRF}}$ *is defined in Figure 20d.*

Lemma 1 directly implies Inequality 1 in Theorem 1, by choosing $\mathcal{B} = \mathcal{A} \to \mathsf{MOD\text{-}NPRF}$.

Proof. Recall that

$$\epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{B}) \overset{\text{def}}{=} \left|\Pr\left[1 = \mathcal{B} \to {\mathsf{CORE}^0}_{\mathsf{NPRF}}\right] - \Pr\left[1 = \mathcal{B} \to {\mathsf{CORE}^3}_{\mathsf{NPRF}}\right]\right|.$$

```
$\texttt{Init}() \to (\{0,1\}^{l})$
return $\texttt{CRInit}()$    (replaced by code)
                              assert $S = \bot$
                              $S \leftarrow\!\!\$\ \{0,1\}^{sl}$
                              return $S$
```

$\texttt{CREval}(\mathbf{h}, \mathit{ctx1}, \mathit{ctx2}) \to ()$

if $T[\mathit{ctx1}] = \bot$:	**if** $T[\mathit{ctx1}] = \bot$:	**if** $T[\mathit{ctx1}] = \bot$:	**if** $T[\mathit{ctx1}] = \bot$:	**if** $T[\mathit{ctx1}] = \bot$:	**if** $T[\mathit{ctx1}] = \bot$:
$T[\mathit{ctx1}] \leftarrow \mathbf{h}$	$T[\mathit{ctx1}] \leftarrow \mathbf{h}$	$T[\mathit{ctx1}] \leftarrow \mathbf{h}$	$T[\mathit{ctx1}] \leftarrow \mathbf{h}$	$T[\mathit{ctx1}] \leftarrow \mathbf{h}$	$T[\mathit{ctx1}] \leftarrow \mathbf{h}$
assert $T[\mathit{ctx1}] = \mathbf{h}$	**assert** $T[\mathit{ctx1}] = \mathbf{h}$	**assert** $T[\mathit{ctx1}] = \mathbf{h}$	**assert** $T[\mathit{ctx1}] = \mathbf{h}$	**assert** $T[\mathit{ctx1}] = \mathbf{h}$	**assert** $T[\mathit{ctx1}] = \mathbf{h}$
				assert $S \neq \bot$	**assert** $S \neq \bot$
			$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}(\mathbf{h})$	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}(\mathbf{h})$	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}(\mathbf{h})$
					$k' \leftarrow \texttt{f}_{\texttt{crNPRF}}(\mathbf{k}, \mathit{ctx1}, ol, S)$
for $i \in \{0, \dots, \lvert\mathbf{h}\rvert - 1\}$:	**for** $i \in \{0, \dots, \lvert\mathbf{h}\rvert - 1\}$:	**for** $i \in \{0, \dots, \lvert\mathbf{h}\rvert - 1\}$:	**for** $i \in \{0, \dots, \lvert\mathbf{h}\rvert - 1\}$:	**for** $i \in \{0, \dots, \lvert\mathbf{h}\rvert - 1\}$:	*(replaced with function* $\texttt{f}_{\texttt{crNPRF}}$*)*
$\texttt{Eval}(\mathbf{h}_i, (\mathit{ctx1}, i))$	*(replaced by code)*				
	if $T_{PRF}[(\mathit{ctx1}, i)] = \bot$:	*(redundant check, removed)*			
	$T_{PRF}[(\mathit{ctx1}, i)] \leftarrow \mathbf{h}_i$	*(redundant assignment, removed)*			
	assert $T_{PRF}[(\mathit{ctx1}, i)] = \mathbf{h}_i$	*(redundant check, removed)*			
	$k, hon \leftarrow \texttt{Get}(\mathbf{h}_i)$	$k, hon \leftarrow \texttt{Get}(\mathbf{h}_i)$	*(using vector assignment notation)*		
	$k' \leftarrow \texttt{f}_{\texttt{PRF}}(k, (\mathit{ctx1}, i), ol)$	$k' \leftarrow \texttt{f}_{\texttt{PRF}}(k, (\mathit{ctx1}, i), ol)$	$k' \leftarrow \texttt{f}_{\texttt{PRF}}(\mathbf{k}_i, (\mathit{ctx1}, i), ol)$	$\mathbf{k}_i \leftarrow \texttt{f}_{\texttt{PRF}}(\mathbf{k}_i, (\mathit{ctx1}, i), ol)$	*(replaced with function* $\texttt{f}_{\texttt{crNPRF}}$*)*
	$\texttt{Set}((\mathit{ctx1}, i), k', hon)$	$\texttt{Set}((\mathit{ctx1}, i), k', hon)$	$\texttt{Set}((\mathit{ctx1}, i), k', \mathbf{hon}_i)$	*(drop Set-then-Get)*	
$\texttt{XOR}(\mathit{ctx1}, (0, \dots, \lvert\mathbf{h}\rvert - 1))$	*(replaced by code)*				
	if $T_{XOR}[\mathit{ctx1}] = \bot$:	*(redundant check, removed)*			
	$T_{XOR}[\mathit{ctx1}] \leftarrow \texttt{sort}((0, \dots, \lvert\mathbf{h}\rvert - 1))$	*(redundant assignment, removed)*			
	assert $T_{XOR}[\mathit{ctx1}] = \texttt{sort}((0, \dots, \lvert\mathbf{h}\rvert - 1))$	*(redundant check, removed)*			
	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}((\mathit{ctx1}, (0, \dots, \lvert\mathbf{h}\rvert - 1)))$	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}((\mathit{ctx1}, (0, \dots, \lvert\mathbf{h}\rvert - 1)))$	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}((\mathit{ctx1}, (0, \dots, \lvert\mathbf{h}\rvert - 1)))$	*(drop Set-then-Get)*	
	$k' \leftarrow \bigoplus \mathbf{k}$	$k' \leftarrow \bigoplus \mathbf{k}$	$k' \leftarrow \bigoplus \mathbf{k}$	$k' \leftarrow \bigoplus \mathbf{k}$	*(replaced with function* $\texttt{f}_{\texttt{crNPRF}}$*)*
	$hon' \leftarrow \bigvee \mathbf{hon}$	$hon' \leftarrow \bigvee \mathbf{hon}$	$hon' \leftarrow \bigvee \mathbf{hon}$	*(moved down)*	
	$\texttt{Set}(\mathit{ctx1}, k', hon')$	$\texttt{Set}(\mathit{ctx1}, k', hon')$	$\texttt{Set}(\mathit{ctx1}, k', hon')$	*(drop Set-then-Get)*	
$\texttt{CREval}(ctx1, ctx1, ctx2)$	*(replaced by code)*				
	assert $S \neq \bot$	**assert** $S \neq \bot$	**assert** $S \neq \bot$	*(moved up)*	
	if $T_{cr}[\mathit{ctx1}] = \bot$:	*(redundant check, removed)*			
	$T_{cr}[\mathit{ctx1}] \leftarrow \mathit{ctx1}$	*(redundant assignment, removed)*			
	assert $T_{cr}[\mathit{ctx1}] = \mathit{ctx1}$	*(redundant check, removed)*			
	$k, hon \leftarrow \texttt{Get}(\mathit{ctx1})$	$k, hon \leftarrow \texttt{Get}(\mathit{ctx1})$	$k, hon \leftarrow \texttt{Get}(\mathit{ctx1})$	*(drop Set-then-Get)*	
	$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (\mathit{ctx1}, \mathit{ctx2}), S)$	$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (\mathit{ctx1}, \mathit{ctx2}), S)$	$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (\mathit{ctx1}, \mathit{ctx2}), S)$	$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k', \mathit{ctx1}, S)$	*(replaced with function* $\texttt{f}_{\texttt{crNPRF}}$*)*
				$hon' \leftarrow \bigvee \mathbf{hon}$	$hon' \leftarrow \bigvee \mathbf{hon}$
	$\texttt{Set}((\mathit{ctx1}, \mathit{ctx2}), k', hon)$	$\texttt{Set}((\mathit{ctx1}, \mathit{ctx2}), k', hon)$	$\texttt{Set}((\mathit{ctx1}, \mathit{ctx2}), k', hon)$	$\texttt{Set}((\mathit{ctx1}, \mathit{ctx2}),k', hon)$	$\texttt{Set}((\mathit{ctx1}, \mathit{ctx2}), k', hon)$

