Optimal-Rate Non-Committing Encryption in a CRS Model[*]

Ran Canetti[†] Oxana Poburinnaya[‡] Mariana Raykova[§]

May 24, 2016

Abstract

Non-committing encryption (NCE) implements secure channels under adaptive corruptions in situations when data erasures are not trustworthy. In this paper we are interested in the *rate* of NCE, i.e. in how many bits the sender and receiver need to send per plaintext bit.

In initial constructions (e.g. Canetti, Feige, Goldreich and Naor, STOC 96) the length of both the receiver message, namely the public key, and the sender message, namely the ciphertext, is $m \cdot \mathrm{poly}(\lambda)$ for an m-bit message, where λ is the security parameter. Subsequent works improve efficiency significantly, achieving rate $\mathrm{poly}\log(\lambda)$.

We construct the first constant-rate NCE. In fact, our scheme has rate $1 + o(1)$, which is comparable to the rate of plain semantically secure encryption. Our scheme operates in the common reference string (CRS) model. Our CRS has size $\mathrm{poly}(m \cdot \lambda)$, but it is reusable for an arbitrary polynomial number of m-bit messages. In addition, it is the first NCE protocol with perfect correctness. We assume one way functions and indistinguishability obfuscation for circuits.

As an essential step in our construction, we develop a technique for dealing with adversaries that modify the inputs to the protocol adaptively depending on a public key or CRS that contains obfuscated programs, while assuming only standard (polynomial) hardness of the obfuscation mechanism. This technique may well be useful elsewhere.

[*]This work was done [in part] while the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467.

[†]Tel-Aviv University and Boston University. canetti@bu.edu. Supported in addition by the Check Point Institute for Information Security and NSF Algorithmic Foundations grant 1218461, NSF grant 1421102.

[‡]Boston University. oxanapob@bu.edu. Supported in addition by the Check Point Institute for Information Security and NSF Algorithmic Foundations grant 1218461, NSF grant 1421102.

[§]SRI, Yale University. mariana@cs.columbia.edu. Supported by NSF grant 1421102

Contents

1 Introduction

Informally, *non-committing*, or *adaptively secure*, encryption (NCE) is an encryption scheme for which it is possible to generate a dummy ciphertext which is indistinguishable from a real one, but can later be opened to any message. This primitive is a central tool in building adaptively secure protocols: one can take an adaptively secure protocol in secure channels setting and convert it into adaptively secure protocol in computational setting by encrypting communications using NCE [CFGN96]. In particular, NCE scheme is secure under selective-opening attacks [DNRS99].

The ability to open dummy ciphertexts to any message has its price in efficiency: while for plain semantically secure encryption we have constructions with $O(\lambda)$-size, reusable public and secret keys for security parameter λ, and $m + \mathrm{poly}(\lambda)$-size ciphertext for m-bit messages, non-committing encryption has been far from being that efficient. Some justification for this state of affairs is the lower bound of Nielsen [Nie02], which shows that the secret key of any NCE has to be at least m where m is the overall number of bits decrypted with this key. Still, no bound is known on the size of the public key or the ciphertext.

We focus on building NCE with better efficiency: specifically, we optimize the *rate* of NCE, i.e. the total amount of communication sent per single bit of a plaintext.

Prior Work. The first construction of adaptively secure encryption, presented by Beaver and Haber [BH92], is interactive (3 rounds) and relies on the ability of parties to reliably erase parts of their internal state. An adaptively secure encryption that does not rely on secure erasures, or *non-committing encryption,* is presented in [CFGN96]. The scheme requires only two messages, just like standard encryption, and is based on joint-domain trapdoor permutations. It requires both the sender and the receiver to send $\Theta(\lambda^2)$ bits per each bit of a plaintext. Subsequent work has focused on reducing rate and number of rounds. Beaver [Bea97] and Damgård and Nielsen [DN00] propose a 3-round NCE protocol from, respectively, DDH and a *simulatable PKE* (which again can be built from similar assumptions to those of [CFGN96]) with $m \cdot \Theta(\lambda^2)$ bits overall communication for m bit messages, but only $m \cdot \Theta(\lambda)$ bits from sender to receiver. These results were improved by Choi et al. [CDMW09] who reduce the number of rounds to two, which matches optimal number of rounds since non-interactive NCE is impossible [Nie02]. Also they reduced simulatable PKE assumption to a weaker *trapdoor simulatable PKE assumption*; such a primitive can be constructed from factoring. A recent work of Hemenway et al. [HOR15] presented a two-round NCE construction based on the Φ-hiding assumption which has $\Theta(m \log m) + \mathrm{poly}(\lambda)$ ciphertext size and $m \cdot \Theta(\lambda)$ communication from receiver to sender. Hemenway, Ostrovsky, Richelson and Rosen [HORR16] construct NCE with rate $\mathrm{poly}\log(\lambda)$ under the ring-LWE assumption.

We remark that the recent results on adaptively secure multiparty computation (MPC) from indistinguishability obfuscation in the common reference string (CRS) model [CGP15, GP15, DKR15] do not provide an improvement of NCE rate. Specifically, [CGP15] and [DKR15] already use NCE as a building block in their constructions, and the resulting NCE is as inefficient as underlying NCE. The scheme by Garg et al. [GP15] does not use NCE, but their second message is of size $\mathrm{poly}(m\lambda)$ due to the statistically sound non-interactive zero knowledge proof involved.

Another line of work focuses on achieving better parameters for weaker notions of NCE where the adversary sees the internal state of only one of the parties (receiver or sender). Jarecki and Lysyanskaya [JL00] propose a scheme that is non-committing for the receiver only, which has two rounds and ciphertext expansion factor 3 (i.e., the ciphertext size is $3m + \mathrm{poly}(\lambda)$), under DDH assumption. Furthermore, their public key is also short and thus their scheme achieves rate 4. Canetti at al. [CHK05] construct a constant-rate NCE with erasures, meaning that the sender has to erase encryption randomness, and the receiver has to erase the randomness used for the initial key generation. Their NCE construction has rate 13.

Our Results. We present two NCE schemes with constant-rate in the CRS model. We first present a simpler construction which gives us rate 4, and then, using more sophisticated techniques, we construct our main scheme with rate $1 + o(1)$.

Our first construction is given by a rate-preserving transformation from any NCE with erasures to full NCE, assuming indistinguishability obfuscation ($i\mathcal{O}$) and one way functions (OWFs). Here we plug-in the scheme of [JL00] to obtain rate 4, or that of [CHK05] to obtain rate 13.

Our main construction assumes only $i\mathcal{O}$ and OWFs and achieves rate $1 + o(1)$. To be more precise, the public key, which is the first protocol message in our scheme, has size $\mathrm{poly}(\lambda)$. The ciphertext, which is the second message, has size $\mathrm{poly}(\lambda) + |m|$[1]. The CRS size is $O(\mathrm{poly}(m\lambda))$, but the CRS is reusable for any polynomially-many executions without an a priori bound on the number of executions. Thus when length $|m|$ of a plaintext is large, the scheme has overall rate that approaches 1.

In addition, this NCE scheme is the first to guarantee perfect correctness. Note that NCE in the plain model cannot be perfectly correct, and therefore some setup assumption is necessary to achieve this property.

Construction and Proof Techniques. Recall the definition of non-committing encryption: Such a scheme consists of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Sim})$, which satisfy usual correctness and security requirements. Additionally, the scheme should remain secure even if the adversary first decides to see the communications in the protocol and later corrupt the parties. This means that the simulator should be able to generate a dummy ciphertext c_f (without knowing which message it encrypts). Later, upon corruption of the parties, the simulator learns a message m, and it should generate internal state of the parties consistent with m and c_f - namely, encryption randomness of the sender and generation randomness of the receiver.

First attempts and our first construction. Recall that the puncturing technique of [SW14] adds a special trapdoor branch to a program, which allows to "explain" any input-output behavior of a program, i.e. to generate randomness consistent with a given input-output pair ([SW14, DKR15]). Given such a technique, we could try to build NCE as follows. Start from any rate-efficient non-committing encryption scheme in a model with erasures. Obfuscate the key generation algorithm Gen and put it in the CRS. The protocol then proceeds as follows: the receiver runs Gen, obtains (pk, sk), sends pk to the sender, gets back c and decrypts it with sk. In order to allow simulation of the receiver, augment Gen with a trapdoor which allows a simulator to come up with randomness for Gen that is consistent with (pk, sk). However, this approach doesn't allow to simulate the sender.

One natural way to allow simulation of the sender is to modify Gen: instead of outputting pk, it should output an obfuscated encryption algorithm $E = i\mathcal{O}(\mathsf{Enc}[pk])$ with the public key hardwired, and the receiver should send E (instead of pk) to the sender in round 1. In the simulation $\mathsf{Enc}[pk]$ can be augmented with a trapdoor, thus allowing to simulate the sender. The problem is that this scheme is no longer efficient: in all known constructions the trapdoor (and therefore the whole program E) has the size of at least $\lambda|m|$, meaning that the rate is at least λ.[2]

Another attempt to allow simulation of the sender is to add to the CRS an obfuscated encryption program $E' = i\mathcal{O}(\mathsf{Enc}(pk, m, r))$, augmented with a trapdoor in the simulation. Just like in the initial scheme, the receiver should send pk to the sender; however, instead of computing c directly using pk, the sender should run the obfuscated program E' on pk, m and r. This scheme allows simulating both the sender and the receiver, and at the same time keeps the communication as short as in the original PKE. However, we can only prove *selective* security of this scheme, meaning that the adversary has to commit to the challenge message m *before* it sees the CRS. This is a limitation of the puncturing technique being used: in the security proof the input to the program Enc, including message m, has to be hardwired into the program.

We get around this issue by using another level of indirection. Instead of publishing $E' = i\mathcal{O}(\mathsf{Enc}(pk, m, r))$ in the CRS, we publish a program GenEnc which generates E' and outputs it.

Thus our first protocol works as follows: the receiver uses Gen to generate (pk, sk) and sends pk to the sender. The sender runs GenEnc and obtains E', and then executes $E'(pk, m, r) \to c$ and sends c back to the receiver. Note that

[1] In fact, the precise length of communication for transmitting m is $4\lambda + |m|$, i.e. it scales linearly with security parameter.

[2] This is due to the fact that this trapdoor uses a punctured PRF applied to the message m, and, to the best of our knowledge, in all known constructions of PPRFs the size of a punctured key is at least $\lambda|m|$.

GenEnc doesn't take m as input, therefore there is no need to hardwire m into CRS and in particular there is no need to know m at the CRS generation step.

Our main construction. We give another construction of NCE, which achieves asymptotically optimal rate. That is, the amount of bits sent is $|m| + \mathrm{poly}(\lambda)$, and by setting m to be long enough, we can achieve rate arbitrarily close to 1. The new scheme assumes only indistinguishability obfuscation and one-way functions.

First we briefly outline the main idea behind our construction. Our starting point is the [SW14] PKE scheme from $i\mathcal{O}$ and one way functions. We first show that this scheme is in fact NCE with erasures. We then use ideas from the deniable encryption of [SW14] to modify the public key so as to allow simulating the sender. However, this results in a long public key. To avoid sending this long public key from the receiver to the sender, we instead send a short token t, which can be used by special programs to generate the same pair of (pk, sk) of the underlying NCE with erasures. This is done by first computing the generation randomness $r_{\mathsf{Gen}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t)$ using a shared key MSK, then sampling (pk, sk) from r_{Gen} and finally outputting whatever is needed for parties, i.e. c or sk. Thus the token t can be thought of as an encapsulation of the generation randomness r_{Gen} of the underlying NCE with erasures. (Clearly r_{Gen} itself must remain unknown to the adversary.) Since we do not send the public key of the underlying NCE, we only need that the underlying NCE has short ciphertexts; public keys can be long.

Security-wise, the main difference between our first construction of NCE and this one is that here we reveal a lot of information about how r_{Gen} was generated. Indeed, in the previous scheme the adversary learned pk but never learned the values used to generate it. Here the adversary sees t in the clear, and the secrecy of MSK is the only thing which keeps r_{Gen} secret. This has its effect on the proof - we have to puncture MSK and hardwire/remove public keys inside programs, which results in additional requirements on the underlying NCE scheme with erasures.

More concretely, our construction proceeds in two steps. We first define and construct an augmented version of NCE with erasures, called *same-public-key non-committing encryption with erasures (seNCE).* Our seNCE scheme will have short ciphertexts, i.e. ciphertext size is $m + \mathrm{poly}(\lambda)$. However, the public keys will still be long, namely $\mathrm{poly}(m\lambda)$.

The second step in our construction is to transform any seNCE into a full NCE scheme in the CRS model where the ciphertext size is preserved and the public key size depends only on security parameter. We now describe these steps in somewhat more detail.

SeNCE schemes. These are NCE schemes with the following additional properties:

- *security with erasures:* the receiver is allowed to erase its generation randomness (but not sk); the sender is allowed to erase its encryption randomness. (This means that sk is the only information the adversary expects to see upon corrupting both parties.)
- *same public key:* the generation and simulation algorithms executed on the same input r produce the same public keys.

Construction of seNCE. Our starting point is the PKE construction from $i\mathcal{O}$ and one way functions by Sahai and Waters [SW14]. Similarly to that scheme we set our public key to be an obfuscated program with a key k inside, which takes as inputs message m and randomness r and outputs a ciphertext $c = (c_1, c_2) = (\mathsf{prg}(r), \mathsf{F}_k(\mathsf{prg}(r)) \oplus m)$, where F is a pseudorandom function (PRF). However, instead of setting k to be a secret key, we set the secret key to be an obfuscated program (with k hardwired) which takes an input $c = (c_1, c_2)$ and outputs $\mathsf{F}_k(c_1) \oplus c_2$. Once the encryption and decryption programs are generated, the key k and the randomness used for the obfuscations are erased, and the only thing the receiver keeps is its secret key. Note that ciphertexts in the above scheme have length $m + \mathrm{poly}(\lambda)$.

To see that this construction is secure with erasures, consider the simulator that sets a dummy ciphertext c_f to be a random value. To generate a fake decryption key sk_f, which behaves like a real secret key except that it decrypts c_f to a challenge message m, the simulator obfuscates a program (with m, c_f, k hardwired) that takes as input (c_1, c_2)

and does the following: if $c_1 = c_{f1}$ then the program outputs $c_{f2} \oplus c_2 \oplus m$, otherwise the output is $\mathsf{F}_k(c_1) \oplus c_2$. Encryption randomness of the sender, as well as k and obfuscation randomness of the receiver, are erased and do not need to be simulated. (Note that the simulated secret key is larger than the real secret key. So, to make sure that the programs have the same size, the real secret key has to be padded appropriately.)

Furthermore, the scheme has the same-public-key property: The simulated encryption key is generated in exactly the same way as the honest encryption key.

Finally note that this scheme has perfect correctness.

From seNCE to full NCE. Our first step is to enhance the given seNCE scheme, such that the scheme remains secure even when the sender is not allowed to erase its encryption randomness. Specifically, following ideas from the deniable encryption of Sahai and Waters [SW14], we add a trapdoor branch to the encryption program, i.e. the public key. This allows the simulator to create fake randomness $r_{f,\mathsf{Enc}}$, which activates this trapdoor branch and makes the program output c_f on input m. In order to create such randomness, the simulator generates $r_{f,\mathsf{Enc}}$ as an encryption (using a scheme with pseudorandom ciphertexts[3]) of an instruction for the program to output c_f. The program will first try to decrypt $r_{f,\mathsf{Enc}}$ and check whether it should output c_f via trapdoor branch, or execute a normal branch instead.