Fig. 19: Inlining NPRF

We can thus prove Lemma 1 by bounding the difference between these two games, i.e., starting with $\Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^0{}_{\mathsf{NPRF}}\right]$ and then proceeding via several ϵ-transformations to $\Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^0{}_{\mathsf{NPRF}}\right]$. In the following transformations, equalities follow from graph equality by visual inspection, and inequalities follow from the positivity of absolute value.

$$\begin{aligned}
&\Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^0{}_{\mathsf{NPRF}}\right] \\
\overset{\text{Fig. 20b}}{=}&\Pr\left[1 = \mathcal{B} \to \mathsf{R}^1{}_{\mathsf{NPRF}} \to \mathsf{GPRF}^0\right] \\
&-\Pr\left[1 = \mathcal{B} \to \mathsf{R}^1{}_{\mathsf{NPRF}} \to \mathsf{GPRF}^1\right] \\
&+\Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^1{}_{\mathsf{NPRF}}\right] \\
\overset{\text{Fig. 20c}}{=}&\Pr\left[1 = \mathcal{B} \to \mathsf{R}^2{}_{\mathsf{NPRF}} \to \mathsf{GXOR}^0\right] \\
&-\Pr\left[1 = \mathcal{B} \to \mathsf{R}^2{}_{\mathsf{NPRF}} \to \mathsf{GXOR}^1\right] \\
&+\Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^2{}_{\mathsf{NPRF}}\right] + \epsilon_{\mathsf{PRF}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{NPRF}}) \\
\overset{\text{Fig. 20d}}{=}&\Pr\left[1 = \mathcal{B} \to \mathsf{R}^3{}_{\mathsf{NPRF}} \to \mathsf{GPRF}^0\right] \\
&-\Pr\left[1 = \mathcal{B} \to \mathsf{R}^3{}_{\mathsf{NPRF}} \to \mathsf{GPRF}^1\right] \\
&+\Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^3{}_{\mathsf{NPRF}}\right] + \epsilon_{\mathsf{PRF}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{NPRF}}) + 0 \\
=\;&\Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^3{}_{\mathsf{NPRF}}\right] + \epsilon_{\mathsf{PRF}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{NPRF}}) \\
&+\epsilon_{\mathsf{PRF}}(\mathcal{B} \to \mathsf{R}^3{}_{\mathsf{NPRF}})
\end{aligned}$$

Lemma 2 ($\mathsf{CORE}_{\mathsf{GcrNPRF}}$)**.** *For all adversaries $\mathcal{B}$*

$$\begin{aligned}
&\epsilon_{\mathsf{CORE\text{-}crNPRF}}(\mathcal{B}) \\
\le&\epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{crNPRF}}) \\
&+ \epsilon_{\mathsf{crPRF}}(\mathcal{B} \to \mathsf{R}^2{}_{\mathsf{crNPRF}}) \\
&+ \epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{B} \to \mathsf{R}^3{}_{\mathsf{crNPRF}})
\end{aligned}$$

where $\mathsf{R}^1{}_{\mathsf{crNPRF}}$ is defined in Figure 21b, $\mathsf{R}^2{}_{\mathsf{crNPRF}}$ is defined in Figure 21c, and $\mathsf{R}^3{}_{\mathsf{crNPRF}}$ is defined in Figure 21d.

Lemma 2 directly implies Inequality 4 in Theorem 2, by choosing $\mathcal{B} = \mathcal{A} \to \mathsf{MOD\text{-}crNPRF}$.

Proof. Recall that

$$\epsilon_{\mathsf{CORE\text{-}crNPRF}}(\mathcal{B}) \overset{\text{def}}{=} \left|\Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^0{}_{\mathsf{crNPRF}}\right] - \Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^3{}_{\mathsf{crNPRF}}\right]\right|.$$

Analogously to the proof of Lemma 1, we start with the probability $\Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^0{}_{\mathsf{crNPRF}}\right]$ and then proceed via several ϵ-transformations to $\Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^0{}_{\mathsf{crNPRF}}\right]$.

$$\begin{aligned}
& \Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^0{}_{\mathsf{crNPRF}}\right] \\
\overset{\text{Fig. 21b}}{=} & \Pr\left[1 = \mathcal{B} \to \mathsf{R}^1{}_{\mathsf{crNPRF}} \to \mathsf{CORE}^0{}_{\mathsf{crNPRF}}\right] \\
& - \Pr\left[1 = \mathcal{B} \to \mathsf{R}^1{}_{\mathsf{crNPRF}} \to \mathsf{CORE}^1{}_{\mathsf{crNPRF}}\right] \\
& + \Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^1{}_{\mathsf{crNPRF}}\right] \\
\overset{\text{Fig. 21c}}{\leq} & \Pr\left[1 = \mathcal{B} \to \mathsf{R}^2{}_{\mathsf{crNPRF}} \to \mathsf{GcrPRF}^0\right] \\
& - \Pr\left[1 = \mathcal{B} \to \mathsf{R}^2{}_{\mathsf{crNPRF}} \to \mathsf{GcrPRF}^1\right] \\
& + \Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^2{}_{\mathsf{crNPRF}}\right] + \epsilon_{\mathsf{CORE\text{-}crNPRF}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{crNPRF}}) \\
\overset{\text{Fig. 21d}}{\leq} & \Pr\left[1 = \mathcal{B} \to \mathsf{R}^3{}_{\mathsf{crNPRF}} \to \mathsf{GPRF}^0\right] \\
& - \Pr\left[1 = \mathcal{B} \to \mathsf{R}^3{}_{\mathsf{crNPRF}} \to \mathsf{GPRF}^1\right] \\
& + \Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^3{}_{\mathsf{crNPRF}}\right] + \epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{crNPRF}}) \\
& + \epsilon_{\mathsf{crPRF}}(\mathcal{B} \to \mathsf{R}^2{}_{\mathsf{crNPRF}}) \\
\leq & \Pr\left[1 = \mathcal{B} \to \mathsf{CORE}^3{}_{\mathsf{crNPRF}}\right] + \epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{crNPRF}}) \\
& + \epsilon_{\mathsf{crPRF}}(\mathcal{B} \to \mathsf{R}^2{}_{\mathsf{crNPRF}}) + \epsilon_{\mathsf{CORE\text{-}NPRF}}(\mathcal{B} \to \mathsf{R}^3{}_{\mathsf{crNPRF}})
\end{aligned}$$

Lemma 3 **($\mathsf{CORE}_{\mathsf{GNKDF}}$).** *For all adversaries $\mathcal{B}$*

$$\begin{aligned}
& \epsilon_{\mathsf{CORE\text{-}NKDF}}(\mathcal{B}) \\
\leq & \epsilon_{\mathsf{XTR}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{NKDF}}) \\
& + \epsilon_{\mathsf{CORE\text{-}NKDF}}(\mathcal{B} \to \mathsf{R}^2{}_{\mathsf{NKDF}}) \\
& + \epsilon_{\mathsf{XTR}}(\mathcal{B} \to \mathsf{R}^3{}_{\mathsf{NKDF}})
\end{aligned}$$

where $\mathsf{R}^1{}_{\mathsf{NKDF}}$ is defined in Figure 20f, $\mathsf{R}^2{}_{\mathsf{NKDF}}$ is defined in Figure 20g, and $\mathsf{R}^3{}_{\mathsf{NKDF}}$ is defined in Figure 20h.