The above construction of enhanced seNCE still has the following shortcomings. First, its public key (recall that it is an encryption program) is long: the program has to be padded to be at least of size $\mathrm{poly}(\lambda) \cdot |m|$, since in the proof the keys for the trapdoor branch are punctured and have an increased size, and therefore the size of an obfuscated program is $\mathrm{poly}(m\lambda)$. [4] Second, the simulator still cannot simulate the randomness which the receiver used to generate its public key, e.g. keys for the trapdoor branch and randomness for obfuscation. Third, the scheme is only selectively secure, meaning that the adversary has to fix the message *before* it sees a public key. This is due to the fact that our way for explaining a given output (i.e. trapdoor branch mechanism) requires hardwiring the message inside the encryption program in the proof.

We resolve these issues by adding another "level of indirection" for the generation of obfuscated programs. Specifically, we introduce a common reference string that will contain two obfuscated programs, called GenEnc and GenDec, which are generated independently of the actual communication of the protocol and can be reused for unboundedly many messages. The CRS allows the sender and the receiver to locally and independently generate their long public and private keys for the underlying enhanced seNCE while communicating only a short token. Furthermore, we will only need to puncture these programs at points which are unrelated to the actual encrypted and decrypted messages.

The NCE scheme. The receiver chooses randomness r_{GenDec} and runs a CRS program $\mathsf{GenDec}(r_{\mathsf{GenDec}})$. This program uses r_{GenDec} to sample a short token t. Next the program uses this token t to internally compute a secret generation randomness r_{seNCE}, from which it derives (pk, sk) pair for underlying seNCE scheme. Finally, the program outputs (t, sk). The public key (i.e., the receiver's message) is the token t.

The sender obtains the program GenEnc from the CRS and computes $\mathsf{GenEnc}(t, r_{\mathsf{GenEnc}})$ where r_{GenEnc} is the sender's fresh randomness. We now describe program GenEnc. Similarly to GenDec, GenEnc first uses t to generate secret randomness r_{seNCE} for seNCE, and samples the same key pair (pk, sk) for seNCE as the receiver. Further, GenEnc generates trapdoor keys and obfuscation randomness, which it uses to compute a public key program $\mathsf{P}_{\mathsf{Enc}}[pk]$ of enhanced seNCE, which extends the underlying seNCE public key with a trapdoor as described above. $\mathsf{P}_{\mathsf{Enc}}[pk]$ is the output of GenEnc. After obtaining $\mathsf{P}_{\mathsf{Enc}}$, the sender chooses encryption randomness r_{Enc} and runs $c \leftarrow \mathsf{P}_{\mathsf{Enc}}[pk](m, r_{\mathsf{Enc}})$. In its response message, the sender sends c to the receiver, who decrypts it using sk.

Correctness of this scheme follows from correctness of the seNCE scheme, since at the end a message is being encrypted and decrypted using the seNCE scheme. To get some idea of why security holds, note that the seNCE generation randomness r_{seNCE} is only computed internally by the programs. This value is never revealed to the adversary,

[3]For this purpose we use a *puncturable deterministic encryption scheme* (PDE), since it is $i\mathcal{O}$-friendly and has pseudorandom ciphertexts.

[4] To the best of our knowledge, in all known puncturable PRFs the size of a punctured key applied to m is at least $\lambda|m|$

and therefore can be thought of as being “erased”. In particular, if we had a VBB obfuscation, we could almost immediately reduce security of our scheme to security of seNCE. Due to the fact that we only have $i\mathcal{O}$, the actual security proof becomes way more intricate.

To see how we resolved the three issues from above (namely, with the length of the public key, with simulating the receiver, and with selective security), note:

(a) The only information communicated between sender and receiver is the short token t which depends only on the security parameter, and the ciphertext c which has size $O(\lambda) + |m|$. Thus the total communication is $O(\lambda) + |m|$.
(b) The simulator will show different programs $\mathrm{GenEnc}\!:\!\mathrm{Sim}$, $\mathrm{GenDec}\!:\!\mathrm{Sim}$ and $\mathrm{Enc}\!:\!\mathrm{Sim}$, which have trapdoor branches inside; they allow the simulator to “explain” the randomness for any desired output, thus allowing it to simulate internal state of both parties.
(c) We no longer need to hardwire message-dependent values into the programs in the CRS, which previously made security only selective. Indeed, in a real world the inputs and outputs of these programs no longer depend on the message sent. They still do depend on the message in the ideal world (for instance, the output of GenDec is sk_m); however, due to the trapdoor branches in the programs it is possible for the simulator to encode sk_m into randomness r_{GenDec} rather than the program GenDec itself. Therefore m can be chosen adaptively after seeing the CRS (and the public key).

In the proof of our NCE we crucially use the same public-key property of underlying seNCE: Our programs use the master secret key MSK to compute the generation randomness r_{seNCE} from token t, and then sample seNCE keys (pk, sk) using this randomness. In the proof we hardwire pk in the CRS, then puncture MSK and choose r_{seNCE} at random. Next we switch the seNCE values, including the public key pk, to simulated ones. Then we choose r_{seNCE} as a result of a PRF, and unhardwire pk. In order to unhardwire (now simulated) pk from the program and compute $(pk, sk) = \mathsf{F}_{\mathsf{MSK}}(r_{\mathsf{seNCE}})$ instead, simulated pk generated from r_{seNCE} should be exactly the same as the real public key pk which the program normally produces by running $\mathsf{seNCE.Gen}(r_{\mathsf{seNCE}})$. This ensures that the programs with and without pk hardwired have the same functionality, and thus security holds by $i\mathcal{O}$.

An additional interesting property of this transformation is that is preserves the correctness of underlying seNCE scheme, meaning that if seNCE is computationally (statistically, perfectly) correct, then the resulting NCE is also computationally (statistically, perfectly) correct. Therefore, when instantiated with our perfectly correct seNCE scheme presented earlier, the resulting NCE achieves perfect correctness. To the best of our knowledge, this is the first NCE scheme with such property.

Shrinking the secret key. The secret key in the above scheme consists of an obfuscated program D, where D is the secret key (i.e. decryption program) for the seNCE scheme, together with some padding that will leave room to “hardwire”, in the hybrid distributions in the proof of security, the $|m|$-bit plaintext m into D. Overall, the description size of D is $|m| + O(\lambda)$; when using standard IO, this means that the obfuscated version of D is of size $\mathrm{poly}(|m|\lambda)$.

Still, using the succinct Turing and RAM machine obfuscation of [KLW15, CHJV15, BGL$^+$15, CH15] it is possible to obtain program obfuscation where the size of the obfuscated program is the size of the original program plus a polynomial in the security parameter. This can be done in a number of ways. One simple way is to generate the following (short) obfuscated TM machine OU: The input is expected to contain a description of a program that is one-time-padded, and then authenticated using a signed accumulator as in [KLW15], all with keys expanded from an internally known short key. The machine decrypts, authenticates, and then runs the input circuit. Now, to obfuscate a program, simply one time pad the program, authenticate it, and present it alongside machine OU with the authentication information and keys hardwired.

Augmented explainability compiler. In order to implement the trapdoor branch in the proof of our NCE scheme, we use among other things the “hidden sparse triggers” method of Sahai and Waters [SW14]. This method proved to be useful in other applications as well, and Dachman-Soled et al [DKR15] abstracted it into a primitive called “explainability compiler”. Roughly speaking, explainability compiler turns a randomized program into its “trapdoored” version, such that it becomes possible, for those who know faking keys, to create fake randomness which is consistent with a given input-output pair.