Proof. Based on Figures 20e-20h, the proof is analogous to the proof of Lemma 1.

Lemma 4 ($\mathsf{CORE}_{\mathsf{GcrNKDF}}$)**.** *For all adversaries $\mathcal{B}$*

$$\begin{aligned}&\epsilon_{\mathsf{CORE\text{-}crNKDF}}(\mathcal{B})\\ \leq&\epsilon_{\mathsf{CORE\text{-}NKDF}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{crNKDF}})\\ &+ \epsilon_{\mathsf{crPRF}}(\mathcal{B} \to \mathsf{R}^2{}_{\mathsf{crNKDF}})\\ &+ \epsilon_{\mathsf{CORE\text{-}NKDF}}(\mathcal{B} \to \mathsf{R}^3{}_{\mathsf{crNKDF}})\end{aligned}$$

where $\mathsf{R}^1{}_{\mathsf{crNKDF}}$ is defined in Figure 21f, $\mathsf{R}^2{}_{\mathsf{crNKDF}}$ is defined in Figure 21g, and $\mathsf{R}^3{}_{\mathsf{crNKDF}}$ is defined in Figure 21h.

Proof. Based on Figure 21e-21h, the proof is analogous to the proof of Lemma 2.

Lemma 5 ($\mathsf{CORE}_{\mathsf{GDHNKDF}}$)**.** *For all adversaries $\mathcal{B}$*

$$\begin{aligned}&\epsilon_{\mathsf{CORE\text{-}DHNKDF}}(\mathcal{B})\\ \leq&\epsilon_{\mathsf{ODH}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{DHNKDF}})\\ &+ \epsilon_{\mathsf{CORE\text{-}NKDF}}(\mathcal{B} \to \mathsf{R}^2{}_{\mathsf{DHNKDF}})\\ &+ \epsilon_{\mathsf{ODH}}(\mathcal{B} \to \mathsf{R}^3{}_{\mathsf{DHNKDF}})\end{aligned}$$

where $\mathsf{R}^1{}_{\mathsf{DHNKDF}}$ is defined in Figure 20j, $\mathsf{R}^2{}_{\mathsf{DHNKDF}}$ is defined in Figure 20k, and $\mathsf{R}^3{}_{\mathsf{DHNKDF}}$ is defined in Figure 20l.

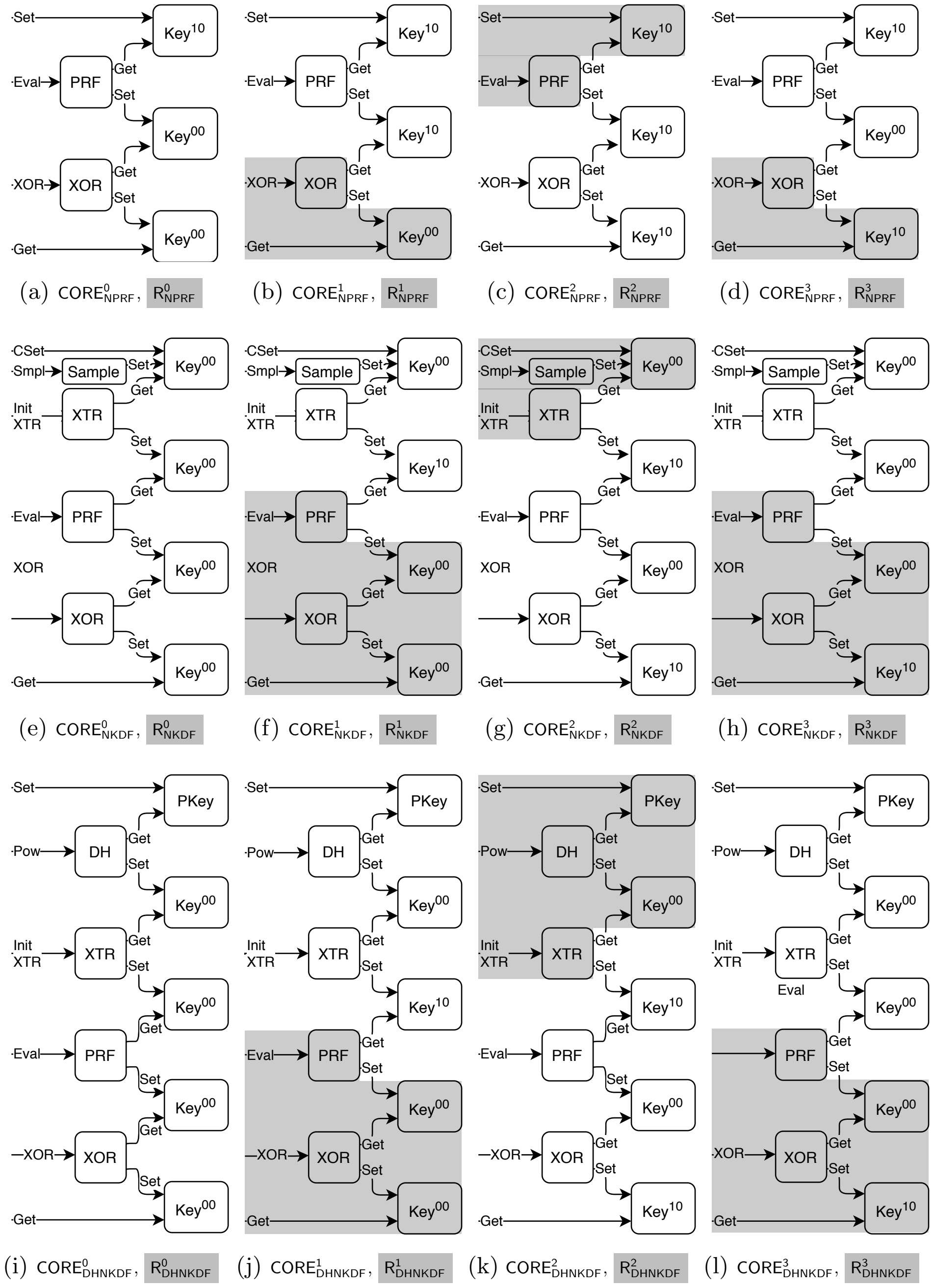
Proof. Based on Figure 20i-20l, the proof is analogous to the proof of Lemma 1.

Lemma 6 ($\mathsf{CORE}_{\mathsf{GcrDHNKDF}}$)**.** *For all adversaries $\mathcal{B}$*

$$\begin{aligned}&\epsilon_{\mathsf{CORE\text{-}crDHNKDF}}(\mathcal{B})\\ \leq&\epsilon_{\mathsf{CORE\text{-}DHNKDF}}(\mathcal{B} \to \mathsf{R}^1{}_{\mathsf{crDHNKDF}})\\ &+ \epsilon_{\mathsf{crPRF}}(\mathcal{B} \to \mathsf{R}^2{}_{\mathsf{crDHNKDF}})\\ &+ \epsilon_{\mathsf{CORE\text{-}DHNKDF}}(\mathcal{B} \to \mathsf{R}^3{}_{\mathsf{crDHNKDF}})\end{aligned}$$

where $\mathsf{R}^1{}_{\mathsf{crDHNKDF}}$ is defined in Figure 21j, $\mathsf{R}^2{}_{\mathsf{crDHNKDF}}$ is defined in Figure 21k, and $\mathsf{R}^3{}_{\mathsf{crDHNKDF}}$ is defined in Figure 21l.

Proof. Based on Figure 21i-21l, the proof is analogous to the proof of Lemma 2.

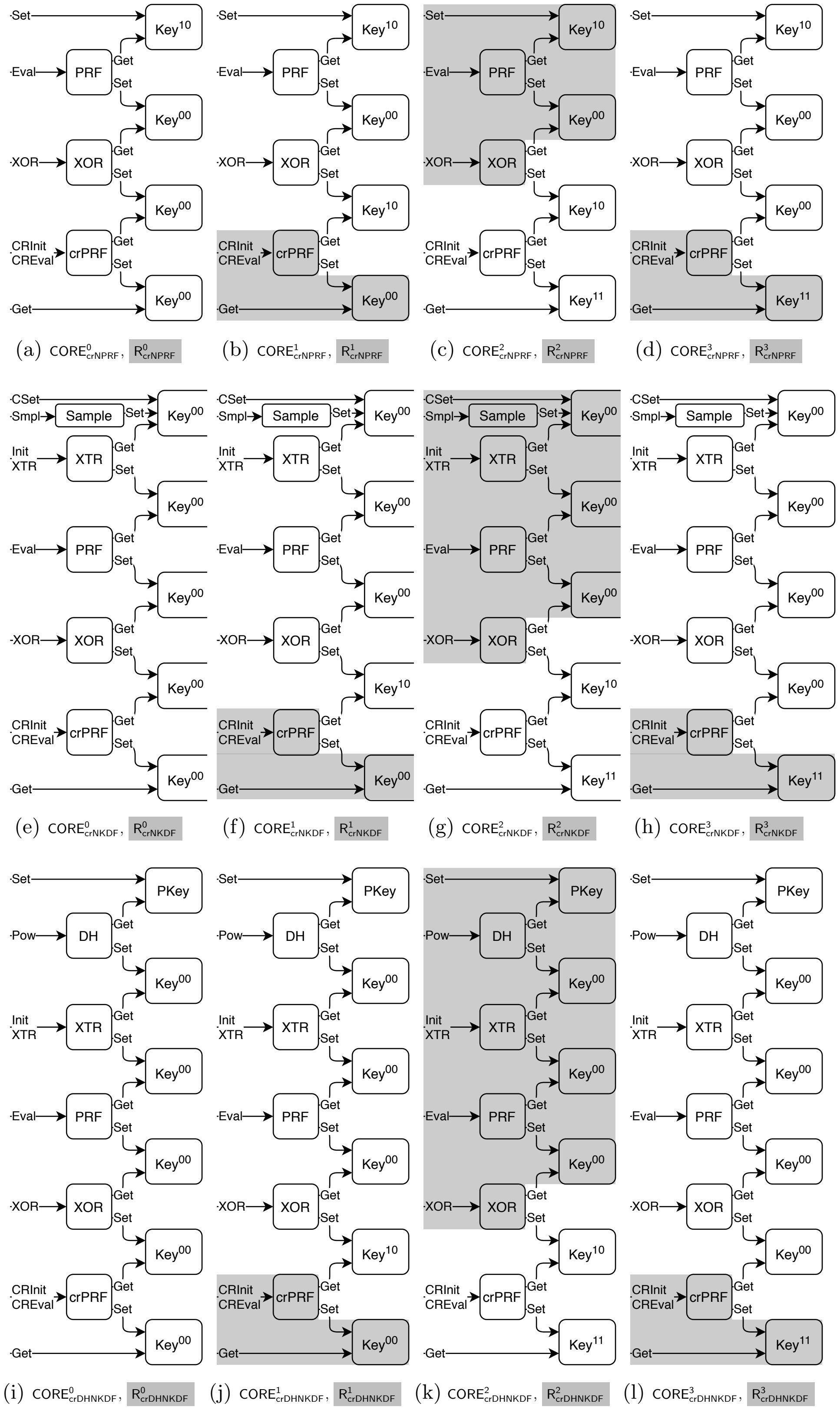


(a) $\mathsf{CORE}^0_{\mathsf{NPRF}}$, $\mathsf{R}^0_{\mathsf{NPRF}}$ (b) $\mathsf{CORE}^1_{\mathsf{NPRF}}$, $\mathsf{R}^1_{\mathsf{NPRF}}$ (c) $\mathsf{CORE}^2_{\mathsf{NPRF}}$, $\mathsf{R}^2_{\mathsf{NPRF}}$ (d) $\mathsf{CORE}^3_{\mathsf{NPRF}}$, $\mathsf{R}^3_{\mathsf{NPRF}}$

(e) $\mathsf{CORE}^0_{\mathsf{NKDF}}$, $\mathsf{R}^0_{\mathsf{NKDF}}$ (f) $\mathsf{CORE}^1_{\mathsf{NKDF}}$, $\mathsf{R}^1_{\mathsf{NKDF}}$ (g) $\mathsf{CORE}^2_{\mathsf{NKDF}}$, $\mathsf{R}^2_{\mathsf{NKDF}}$ (h) $\mathsf{CORE}^3_{\mathsf{NKDF}}$, $\mathsf{R}^3_{\mathsf{NKDF}}$

(i) $\mathsf{CORE}^0_{\mathsf{DHNKDF}}$, $\mathsf{R}^0_{\mathsf{DHNKDF}}$ (j) $\mathsf{CORE}^1_{\mathsf{DHNKDF}}$, $\mathsf{R}^1_{\mathsf{DHNKDF}}$ (k) $\mathsf{CORE}^2_{\mathsf{DHNKDF}}$, $\mathsf{R}^2_{\mathsf{DHNKDF}}$ (l) $\mathsf{CORE}^3_{\mathsf{DHNKDF}}$, $\mathsf{R}^3_{\mathsf{DHNKDF}}$

Fig. 20: Proof steps for the modular constructions in Figure 18.



(a) $\mathsf{CORE}^0_{\mathsf{crNPRF}}$, $\mathsf{R}^0_{\mathsf{crNPRF}}$ (b) $\mathsf{CORE}^1_{\mathsf{crNPRF}}$, $\mathsf{R}^1_{\mathsf{crNPRF}}$ (c) $\mathsf{CORE}^2_{\mathsf{crNPRF}}$, $\mathsf{R}^2_{\mathsf{crNPRF}}$ (d) $\mathsf{CORE}^3_{\mathsf{crNPRF}}$, $\mathsf{R}^3_{\mathsf{crNPRF}}$

(e) $\mathsf{CORE}^0_{\mathsf{crNKDF}}$, $\mathsf{R}^0_{\mathsf{crNKDF}}$ (f) $\mathsf{CORE}^1_{\mathsf{crNKDF}}$, $\mathsf{R}^1_{\mathsf{crNKDF}}$ (g) $\mathsf{CORE}^2_{\mathsf{crNKDF}}$, $\mathsf{R}^2_{\mathsf{crNKDF}}$ (h) $\mathsf{CORE}^3_{\mathsf{crNKDF}}$, $\mathsf{R}^3_{\mathsf{crNKDF}}$

(i) $\mathsf{CORE}^0_{\mathsf{crDHNKDF}}$, $\mathsf{R}^0_{\mathsf{crDHNKDF}}$ (j) $\mathsf{CORE}^1_{\mathsf{crDHNKDF}}$, $\mathsf{R}^1_{\mathsf{crDHNKDF}}$ (k) $\mathsf{CORE}^2_{\mathsf{crDHNKDF}}$, $\mathsf{R}^2_{\mathsf{crDHNKDF}}$ (l) $\mathsf{CORE}^3_{\mathsf{crDHNKDF}}$, $\mathsf{R}^3_{\mathsf{crDHNKDF}}$

Fig. 21: Proof steps for the modular constructions in Figure 18.