We use a slightly modified version this primitive, which we call an *augmented explainability compiler*. The main difference here is that we can use the original (unmodified) program in the protocol, and only in the proof replace it with its trapdoor version. This is important for perfect correctness of NCE: none of the programs GenEnc, GenDec, and Enc in the real world contain trapdoor branches (indeed, if there was a trapdoor branch in, say, encryption program Enc, it would be possible that an honest sender accidentally chooses randomness which contains an instruction to output an encryption of 0, making the program output this encryption of 0 instead of an encryption of m). Another difference is technical: both their and our compiler add an extracting PRF to the program, but our compiler uses a special PRF with a sparse image. In the proof of security of NCE we exploit the structure of our compiler and make use of the sparseness of this PRF.

2 Preliminaries

2.1 Non-committing Encryption and its Variants.

Non-committing encryption. Non-committing encryption is an adaptively secure encryption scheme, i.e., it remains secure even if the adversary decides to see the ciphertext first and only later corrupt parties. This means that the simulator should be able to first present a "dummy" ciphertext without knowing what the real message m is. Later, when parties are corrupted and the simulator learns m, the simulator should be able to present receiver decryption key (or receiver randomness) which decrypts dummy c to m and sender randomness under which m is encrypted to c.

Definition 1. *A non-committing encryption scheme for a message space $M = \{0,1\}^l$ is a tuple of algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Sim})$*, such that correctness and security hold:*

- **Correctness:** *For all $m \in M$* $\Pr\left[m = m' \;\middle|\; \begin{array}{l} (pk, sk) \leftarrow \mathsf{Gen}(1^\lambda, r_{\mathsf{Gen}}); \\ c \leftarrow \mathsf{Enc}_{pk}(m, r_{\mathsf{Enc}}); \\ m' \leftarrow \mathsf{Dec}_{sk}(c) \end{array} \right] \geq 1 - \mathsf{negl}(\lambda).$

- **Security:** *An adversary cannot distinguish between real and simulated ciphertexts and internal state even if it chooses message m adaptively depending on the public key pk. More concretely, no PPT adversary $\mathcal{A}$ can win the following game with more than negligible probability:*

  A challenger chooses random $b \in \{0,1\}$. If $b = 0$, it runs the following experiment (real):

  1. *It chooses randomness r_{Gen} and creates $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda, r_{\mathsf{Gen}})$. It shows pk to the adversary.*
  2. *The adversary chooses message m.*
  3. *The challenger chooses randomness r_{Enc} and creates $c \leftarrow \mathsf{Enc}(pk, m; r_{\mathsf{Enc}})$. It shows $(c, r_{\mathsf{Enc}}, r_{\mathsf{Gen}})$ to the adversary.*

  If $b = 1$, the challenger runs the following experiment (simulated):

  1. *It runs $(pk^s, c^s) \leftarrow \mathsf{Sim}(1^\lambda)$. It shows pk^s to the adversary.*
  2. *The adversary chooses message m.*
  3. *The challenger runs $(r^s_{\mathsf{Enc}}, r^s_{\mathsf{Gen}}) \leftarrow \mathsf{Sim}(m)$ and shows $(c^s, r^s_{\mathsf{Enc}}, r^s_{\mathsf{Gen}})$ to the adversary.*

  The adversary outputs a guess b' and wins if $b = b'$.

In this definition we only spell out the case where *both* parties are corrupted, and all corruptions happen *after* the execution and *simultaneously*. Indeed, if any of the parties is corrupted *before* the ciphertext is sent, then the simulator learns m and can present honest execution of the protocol; therefore we concentrate on the case where corruptions happen afterwards. Next, m is the only information the simulator needs, and after learning it (regardless of which party was corrupted) the simulator can already simulate *both* parties; thus we assume that corruptions of parties happen simultaneously. Finally, without loss of generality we assume that *both* parties are corrupted: if only one or no party is corrupted, then the adversary sees strictly less information in the experiment, and therefore cannot distinguish between real execution and simulation, as long as the scheme is secure under our definition.

Note that this definition only allows parties to encrypt a single message under a given public key. This is due to impossibility result of Nielsen [Nie02], who showed that a secret key of any NCE can support only bounded number of ciphertexts. If one needs to send many messages, it can run several instances of a protocol (each with a fresh pair of keys). Security for this case can be shown via a simple hybrid argument.

Non-committing encryption in a programmable common reference string model. In this work we build NCE in a CRS model, meaning that both parties and the adversary are given access to a CRS, and the simulator, in addition to simulating communications and parties' internal state, also has to simulate the CRS. Before giving a formal definition, we briefly discuss possible variants of this definition.

Programmable CRS. One option is to consider a *global* (non-programmable) CRS model, where the CRS is given to the simulator, or *local* (programmable) CRS model, where the simulator is allowed to generate a CRS. The first variant is stronger and more preferable, but in our construction the simulator needs to know underlying trapdoors and we therefore focus on a weaker type.

Reusable CRS. Given the fact that in a non-committing encryption a public key can be used to send only bounded number of bits, a bounded-use CRS would force parties to reestablish CRS after sending each block of messages. Since sampling a CRS is usually an expensive operation, it is good to be able to generate a CRS which can be reused for any number of times set a priori. It is even better to have a CRS which can be reused any polynomially many times without any a priori bound. In our definition we ask a CRS to be reusable in this stronger sense.

Security of multiple executions. Unlike NCE in the standard model, in the CRS model single-execution security of NCE does not immediately imply multi-execution security. Indeed, in a reduction to a single-execution security we would have to, given a challenge and a CRS, simulate other executions. But we cannot do this since we didn't generate this CRS ourselves and do not know trapdoors. Therefore in our definition we explicitly require that the protocol remains secure even when the adversary sees many executions with the same CRS.

Definition 2. *An NCE scheme for a message space $M = \{0,1\}^l$ in a common reference string model is a tuple of algorithms* $(\mathsf{GenCRS}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Sim})$ *which satisfy correctness and security.*

Correctness: *For all $m \in M$* $$\Pr\left[m = m' \;\middle|\; \begin{array}{l} \mathsf{CRS} \leftarrow \mathsf{GenCRS}(1^\lambda); \\ (pk, sk) \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{CRS}; r_{\mathsf{Gen}}); \\ c \leftarrow \mathsf{Enc}_{pk}(m, \mathsf{CRS}; r_{\mathsf{Enc}}); \\ m' \leftarrow \mathsf{Dec}_{sk}(\mathsf{CRS}, c) \end{array} \right] \geq 1 - \mathsf{negl}(\lambda).$$

If this probability is equal to 1, then we say that the scheme is perfectly correct. [5]

Security: *For any PPT adversary $\mathcal{A}$, advantage of $\mathcal{A}$ in distinguishing the following two cases is negligible:*

A challenger chooses random $b \in \{0,1\}$. If $b = 0$, it runs the following experiment (real):

First it generates a CRS as $\mathsf{CRS} \leftarrow \mathsf{GenCRS}(1^\lambda, l)$. CRS is given to the adversary. Next the challenger does the following, depending on the adversary's request:

- On a request to initiate a protocol instance with session ID id, the challenger chooses randomness $r_{\mathsf{Gen,id}}$ and creates $(pk_{\mathsf{id}}, sk_{\mathsf{id}}) \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{CRS}, r_{\mathsf{Gen,id}})$. It shows pk_{id} to the adversary.
- On a request to encrypt a message m_{id} in a protocol instance with session ID id, the challenger chooses randomness $r_{\mathsf{Enc,id}}$ and creates $c_{\mathsf{id}} \leftarrow \mathsf{Enc}(pk_{\mathsf{id}}, m_{\mathsf{id}}; r_{\mathsf{Enc,id}})$. It shows c_{id} to the adversary.

[5] Note that this definition implies that there are no decryption errors for *any* CRS, computed by GenCRS.

- On a request to corrupt the sender of a protocol instance with ID id, the challenger shows $r_{\mathsf{Enc},\mathsf{id}}$ to the adversary.
- On a request to corrupt the receiver of a protocol instance with ID id, the challenger shows $r_{\mathsf{Gen},\mathsf{id}}$ to the adversary.