References

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 143–158. Springer, Heidelberg, April 2001.

[BDF$^+$] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. Handshake security for the TLS 1.3 standard. In progress.

[BDF$^+$18] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 222–249. Springer, Heidelberg, December 2018.

[BDK$^+$11] Boaz Barak, Yevgeniy Dodis, Hugo Krawczyk, Olivier Pereira, Krzysztof Pietrzak, François-Xavier Standaert, and Yu Yu. Leftover hash lemma, revisited. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 1–20. Springer, Heidelberg, August 2011.

[BFGJ17] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. PRF-ODH: Relations, instantiations, and impossibility results. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 651–681. Springer, Heidelberg, August 2017.

[BL15] Mihir Bellare and Anna Lysyanskaya. Symmetric and dual PRFs from standard assumptions: A generic validation of an HMAC assumption. Cryptology ePrint Archive, Report 2015/1198, 2015. http://eprint.iacr.org/2015/1198.

[Bon98] Dan Boneh. The decision Diffie-Hellman problem. In *Third Algorithmic Number Theory Symposium (ANTS)*, volume 1423 of *LNCS*. Springer, Heidelberg, 1998. Invited paper.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006.

[CCD$^+$16] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013, 2016. http://eprint.iacr.org/2016/1013.

[DRS20] Benjamin Dowling, Paul Rösler, and Jörg Schwenk. Flexible authenticated and confidential channel establishment (fACCE): Analyzing the noise protocol framework. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part I*, volume 12110 of *LNCS*, pages 341–373. Springer, Heidelberg, May 2020.

[JKSS10] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. Generic compilers for authenticated key exchange. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 232–249. Springer, Heidelberg, December 2010.

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Heidelberg, August 2010.

[Kra18] Hugo Krawczyk. Bar-ilan winter school. *IACR Cryptology ePrint Archive*, 2018.

[KW15] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and TLS 1.3. Cryptology ePrint Archive, Report 2015/978, 2015. http://eprint.iacr.org/2015/978.

[Res18] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.

[Rey11] Leonid Reyzin. Some notions of entropy for cryptography - (invited talk). In Serge Fehr, editor, *ICITS 11*, volume 6673 of *LNCS*, pages 138–142. Springer, Heidelberg, May 2011.

```
$\texttt{Init}() \rightarrow (\{0,1\}^l)$
return $\texttt{Init}.\texttt{CRInit}()$   (replaced by code)
                           assert $S_{Ext} = \bot$
                           $S_{Ext} \leftarrow_\$ \{0,1\}^{sl}$
                           assert $S_{cr} = \bot$
                           $S_{cr} \leftarrow_\$ \{0,1\}^{sl}$
                           return $S_{Ext}, S_{cr}$
```

$\texttt{CRDerive}(\mathbf{h}, ctx1, ctx2) \rightarrow ()$

if $T[ctx1] = \bot$:	**if** $T[ctx1] = \bot$:	**if** $T[ctx1] = \bot$:	**if** $T[ctx1] = \bot$:	**if** $T[ctx1] = \bot$:	**if** $T[ctx1] = \bot$:
$\quad T[ctx1] \leftarrow \mathbf{h}$	$\quad T[ctx1] \leftarrow \mathbf{h}$	$\quad T[ctx1] \leftarrow \mathbf{h}$	$\quad T[ctx1] \leftarrow \mathbf{h}$	$\quad T[ctx1] \leftarrow \mathbf{h}$	$\quad T[ctx1] \leftarrow \mathbf{h}$
assert $T[ctx1] = \mathbf{h}$	**assert** $T[ctx1] = \mathbf{h}$	**assert** $T[ctx1] = \mathbf{h}$	**assert** $T[ctx1] = \mathbf{h}$	**assert** $T[ctx1] = \mathbf{h}$	**assert** $T[ctx1] = \mathbf{h}$
				assert $S_{Ext} \neq \bot, S_{cr} \neq \bot$	**assert** $S_{Ext} \neq \bot, S_{cr} \neq \bot$
		$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}(\mathbf{h})$	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}(\mathbf{h})$	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}(\mathbf{h})$	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}(\mathbf{h})$
					$k' \leftarrow \texttt{f}_{\texttt{crNKDF}}(\mathbf{k}, ctx1, ol, S_{Ext}, ctx2, S_{cr})$
			$\mathbf{k} \overset{\text{vec}}{\leftarrow} \texttt{xtr}(S_{Ext}, \mathbf{k})$	$\mathbf{k} \overset{\text{vec}}{\leftarrow} \texttt{xtr}(S_{Ext}, \mathbf{k})$	*(replaced with function* $\texttt{f}_{\texttt{crNKDF}}$*)*
for $i \in \{0, ..., \lvert\mathbf{h}\rvert - 1\}$:	**for** $i \in \{0, ..., \lvert\mathbf{h}\rvert - 1\}$:	**for** $i \in \{0, ..., \lvert\mathbf{h}\rvert - 1\}$:	**for** $i \in \{0, ..., \lvert\mathbf{h}\rvert - 1\}$:	**for** $i \in \{0, ..., \lvert\mathbf{h}\rvert - 1\}$:	*(replaced with function* $\texttt{f}_{\texttt{crNKDF}}$*)*
$\quad \texttt{XTR}(\mathbf{h}_i)$	$\quad$ *(replaced by code)*				
	$\quad$ **assert** $S_{Ext} \neq \bot$	$\quad$ **assert** $S_{Ext} \neq \bot$	$\quad$ **assert** $S_{Ext} \neq \bot$	$\quad$ *(moved up)*	
	$\quad k, hon \leftarrow \texttt{Get}(\mathbf{h}_i)$	$\quad$ *(using vector assignment notation)*			
	$\quad k' \leftarrow \texttt{xtr}(S_{Ext}, k)$	$\quad k' \leftarrow \texttt{xtr}(S_{Ext}, \mathbf{k}_i)$	*(use vector assigment notation)*		
	$\quad \texttt{Set}(\mathbf{h}_i, k', hon)$	$\quad \texttt{Set}(\mathbf{h}_i, k', \mathbf{hon}_i)$	*(drop Set-then-Get)*		
$\quad \texttt{Eval}(\mathbf{h}_i, (ctx1, i))$	$\quad$ *(replaced by code)*				
	$\quad$ **if** $T_{PRF}[(ctx1, i)] = \bot$:	$\quad$ *(redundant check, removed)*			
	$\quad\quad T_{PRF}[(ctx1, i)] \leftarrow \mathbf{h}_i$	$\quad$ *(redundant assignment, removed)*			
	$\quad$ **assert** $T_{PRF}[(ctx1, i)] = \mathbf{h}_i$	$\quad$ *(redundant check, removed)*			
	$\quad k, hon \leftarrow \texttt{Get}(\mathbf{h}_i)$	$\quad k, hon \leftarrow \texttt{Get}(\mathbf{h}_i)$	$\quad$ *(drop Set-then-Get)*		
	$\quad k' \leftarrow \texttt{f}_{\texttt{PRF}}(k, (ctx1, i), ol)$	$\quad k' \leftarrow \texttt{f}_{\texttt{PRF}}(k, (ctx1, i), ol)$	$\quad k' \leftarrow \texttt{f}_{\texttt{PRF}}(\mathbf{k}_i, (ctx1, i), ol)$	$\quad \mathbf{k}_i \leftarrow \texttt{f}_{\texttt{PRF}}(\mathbf{k}_i, (ctx1, i), ol)$	*(replaced with function* $\texttt{f}_{\texttt{crNKDF}}$*)*
	$\quad \texttt{Set}((ctx1, i), k', hon)$	$\quad \texttt{Set}((ctx1, i), k', hon)$	$\quad \texttt{Set}((ctx1, i), k', \mathbf{hon}_i)$	$\quad$ *(drop Set-then-Get)*	
$\texttt{XOR}(ctx1, (0, ..., \lvert\mathbf{h}\rvert - 1))$	*(replaced by code)*				
	if $T_{XOR}[ctx1] = \bot$:	*(redundant check, removed)*			
	$\quad T_{XOR}[ctx1] \leftarrow \texttt{sort}((0, ..., \lvert\mathbf{h}\rvert - 1))$	$\quad$ *(redundant assignment, removed)*			
	assert $T_{XOR}[ctx1] = \texttt{sort}((0, ..., \lvert\mathbf{h}\rvert - 1))$	*(redundant check, removed)*			
	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}((ctx1, (0, ..., \lvert\mathbf{h}\rvert - 1)))$	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}((ctx1, (0, ..., \lvert\mathbf{h}\rvert - 1)))$	$\mathbf{k}, \mathbf{hon} \overset{\text{vec}}{\leftarrow} \texttt{Get}((ctx1, (0, ..., \lvert\mathbf{h}\rvert - 1)))$	*(drop Set-then-Get)*	
	$k' \leftarrow \bigoplus \mathbf{k}$	$k' \leftarrow \bigoplus \mathbf{k}$	$k' \leftarrow \bigoplus \mathbf{k}$	$k' \leftarrow \bigoplus \mathbf{k}$	*(replaced with function* $\texttt{f}_{\texttt{crNKDF}}$*)*
	$hon' \leftarrow \bigvee \mathbf{hon}$	$hon' \leftarrow \bigvee \mathbf{hon}$	$hon' \leftarrow \bigvee \mathbf{hon}$	*(moved down)*	
	$\texttt{Set}(ctx1, k', hon')$	$\texttt{Set}(ctx1, k', hon')$	$\texttt{Set}(ctx1, k', hon')$	*(drop Set-then-Get)*	
$\texttt{CREval}(ctx1, ctx1, ctx2)$	*(replaced by code)*				
	assert $S_{cr} \neq \bot$	**assert** $S_{cr} \neq \bot$	**assert** $S_{cr} \neq \bot$	*(moved up)*	
	if $T_{cr}[ctx1] = \bot$:	*(redundant check, removed)*			
	$\quad T_{cr}[ctx1] \leftarrow ctx1$	$\quad$ *(redundant assignment, removed)*			
	assert $T_{cr}[ctx1] = ctx1$	*(redundant check, removed)*			
	$k, hon \leftarrow \texttt{Get}(ctx1)$	$k, hon \leftarrow \texttt{Get}(ctx1)$	$k, hon \leftarrow \texttt{Get}(ctx1)$	*(drop Set-then-Get)*	
	$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (ctx1, ctx2), S_{cr})$	$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (ctx1, ctx2), S_{cr})$	$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (ctx1, ctx2), S_{cr})$	$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k', ctx1, S_{cr})$	*(replaced with function* $\texttt{f}_{\texttt{crNKDF}}$*)*
				$hon' \leftarrow \bigvee \mathbf{hon}$	$hon' \leftarrow \bigvee \mathbf{hon}$
	$\texttt{Set}((ctx1, ctx2), k', hon)$	$\texttt{Set}((ctx1, ctx2), k', hon)$	$\texttt{Set}((ctx1, ctx2), k', hon)$	$\texttt{Set}((ctx1, ctx2), k', hon)$	$\texttt{Set}((ctx1, ctx2), k', hon)$