If $b = 1$, it runs the following experiment (simulated):

First it generates a CRS as $\mathsf{CRS}^s \leftarrow \mathsf{Sim}(1^\lambda, l)$. CRS^s is given to the adversary. Next the challenger does the following, depending on the adversary's request:

- On a request to initiate a protocol instance with session ID id, the challenger runs $(pk^s_{\mathsf{id}}, c^s_{\mathsf{id}}) \leftarrow \mathsf{Sim}(1^\lambda)$ and shows pk^s_{id} to the adversary.
- On a request to encrypt a message m_{id} in a protocol instance with session ID id, the challenger shows c^s_{id} to the adversary.
- On a request to corrupt the sender of a protocol instance with ID id, the challenger shows $r^s_{\mathsf{Enc},\mathsf{id}} \leftarrow \mathsf{Sim}(m_{\mathsf{id}})$ to the adversary.
- On a request to corrupt the receiver of a protocol instance with ID id, the challenger shows $r^s_{\mathsf{Gen},\mathsf{id}} \leftarrow \mathsf{Sim}(m_{\mathsf{id}})$ to the adversary.

The adversary outputs a guess b' and wins if $b = b'$.

Constant rate NCE. The rate of an NCE scheme is how many bits the sender and receiver need to communicate in order to transmit a single bit of a plaintext: NCE scheme for a message space $M = \{0,1\}^l$ has *rate* $f(l, \lambda)$, if $(|pk| + |c|)/l = f(l, \lambda)$. If $f(l, \lambda)$ is a constant, the scheme is said to have constant rate.

Same-public-key non-committing encryption with erasures (seNCE). Here we define a different notion of NCE which we call *same-public-key non-committing encryption with erasures (seNCE)*; it is used as a building block in our main construction. First, such a scheme allows parties to erase unnecessary information: the sender is allowed to erase its encryption randomness, and the receiver is allowed to erase its generation randomness r_{Gen} (but not its public or secret key). Furthermore, this scheme should have "the same public key" property, which says that both real generation and simulated generation algorithms should output *exactly the same* public key pk, if they are executed with the same random coins.

Definition 3. *The same-public-key non-committing encryption with erasures (seNCE) for a message space* $M = \{0,1\}^l$ *is a tuple of algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Sim})$*, such that correctness, security, and same-public-key property hold:*

- **Correctness:** *For all* $m \in M$ $\Pr\left[m = m' \;\middle|\; \begin{array}{l} (pk, sk) \leftarrow \mathsf{Gen}(1^\lambda, r_{Gen}); \\ c \leftarrow \mathsf{Enc}_{pk}(m, r_{\mathsf{Enc}}); \\ m' \leftarrow \mathsf{Dec}_{sk}(c) \end{array} \right] \geq 1 - \mathsf{negl}(\lambda)$.
- **Security with erasures:** *No PPT adversary* $\mathcal{A}$ *can win the following game with more than negligible probability:*

  A challenger chooses random $b \in \{0,1\}$. *If* $b = 0$*, it runs a real experiment:*

  1. *The challenger chooses randomness* r_{Gen} *and creates* $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda, r_{\mathsf{Gen}})$. *It shows* pk *to the adversary.*
  2. *The adversary chooses a message* m.
  3. *The challenger chooses randomness* r_{Enc} *and creates* $c \leftarrow \mathsf{Enc}(pk, m; r_{\mathsf{Enc}})$. *It shows* c *to the adversary.*
  4. *Upon corruption request, the challenger shows to the adversary the secret key* sk.

  If $b = 1$*, the challenger runs a simulated experiment:*

1. *A challenger generates simulated public key and ciphertext* $(pk^s, c^s) \leftarrow \mathsf{Sim}(1^\lambda)$[6]. *It shows* pk^s *to the adversary.*

2. *The adversary chooses a message* m.

3. *The challenger shows the ciphertext* c^s *to the adversary.*

4. *Upon corruption request, the challenger runs* $sk^s \leftarrow \mathsf{Sim}(m)$ *and shows to the adversary simulated secret key* sk^s.

The adversary outputs a guess b' *and wins if* $b = b'$.

- **The same public key:** *For any* r *let* $\mathsf{Gen}(1^\lambda, r) = (pk, sk); \mathsf{Sim}(1^\lambda, r) = (pk^s, c^s)$. *Then* $pk = pk^s$.

3 Explainability Compiler

In this section we briefly and very informally describe the properties of an explainability compiler, constructed in [SW14, DKR15], since we heavily use it in both constructions of NCE. Those familiar with the techniques of [SW14, DKR15] can skip this section and move to the actual NCE scheme in section 5. However, note that our compiler is slightly different. For a formal definition, proofs, and differences between [SW14, DKR15] compiler and ours, refer to appendix B.

A compiler takes a program and outputs a new program which now can be "explained" - i.e. if you know a special faking key f, you can give randomness consistent with any input-output pair. Just as an example, consider a program $\mathsf{Encrypt}[pk](m, r)$ which simply outputs an encryption of m with randomness r under a public key pk (fig. 1). Let's say our ultimate goal is to cheat the adversary and convince him that a zero-encrypting ciphertext $c = \mathsf{Enc}_{\mathsf{pk}}(0)$ was produced by running a program $\mathsf{Encrypt}[pk]$ on a message $m = 1$. Thus, we need to come up with fake, but innocent-looking randomness r_f such that $\mathsf{Encrypt}[pk](1, r_f) = c$.

It is not clear how to achieve this with the original program Encrypt (unless Enc is already a deniable encryption itself). However, we can achieve this by slightly changing an encryption program. Consider a trapdoored program Encrypt:td (fig. 1)) which differs from original Encrypt in two aspects. First, it doesn't use randomness r as it is: it generates internal (and hidden) randomness u using a PRF (which in fact also has to be an extractor). Second and more important, it has an additional, *trapdoor*, branch: before doing anything, it tries to interpret r as a ciphertext under a special faking key f and decrypt it. If decryption was successful, then it interprets the result as a pair (m', c'), which it views as an instruction to output c' on input m'. Thus, if the input to the program m is indeed equal to m', then it outputs c' and halts. If any of these fails (i.e. it couldn't decrypt, or $m \neq m'$), then it aborts the trapdoor branch and goes to a normal branch, encrypting m honestly.

If such a program Encrypt:td is used in encryption scheme, then it is possible for the simulator to explain $c = \mathsf{Enc}_{\mathsf{pk}}(0)$ as an encryption of 1. In order to do this, it should generate r_f as an encryption (under a faking key f) of $(1, c)$. Then the program Encrypt:td on input $1, r_f$ will decrypt r_f to an instruction "output c on input 1" and, since 1 is indeed the input, output c.

Several comments follow. First, to make sure fake r_f is indistinguishable from a real randomness, encryption under a faking key f should have pseudorandom ciphertexts.

Second, it is important that the actual encryption randomness u remains hidden even given randomness r (and in particular that the program doesn't directly use r to encrypt). Indeed, if the encryption randomness u was known, everybody could run a public encryption procedure $\mathsf{Enc}_{\mathsf{pk}}(1, u)$ and see that it doesn't result in the claimed ciphertext c.

Third, the set of randomness r which is a valid encryption under a faking key f should be sparse, otherwise encryption will not be correct. This is because each such r becomes "a point of incorrect behavior", outputting a ciphertext which is possibly unrelated to the input bit. In a real execution, when r is chosen at random, r must happen to be a valid

[6]We omit the random coins and state of Sim.