Fig. 22: Inlining NKDF

```
$\texttt{Init}() \to (\{0,1\}^l)$
return $\texttt{Init}, \texttt{CRInit}()$      (replaced by code)
                                  assert $S_{Ext} = \bot$
                                  $S_{Ext} \leftarrow\$ \{0,1\}^{sl}$
                                  assert $S_{cr} = \bot$
                                  $S_{cr} \leftarrow\$ \{0,1\}^{sl}$
                                  return $S_{Ext}, S_{cr}$
```

```
$\texttt{CRDerive}(\mathbf{pkX}, \mathbf{pkY}, ctx1, ctx2) \to ()$
assert $|\mathbf{pkX}| = |\mathbf{pkY}|$
$(\mathbf{pkX}, \mathbf{pkY}) \overset{vec}{\leftarrow} \texttt{sort}(\mathbf{pkX}, \mathbf{pkY})$
if $T[ctx1] = \bot$ :
  $T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$

for $i \in \{0, ..., |\mathbf{pkX}| - 1\}$ :
  $\texttt{Pow}(\mathbf{pkX}_i, \mathbf{pkY}_i)$
  $\texttt{XTR}(\mathbf{pkX}_i, \mathbf{pkY}_i)$
  $\texttt{Eval}(\mathbf{pkX}_i, \mathbf{pkY}_i, (ctx1, i))$
$\texttt{XOR}(ctx1, (0, ..., |\mathbf{h}| - 1))$
$\texttt{CREval}(ctx1, ctx1, ctx2)$
```

```
if $T[ctx1] = \bot$ :
  $T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$

for $i \in \{0, ..., |\mathbf{pkX}| - 1\}$ :
  (replaced by code)
  $skX, xHon \leftarrow \texttt{Get}(\mathbf{pkX}_i)$
  $skY, yHon \leftarrow \texttt{Get}(\mathbf{pkY}_i)$
  assert $skX \neq \bot \vee skY \neq \bot$
  if $skX \neq \bot$ :
    $k \leftarrow \mathbf{pkY}_i^{skX}$
  else :
    $k \leftarrow \mathbf{pkX}_i^{skY}$
  $\mathbf{pkX}_i, \mathbf{pkY}_i \leftarrow \texttt{sort}(\mathbf{pkX}_i, \mathbf{pkY}_i)$
  $\texttt{Set}((\mathbf{pkX}_i, \mathbf{pkY}_i), k, xHon \wedge yHon)$
  (replaced by code)
  assert $S_{Ext} \neq \bot$
  $k, hon \leftarrow \texttt{Get}((\mathbf{pkX}_i, \mathbf{pkY}_i))$
  $k' \leftarrow \texttt{xtr}(S_{Ext}, k)$
  $\texttt{Set}((\mathbf{pkX}_i, \mathbf{pkY}_i), k', hon)$
  (replaced by code)
  if $T_{PRF}[(ctx1, i)] = \bot$ :
    $T_{PRF}[(ctx1, i)] \leftarrow (\mathbf{pkX}_i, \mathbf{pkY}_i)$
  assert $T_{PRF}[(ctx1, i)] = (\mathbf{pkX}_i, \mathbf{pkY}_i)$
  $k, hon \leftarrow \texttt{Get}((\mathbf{pkX}_i, \mathbf{pkY}_i))$
  $k' \leftarrow \texttt{f}_{\texttt{PRF}}(k, (ctx1, i), ol)$
  $\texttt{Set}((ctx1, i), k', hon)$
(replaced by code)
if $T_{XOR}[ctx1] = \bot$ :
  $T_{XOR}[ctx1] \leftarrow \texttt{sort}((0, ..., |\mathbf{h}| - 1))$
assert $T_{XOR}[ctx1] = \texttt{sort}((0, ..., |\mathbf{h}| - 1))$
$\mathbf{k}, \mathbf{hon} \overset{vec}{\leftarrow} \texttt{Get}((ctx1, (0, ..., |\mathbf{h}| - 1)))$
$k' \leftarrow \bigoplus \mathbf{k}$
$hon' \leftarrow \bigvee \mathbf{hon}$
$\texttt{Set}(ctx1, k', hon')$
(replaced by code)
assert $S_{cr} \neq \bot$
if $T_{cr}[ctx1] = \bot$ :
  $T_{cr}[ctx1] \leftarrow ctx1$
assert $T_{cr}[ctx1] = ctx1$
$k, hon \leftarrow \texttt{Get}(ctx1)$
$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (ctx1, ctx2), S_{cr})$

$\texttt{Set}((ctx1, ctx2), k', hon)$
```

```
if $T[ctx1] = \bot$ :
  $T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$

$\mathbf{skX}, \mathbf{honX} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkX})$
$\mathbf{skY}, \mathbf{honY} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkY})$
$\mathbf{hon} \overset{vec}{\leftarrow} \mathbf{honX} \wedge \mathbf{honY}$

for $i \in \{0, ..., |\mathbf{pkX}| - 1\}$ :
(replaced by vector assignment notation)
(replaced by vector assignment notation)
  assert $\mathbf{skX}_i \neq \bot \vee \mathbf{skY}_i \neq \bot$
  if $\mathbf{skX}_i \neq \bot$ :
    $k \leftarrow \mathbf{pkY}^{\mathbf{skX}_i}$
  else :
    $k \leftarrow \mathbf{pkX}_i^{\mathbf{skY}_i}$
  (redundant operation)
  $\texttt{Set}((\mathbf{pkX}_i, \mathbf{pkY}_i), k, \mathbf{hon}_i)$

  assert $S_{Ext} \neq \bot$
  $k, hon \leftarrow \texttt{Get}((\mathbf{pkX}_i, \mathbf{pkY}_i))$
  $k' \leftarrow \texttt{xtr}(S_{Ext}, k)$
  $\texttt{Set}((\mathbf{pkX}_i, \mathbf{pkY}_i), k', hon)$

  (redundant check, removed)
  (redundant assignment, removed)
  (redundant check, removed)
  $k, hon \leftarrow \texttt{Get}((\mathbf{pkX}_i, \mathbf{pkY}_i))$
  $k' \leftarrow \texttt{f}_{\texttt{PRF}}(k, (ctx1, i), ol)$
  $\texttt{Set}((ctx1, i), k', hon)$

(redundant check, removed)
(redundant assignment, removed)
(redundant check, removed)
$\mathbf{k}, \mathbf{hon} \overset{vec}{\leftarrow} \texttt{Get}((ctx1, (0, ..., |\mathbf{h}| - 1)))$
$k' \leftarrow \bigoplus \mathbf{k}$
$hon' \leftarrow \bigvee \mathbf{hon}$
$\texttt{Set}(ctx1, k', hon')$

assert $S_{cr} \neq \bot$
(redundant check, removed)
(redundant assignment, removed)
(redundant check, removed)
$k, hon \leftarrow \texttt{Get}(ctx1)$
$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (ctx1, ctx2), S_{cr})$

$\texttt{Set}((ctx1, ctx2), k', hon)$
```

```
if $T[ctx1] = \bot$ :
  $T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$

$\mathbf{skX}, \mathbf{honX} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkX})$
$\mathbf{skY}, \mathbf{honY} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkY})$
$\mathbf{hon} \overset{vec}{\leftarrow} \mathbf{honX} \wedge \mathbf{honY}$
$\mathbf{sk}, \mathbf{pk} \leftarrow \texttt{pick}(\mathbf{skX}, \mathbf{pkX}, \mathbf{skY}, \mathbf{pkY})$
$\mathbf{k} \overset{vec}{\leftarrow} \mathbf{pk}^{\mathbf{sk}}$

for $i \in \{0, ..., |\mathbf{pkX}| - 1\}$ :

  (moved up - pick then exponentiate)
  (moved up - pick then exponentiate)
  (moved up - pick then exponentiate)
  (moved up - pick then exponentiate)
  (moved up - pick then exponentiate)

  $\texttt{Set}((\mathbf{pkX}_i, \mathbf{pkY}_i), \mathbf{k}_i, \mathbf{hon}_i)$

  assert $S_{Ext} \neq \bot$
  $k, hon \leftarrow \texttt{Get}((\mathbf{pkX}_i, \mathbf{pkY}_i))$
  $k' \leftarrow \texttt{xtr}(S_{Ext}, k)$
  $\texttt{Set}((\mathbf{pkX}_i, \mathbf{pkY}_i), k', hon)$

  $k, hon \leftarrow \texttt{Get}((\mathbf{pkX}_i, \mathbf{pkY}_i))$
  $k' \leftarrow \texttt{f}_{\texttt{PRF}}(k, (ctx1, i), ol)$
  $\texttt{Set}((ctx1, i), k', hon)$

$\mathbf{k}, \mathbf{hon} \overset{vec}{\leftarrow} \texttt{Get}((ctx1, (0, ..., |\mathbf{h}| - 1)))$
$k' \leftarrow \bigoplus \mathbf{k}$