Example of the explainability compiler

```
Program Encrypt[pk](m, r)
  1. Set $c \leftarrow Enc_{\mathsf{pk}}(m; r)$
  2. Output $c$
Program Encrypt:td[pk](m, r)
  1. trapdoor branch:
     (a) Try to decrypt $r$ using faking keys. If decryption failed, goto normal branch;
     (b) If decryption succeeded, interpret the result as $m', c'$.
     (c) If $m' = m$, output $c'$ and halt. Otherwise goto normal branch.
  2. normal branch:
     (a) Set $u \leftarrow \mathsf{PRF}(m, r)$
     (b) Set $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m; u)$
     (c) Output $c$
Program Encrypt:r[pk](m, r)
  1. Set $u \leftarrow \mathsf{PRF}(m, r)$
  2. Set $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m; u)$
  3. Output $c$
Program Explain(m, c; $\rho$)
  1. Set $r_f$ to be an encryption under a faking key f of $(m, c, \rho)$
  2. Output $r_f$
```

Figure 1: Original program Encrypt and programs Encrypt:td, Encrypt:r and Explain, which are the result of running a compiler on Encrypt.

encryption under f with only negligible probability; this ensures that Encrypt:td almost always executes its normal branch and therefore encryption remains correct.

As follows from the previous comment, Encrypt:td which is run on fake r_f and m always executes its trapdoor branch, whereas when it is run on a truly random r, it executes its normal branch almost always (unless r accidentally happened to be a valid encryption under a faking key f). Therefore it should remain hidden from the adversary which branch is being run - otherwise it can easily distinguish. In particular, faking key f should remain hidden from the adversary, although it is hardcoded inside Encrypt:td.

Next, it is important that the trapdoor branch checks that m' in the instruction is the same as m in the input. If it didn't perform this input check and instead run "Decrypt r to m', c'; output c'" code, then the adversary could easily distinguish between a real r and fake r_f: Namely, executing Encrypt:td$(0, r_f)$ and Encrypt:td$(1, r_f)$ with a fake r_f would yield the same output c', whereas running Encrypt:td$(0, r)$ and Encrypt:td$(1, r)$ would result in two different ciphertexts, since a change in m results in a change in encryption randomness u.

Finally, note that this technique ensures that real r and $c =$ Encrypt:td$(1, r)$ is indistinguishable from $c =$ Encrypt:td$(0, r)$ and fake r_f which explains c on input $m = 1$; in words, this means that a ciphertexs encrypting 0 can be explained as encrypting 1. However, it is more convenient to require a different guarantee from the compiler: that it is possible to generate fake randomness r_f which explains c on its true plaintext m. This property is called *indistinguishability of explanations*, meaning that the adversary cannot distinguish between true r and fake r_f which both result in the same c on input m.

Preserving correctness of encryption. As discussed above, such a method inherently introduces a decryption error, since it is possible for a randomly chosen r to be a valid encryption of some m and c, so that Encrypt:td on input m, r outputs c, which can be unrelated to m. If we want our encryption to be perfectly correct, we need to make sure that in the real world there is no trapdoor branch in the program - it should only be in the simulated Encrypt:td. This can be achieved by generating Encrypt:r ("r" for "rerandomized", see fig. 1) in the real world and Encrypt:td in the ideal world. Needless to say, we need to prove that these two programs are indistinguishable.

Program Explain. Previously we used a faking key f to generate fake randomness r_f. As we discussed, f has to remain hidden from the adversary, and thus the adversary couldn't generate fake randomness itself. However, it is possible to give it such a possibility: one can show that real and fake randomness remains indistinguishable even if the adversary has an obfuscated program Explain, which generates fake randomness. More specifically, it takes as input the desired input-output pair m, c, as well as randomness ρ, and outputs r_f, which is an encryption of (m, c, ρ) under f. It is crucial that r_f is randomized with ρ: if it weren't, then the adversary could catch us by running $\mathsf{Explain}(m, c)$ itself and noticing that we gave it exactly the same r_f.

Properties of the compiled programs. Here we informally state the properties we achieve:

- **Indistinguishability of programs with an without the trapdoor**, namely Encrypt:td and Encrypt:r. Intuitively, this holds since the set of inputs on which they differ is tiny and hard to find. Note that if the adversary was given a program Explain, it could easily distinguish the two by running them on any fake r.
- **Indistinguishability of the source of output**: the adversary cannot distinguish whether a given c is the result of running Encrypt:td or original Encrypt, even given Encrypt:td and Explain.
- **Indistinguishability of explanations:** random r and $c = \mathsf{Encrypt{:}td}(m, r)$ is indistinguishable from $c = \mathsf{Encrypt{:}td}(m, r)$ and $r_f = \mathsf{Explain}(m, c; \rho)$, even given Encrypt:td and Explain. I.e. Encrypt:td on both (m, r) and (m, r_f) outputs the same c, but does this using different branches, which should be indistinguishable for the adversary.

4 Constant-rate Non-committing Encryption in the CRS model

In this section we give our simpler construction of a constant-rate non-committing encryption. For a construction of NCE with rate 1, see section 5.

More precisely, we describe the transformation from any non-committing encryption with erasures (eNCE) in the plain model to a full non-committing encryption in a CRS model, such that this transformation preserves the length of communication. Thus by using any constant-rate NCE with erasures (for instance, [JL00] or [CHK05]) we achieve a full constant-rate NCE.

The scheme is given on Fig. 2. The CRS contains a description of two programs, GenEnc and Gen. $\mathsf{Gen}(r_{\mathsf{Gen}})$ simply runs a generation algorithm of eNCE and outputs pk, sk. $\mathsf{GenEnc}(r_{\mathsf{GenEnc}})$ outputs an obfuscated program $\mathsf{Encrypt}(pk, m, r)$. $\mathsf{Encrypt}(pk, m, r)$ just computes a normal encryption of m under pk. All three algorithms are "rerandomized", i.e. randomness they use is obtained by applying an extracting PRF on their input. The protocols goes as follows: the receiver runs Gen to produce (pk, sk) and sends pk to the sender. Then the sender first runs GenEnc to produce a program E, and then runs E on pk, m, and obtains the ciphertext c, which it sends to the receiver.

Theorem 1. *Assume indistinguishability obfuscation for circuits and one way functions. Let* (eNCE.Gen, eNCE.Enc, eNCE.Dec) *be a non-committing encryption with erasures in a plain model. Then:*

1. *The scheme given on Fig. 2 is a full non-committing encryption scheme in a CRS model.*
2. *The length of communication (i.e. $|pk| + |c|$) is the same as the length of communication in underlying eNCE.*

Since a constant-rate NCE schemes with erasures do exist ([CHK05, JL00], under decisional composite residuosity assumption and DDH), we immediately get the following corollary:

Corollary 1. *Assuming one way functions, indistinguishability obfuscation, and hardness of decisional composite residuosity, there exist a constant-rate non-committing encryption in the CRS model.*

NCE from eNCE

CRS: obfuscated programs Gen and GenEnc

Inputs: sender's message m

- **Round 1.** The receiver chooses randomness r_{Gen} and generates keys $(pk, sk) \leftarrow \mathsf{Gen}(r_{\mathsf{Gen}})$. It sends pk to the sender.
- **Round 2.** The sender chooses randomness $r_{\mathsf{Enc}}, r_{\mathsf{GenEnc}}$. It runs $E \leftarrow \mathsf{GenEnc}(r_{\mathsf{GenEnc}})$ and then $c \leftarrow E(pk, m, r_{\mathsf{Enc}})$. It sends c to the receiver.
- The receiver decrypts $m' \leftarrow \mathsf{eNCE.Dec}(sk, c)$.

Program Gen(r_{Gen})

Inputs: randomness r_{Gen}

```
// hardcoded values: key for extracting PRF $\mathsf{Ext}_{\mathsf{Gen}}$
1. Set $r_{\mathsf{NCE}} \leftarrow F_{\mathsf{Ext}_{\mathsf{Gen}}}(r_{\mathsf{Gen}})$.
2. Set $(pk, sk) \leftarrow \mathsf{eNCE.Gen}(r_{\mathsf{NCE}})$
3. Output $(pk, sk)$
```

Program GenEnc(r$_{\mathsf{GenEnc}}$)

```
// hardcoded values: key for extracting PRF $\mathsf{Ext}_{\mathsf{GenEnc}}$
```

Inputs: message m, randomness r_{Enc}

```
1. Set $e \leftarrow \mathsf{F}_{\mathsf{Ext}_{\mathsf{GenEnc}}}(r_{\mathsf{Enc}})$;
2. Use $e$ to sample a key $\mathsf{Ext}_{\mathsf{Encrypt}}$ for an extracting PRF and obfuscation randomness $r_{i\mathcal{O}}$
3. Compute $E \leftarrow i\mathcal{O}(\mathsf{Encrypt}[\mathsf{Ext}_{\mathsf{Encrypt}}]; r_{i\mathcal{O}})$
4. Output $E$.
```

Program Encrypt(pk, m, r$_{\mathsf{Encrypt}}$)

```
// hardcoded values: key for extracting PRF $\mathsf{Ext}_{\mathsf{Encrypt}}$
```

Inputs: eNCE public key pk, message m, randomness r_{Encrypt}

```
1. Set $u \leftarrow \mathsf{F}_{\mathsf{Ext}_{\mathsf{Encrypt}}}(r_{\mathsf{Encrypt}})$.;
2. Compute $c \leftarrow Enc_{pk}(m; u)$
3. Output $c$.
```

Figure 2: Transformation from a non-committing encryption with erasures to a full non-committing encryption. Programs are padded to be of the same size as programs in the proof.

Simulation:

1. **Simulation of the CRS:**
   (a) Sample keys $\mathsf{Ext}_{\mathsf{Gen}}$, f_{Gen}, $\mathsf{Ext}_{\mathsf{GenEnc}}$, f_{GenEnc}, choose random s_{Gen}, s_{GenEnc} and set $S_{\mathsf{Gen}} \leftarrow \mathsf{prg}(s_{\mathsf{Gen}})$, $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$.
   (b) Set the CRS to be obfuscated programs Gen:Sim$[\mathsf{Ext}_{\mathsf{Gen}}, f_{\mathsf{Gen}}, S_{\mathsf{Gen}}]$ and GenEnc:Sim$[\mathsf{Ext}_{\mathsf{GenEnc}}, f_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}]$.

   For every execution, generate the following:
2. **Simulation of communication:** Run $\mathsf{eNCE.Sim}$ to generate pk, c, state.
3. **Simulation of internal state of the sender for message** m**:**
   (a) Sample keys $\mathsf{Ext}_{\mathsf{Encrypt}}$, f_{Encrypt} and randomness s_{Encrypt}. Set $S_{\mathsf{Encrypt}} \leftarrow \mathsf{prg}(s_{\mathsf{Encrypt}})$.
   (b) Set E to be an obfuscated program $\mathsf{Encrypt} : \mathsf{Sim}[\mathsf{Ext}_{\mathsf{Encrypt}}, f_{\mathsf{Encrypt}}, S_{\mathsf{Encrypt}}]$.
   (c) set $r_{\mathsf{Enc}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{Encrypt}}}(pk, m, c, \mathsf{prg}(\rho_1))$ for random ρ_1
   (d) set $r_{\mathsf{GenEnc}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{GenEnc}}}(E, \mathsf{prg}(\rho_2))$ for random ρ_2
   (e) set $r_{\mathsf{Enc}}, r_{\mathsf{GenEnc}}$ to be the internal state of the sender.
4. **Simulation of internal state of the receiver for message** m**:**
   (a) run $sk_m \leftarrow \mathsf{eNCE.Sim}(m, state)$. Set $r_{\mathsf{Gen}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{Gen}}}(pk, sk_m, \mathsf{prg}(\rho_3))$ for random ρ_3
   (b) set r_{Gen} to be the internal state of the receiver.

Figure 3: NCE Simulator. Description of programs Gen:Sim, GenEnc:Sim, and Encrypt:Sim are given in Fig. 4.

Proof. It is clear from the description of the scheme that the transformation preserves the length of communication. We show that it also preserves correctness and security of the underlying eNCE.

Correctness. By the correctness of the underlying eNCE scheme, when generation randomness and encryption randomness are chosen uniformly at random, a decryption error happens with only negligible probability. Since in our scheme both generation randomness r_{NCE} and encryption randomness u are generated using an extracting PRF, their distribution is close to uniform (with overwhelming probability over the choice of PRF keys), and therefore decryption error remains negligible.

Security. We prove in a sequence of hybrids that the real distribution is indistinguishable from the simulated one. To keep the proof concise, we only prove it for one execution.

The proof consists of a repeated application of the following properties of explainability compiler 7:

- The program is indistinguishable from its version with a trapdoor branch (for instance, Gen and Gen:Sim);
- The ciphertext produced by running Encrypt:Sim (which has a trapdoor branch) is indistinguishable from computing a ciphertext using a truly random encryption randomness u (the same holds for output of other two programs);
- the true randomness is indistinguishable from fake randomness explaining given input-output behavior.

These properties allow to make pk, sk, m independent of programs, after which we can use security of underlying eNCE to switch them to simulated pk, sk, c. Now we briefly describe hybrids:

1. **Real execution.** In this distribution the CRS contains obfuscated programs GenEnc and Gen; r_{Gen}, r_{GenEnc}, and r_{Encrypt} are truly random, and pk and c are generated honestly, as specified in Fig. 2.
2. **Hybrid 1.** In this hybrid the CRS contains obfuscated programs GenEnc:Sim and Gen:Sim (Fig. 4).

   Indistinguishability holds by indistinguishability of programs with and without a trapdoor (def. 7 and theorem 5).

Programs generated by NCE Simulator.

```
Program Gen:Sim($r_{\mathsf{Gen}}$)
Inputs: randomness $r_{\mathsf{Gen}}$
// hardcoded values: key for extracting PRF $\mathsf{Ext}_{\mathsf{Gen}}$, faking key $f_{\mathsf{Gen}}$, prg image $S_{\mathsf{Gen}}$
  1. Trapdoor branch:
     (a) Decrypt $\mathsf{out} \leftarrow \mathsf{PDE.Dec}_{f_{\mathsf{Gen}}}(r_{\mathsf{Gen}})$; if $\mathsf{out} = \perp$, then execute normal branch.
     (b) Else parse out as $(pk', sk', s', \tilde{\rho})$. If $\mathsf{prg}(s') = S_{\mathsf{Gen}}$ then output $(pk', sk')$ and halt; otherwise execute the
         normal branch.
  2. Normal branch:
     (a) Set $r_{\mathsf{NCE}} \leftarrow F_{\mathsf{Ext}_{\mathsf{Gen}}}(r_{\mathsf{Gen}})$.
     (b) Set $(pk, sk) \leftarrow \mathsf{eNCE.Gen}(r_{\mathsf{NCE}})$
     (c) Output $(pk, sk)$

Program GenEnc:Sim($\mathbf{r}_{\mathsf{GenEnc}}$)
// hardcoded values: key for extracting PRF $\mathsf{Ext}_{\mathsf{GenEnc}}$, faking key $f_{\mathsf{GenEnc}}$, prg image $S_{\mathsf{GenEnc}}$
Inputs: message $m$, randomness $r_{\mathsf{Enc}}$
  1. Trapdoor branch:
     (a) Decrypt $\mathsf{out} \leftarrow \mathsf{PDE.Dec}_{f_{\mathsf{GenEnc}}}(r_{\mathsf{GenEnc}})$; if $\mathsf{out} = \perp$, then execute normal branch.
     (b) Else parse out as $(E', s', \tilde{\rho})$. If $\mathsf{prg}(s') = S_{\mathsf{GenEnc}}$ then output $(E')$ and halt; otherwise execute the normal
         branch.
  2. Normal branch:
     (a) Set $e \leftarrow \mathsf{F}_{\mathsf{Ext}_{\mathsf{GenEnc}}}(r_{\mathsf{Enc}})$;
     (b) Use $e$ to sample a key $\mathsf{Ext}_{\mathsf{Encrypt}}$ for an extracting PRF and obfuscation randomness $r_{i\mathcal{O}}$
     (c) Compute $E \leftarrow i\mathcal{O}(\mathsf{Encrypt}[\mathsf{Ext}_{\mathsf{Encrypt}}]; r_{i\mathcal{O}})$
     (d) Output $E$.

Program Encrypt:Sim(pk, m, $\mathbf{r}_{\mathsf{Encrypt}}$)
// hardcoded values: key for extracting PRF $\mathsf{Ext}_{\mathsf{Encrypt}}$ , faking key $f_{\mathsf{Encrypt}}$, prg image $S_{\mathsf{Encrypt}}$
Inputs: $\mathsf{eNCE}$ public key $pk$, message $m$, randomness $r_{\mathsf{Encrypt}}$
  1. Trapdoor branch:
     (a) Decrypt $\mathsf{out} \leftarrow \mathsf{PDE.Dec}_{f_{\mathsf{Encrypt}}}(r_{\mathsf{Encrypt}})$; if $\mathsf{out} = \perp$, then execute normal branch.
     (b) Else parse out as $(c', s', \tilde{\rho})$. If $\mathsf{prg}(s') = S_{\mathsf{Encrypt}}$ then output $(c')$ and halt; otherwise execute the normal
         branch.
  2. Normal branch:
     (a) Set $u \leftarrow \mathsf{F}_{\mathsf{Ext}_{\mathsf{Encrypt}}}(r_{\mathsf{Encrypt}})$;
     (b) Compute $c \leftarrow Enc_{pk}(m; u)$
     (c) Output $c$.
```