$hon' \leftarrow \bigvee \mathbf{hon}$
$\texttt{Set}(ctx1, k', hon')$

assert $S_{cr} \neq \bot$

$k, hon \leftarrow \texttt{Get}(ctx1)$
$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (ctx1, ctx2), S_{cr})$

$\texttt{Set}((ctx1, ctx2), k', hon)$
```

```
if $T[ctx1] = \bot$ :
  $T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$

$\mathbf{skX}, \mathbf{honX} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkX})$
$\mathbf{skY}, \mathbf{honY} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkY})$
$\mathbf{hon} \overset{vec}{\leftarrow} \mathbf{honX} \wedge \mathbf{honY}$
$\mathbf{sk}, \mathbf{pk} \leftarrow \texttt{pick}(\mathbf{skX}, \mathbf{pkX}, \mathbf{skY}, \mathbf{pkY})$
$\mathbf{k} \overset{vec}{\leftarrow} \mathbf{pk}^{\mathbf{sk}}$

for $i \in \{0, ..., |\mathbf{pkX}| - 1\}$ :

  (drop Set-then-Get)

  assert $S_{Ext} \neq \bot$
  (drop Set-then-Get)
  $k' \leftarrow \texttt{xtr}(S_{Ext}, \mathbf{k}_i)$
  $\texttt{Set}((\mathbf{pkX}_i, \mathbf{pkY}_i), k', \mathbf{hon}_i)$

  $k, hon \leftarrow \texttt{Get}((\mathbf{pkX}_i, \mathbf{pkY}_i))$
  $k' \leftarrow \texttt{f}_{\texttt{PRF}}(k, (ctx1, i), ol)$
  $\texttt{Set}((ctx1, i), k', hon)$

$\mathbf{k}, \mathbf{hon} \overset{vec}{\leftarrow} \texttt{Get}((ctx1, (0, ..., |\mathbf{h}| - 1)))$
$k' \leftarrow \bigoplus \mathbf{k}$
$hon' \leftarrow \bigvee \mathbf{hon}$
$\texttt{Set}(ctx1, k', hon')$

assert $S_{cr} \neq \bot$

$k, hon \leftarrow \texttt{Get}(ctx1)$
$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (ctx1, ctx2), S_{cr})$

$\texttt{Set}((ctx1, ctx2), k', hon)$
```

```
if $T[ctx1] = \bot$ :
  $T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$
assert $S_{Ext} \neq \bot$
$\mathbf{skX}, \mathbf{honX} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkX})$
$\mathbf{skY}, \mathbf{honY} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkY})$
$\mathbf{hon} \overset{vec}{\leftarrow} \mathbf{honX} \wedge \mathbf{honY}$
$\mathbf{sk}, \mathbf{pk} \leftarrow \texttt{pick}(\mathbf{skX}, \mathbf{pkX}, \mathbf{skY}, \mathbf{pkY})$
$\mathbf{k} \overset{vec}{\leftarrow} \mathbf{pk}^{\mathbf{sk}}$

$\mathbf{k} \overset{vec}{\leftarrow} \texttt{xtr}(S_{Ext}, \mathbf{k})$
for $i \in \{0, ..., |\mathbf{pkX}| - 1\}$ :

  (moved up)

  (use vector assigment notation)
  (drop Set-then-Get)

  (drop Set-then-Get)
  $\mathbf{k}_i \leftarrow \texttt{f}_{\texttt{PRF}}(\mathbf{k}_i, (ctx1, i), ol)$
  $\texttt{Set}((ctx1, i), \mathbf{k}_i, \mathbf{hon}_i)$

$\mathbf{k}, \mathbf{hon} \overset{vec}{\leftarrow} \texttt{Get}((ctx1, (0, ..., |\mathbf{h}| - 1)))$
$k' \leftarrow \bigoplus \mathbf{k}$
$hon' \leftarrow \bigvee \mathbf{hon}$
$\texttt{Set}(ctx1, k', hon')$

assert $S_{cr} \neq \bot$

$k, hon \leftarrow \texttt{Get}(ctx1)$
$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (ctx1, ctx2), S_{cr})$

$\texttt{Set}((ctx1, ctx2), k', hon)$
```

```
if $T[ctx1] = \bot$ :
  $T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$
assert $S_{Ext} \neq \bot$
$\mathbf{skX}, \mathbf{honX} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkX})$
$\mathbf{skY}, \mathbf{honY} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkY})$
$\mathbf{hon} \overset{vec}{\leftarrow} \mathbf{honX} \wedge \mathbf{honY}$
$\mathbf{sk}, \mathbf{pk} \leftarrow \texttt{pick}(\mathbf{skX}, \mathbf{pkX}, \mathbf{skY}, \mathbf{pkY})$
$\mathbf{k} \overset{vec}{\leftarrow} \mathbf{pk}^{\mathbf{sk}}$

$\mathbf{k} \overset{vec}{\leftarrow} \texttt{xtr}(S_{Ext}, \mathbf{k})$
for $i \in \{0, ..., |\mathbf{pkX}| - 1\}$ :

  $\mathbf{k}_i \leftarrow \texttt{f}_{\texttt{PRF}}(\mathbf{k}_i, (ctx1, i), ol)$
  (drop Set-then-Get)

(drop Set-then-Get)
$k' \leftarrow \bigoplus \mathbf{k}$
$hon' \leftarrow \bigvee \mathbf{hon}$
$\texttt{Set}(ctx1, k', hon')$

assert $S_{cr} \neq \bot$

$k, hon \leftarrow \texttt{Get}(ctx1)$
$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k, (ctx1, ctx2), S_{cr})$

$\texttt{Set}((ctx1, ctx2), k', hon)$
```

```
if $T[ctx1] = \bot$ :
  $T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$
assert $S_{Ext} \neq \bot, S_{cr} \neq \bot$
$\mathbf{skX}, \mathbf{honX} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkX})$
$\mathbf{skY}, \mathbf{honY} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkY})$
$\mathbf{hon} \overset{vec}{\leftarrow} \mathbf{honX} \wedge \mathbf{honY}$
$\mathbf{sk}, \mathbf{pk} \leftarrow \texttt{pick}(\mathbf{skX}, \mathbf{pkX}, \mathbf{skY}, \mathbf{pkY})$
$\mathbf{k} \overset{vec}{\leftarrow} \mathbf{pk}^{\mathbf{sk}}$

$\mathbf{k} \overset{vec}{\leftarrow} \texttt{xtr}(S_{Ext}, \mathbf{k})$
for $i \in \{0, ..., |\mathbf{pkX}| - 1\}$ :

  $\mathbf{k}_i \leftarrow \texttt{f}_{\texttt{PRF}}(\mathbf{k}_i, (ctx1, i), ol)$

$k' \leftarrow \bigoplus \mathbf{k}$
(moved down)
(drop Set-then-Get)

(moved up)

(drop Set-then-Get)
$k' \leftarrow \texttt{f}_{\texttt{crPRF}}(k', (ctx1\, ctx2), , S_{cr})$
$hon' \leftarrow \bigvee \mathbf{hon}$
$\texttt{Set}((ctx1, ctx2), k', hon)$
```

```
if $T[ctx1] = \bot$ :
  $T[ctx1] \leftarrow (\mathbf{pkX}, \mathbf{pkY})$
assert $T[ctx1] = (\mathbf{pkX}, \mathbf{pkY})$
assert $S_{Ext} \neq \bot, S_{cr} \neq \bot$
$\mathbf{skX}, \mathbf{honX} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkX})$
$\mathbf{skY}, \mathbf{honY} \overset{vec}{\leftarrow} \texttt{Get}(\mathbf{pkY})$
$\mathbf{hon} \overset{vec}{\leftarrow} \mathbf{honX} \wedge \mathbf{honY}$
$\mathbf{sk}, \mathbf{pk} \leftarrow \texttt{pick}(\mathbf{skX}, \mathbf{pkX}, \mathbf{skY}, \mathbf{pkY})$
$\mathbf{k} \overset{vec}{\leftarrow} \mathbf{pk}^{\mathbf{sk}}$
$k' \leftarrow \texttt{f}_{\texttt{crNKDF}}(\mathbf{k}, ctx1, ol, S_{Ext}, ctx2, S_{cr})$
(replaced with function $\texttt{f}_{\texttt{crNKDF}}$)
(replaced with function $\texttt{f}_{\texttt{crNKDF}}$)

(replaced with function $\texttt{f}_{\texttt{crNKDF}}$)

(replaced with function $\texttt{f}_{\texttt{crNKDF}}$)

(replaced with function $\texttt{f}_{\texttt{crNKDF}}$)
$hon' \leftarrow \bigvee \mathbf{hon}$
$\texttt{Set}((ctx1, ctx2), k', hon)$
```

Fig. 23: Inlining DHNKDF