Figure 4: Programs generated by NCE Simulator. These programs have an additional trapdoor branch which allows the simulator, who knows faking keys f_{GenEnc}, f_{Gen}, f_{Encrypt}, to "explain" any input-output pair of the program, i.e. show randomness consistent with this input-output pair.

3. **Hybrid 2.** In this hybrid upon corruption of parties the adversary is given fake randomness $r_{f,\mathsf{Gen}}$ and $r_{f,\mathsf{GenEnc}}$ (instead of real r_{Gen} and r_{GenEnc}), generated as $r_{f,\mathsf{Gen}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{Gen}}}(pk, sk, \mathsf{prg}(\rho_3))$ and $r_{f,\mathsf{GenEnc}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{GenEnc}}}(E, \mathsf{prg}(\rho_2))$. Here ρ_2, ρ_3 are chosen at random, and E is an obfuscated Encrypt generated as in a real execution.

   Indistinguishability holds by selective explainability (def. 7 and theorem 5). Note that an input to GenEnc (i.e. pk) doesn't depend on m, and Gen doesn't have non-random input at all; thus we can indeed use selective explainability.

4. **Hybrid 3.** In this hybrid we choose r_{NCE} (generation randomness for eNCE) and e (used to generate and obfuscate Encrypt) at random, instead of checking for a trapdoor branch and then computing extracting PRFs $F_{\mathsf{Ext}_{\mathsf{Gen}}}$ and $F_{\mathsf{Ext}_{\mathsf{GenEnc}}}$, as in programs Gen:Sim and GenEnc:Sim.

   Indistinguishability holds by indistinguishability of the source of the output for programs Gen:Sim and GenEnc:Sim (def. 7 and theorem 5).

5. **Hybrid 4.** In this hybrid we generate E as obfuscation of Encrypt:Sim, instead of Encrypt.

   Indistinguishability holds by indistinguishability of programs with and without a trapdoor (def. 7 and theorem 5).

6. **Hybrid 5.** In this hybrid upon corruption of parties the adversary is given fake randomness $r_{f,\mathsf{Encrypt}}$ (instead of real r_{Encrypt}), generated as $r_{f,\mathsf{Encrypt}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{Encrypt}}}(pk, m, c, \mathsf{prg}(\rho_1))$. Here ρ_1 is chosen at random.

   Indistinguishability holds by selective explainability (def. 7 and theorem 5). Note that the program Encrypt needs to be generated only upon corruption, when m is already determined; thus we can indeed use selective explainability.

7. **Hybrid 6.** In this hybrid we choose encryption randomness u at random, instead of checking for a trapdoor branch and then computing extracting PRFs $F_{\mathsf{Ext}_{\mathsf{Encrypt}}}$, as in the program Encrypt:Sim.

   Indistinguishability holds by indistinguishability of the source of the output for the program Encrypt:Sim (def. 7 and theorem 5).

8. **Hybrid 7.** In this hybrid we switch pk, c, sk from real to simulated, relying on security of underlying eNCE. This is a simulation distribution.

The proof for many executions can be carried out in the same way, by first switching the CRS to simulated (hybrid 1) and then switching executions from real to simulated one by one (hybrids 2-7 for each). □

5 Optimal-rate Non-committing Encryption in the CRS Model.

In this section we show how to construct a fully non-committing encryption with rate $1 + o(1)$. A crucial part of our protocol is the underlying seNCE scheme with short ciphertexts, which we will transform into a full NCE in section 5.2.

5.1 Same-public-key Non-committing Encryption with Erasures

In this subsection we present our construction of the same-public-key non-committing encryption with erasures (seNCE for short) (defined in section 2, definition 3), which is a building block in our construction of a full fledged NCE.

Inspired by Sahai and Waters [SW14] way of converting a secret key encryption scheme into a public-key encryption, we set our public key to be an obfuscated encryption algorithm $pk = i\mathcal{O}(\mathsf{Enc}[k])$ (see Figure 5). To allow the simulator

The seNCE Protocol:
Inputs: sender's message m

- **Round 1.** The receiver chooses randomness r_{Gen} and generates keys $(\mathsf{P}_{\mathsf{Enc}}, \mathsf{P}_{\mathsf{Dec}}) \leftarrow \mathsf{Gen}(r_{\mathsf{Gen}})$. It sends $\mathsf{P}_{\mathsf{Enc}}$ to the sender and erases r_{Gen}.
- **Round 2.** The sender chooses randomness r_{Enc} and generates a ciphertext $c \leftarrow \mathsf{P}_{\mathsf{Enc}}(m; r_{\mathsf{Enc}})$. It sends c to the receiver and erases r_{Enc}.
- The receiver decrypts $m' \leftarrow \mathsf{P}_{\mathsf{Dec}}(c)$ and outputs m'.

Program Gen(r)
Inputs: randomness r which consists of three parts $r = (r_1, r_2, r_3)$

1. Set $k \leftarrow r_1$ and generate $\mathsf{P}_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathsf{Enc}[k]; r_2)$ and $\mathsf{P}_{\mathsf{Dec}} \leftarrow i\mathcal{O}(\mathsf{Dec}[k]; r_3)$.
2. Output $(\mathsf{P}_{\mathsf{Enc}}, \mathsf{P}_{\mathsf{Dec}})$

Program Enc[k](m, r) // hardcoded PRF key k
Inputs: message m, randomness r
Program Size: this program is padded to be of maximum size of Enc and Enc:1

1. Set $c_1 \leftarrow \mathsf{prg}(r)$ and $c_2 \leftarrow \mathsf{F}_k(c_1) \oplus m$.
2. output $c = (c_1, c_2)$

Program Dec[k](c) // hardcoded PRF key k
Inputs: ciphertext c consisting of two parts (c_1, c_2)
Program Size: this program is padded to be of maximum size of Dec and SimDec.

1. Output $\mathsf{F}_k(c_1) \oplus c_2$.

Figure 5: seNCE protocol

to generate a fake secret key, we apply the same trick to the secret key: we set the secret key to be an obfuscated decryption algorithm with hardcoded PRF key, namely $sk = i\mathcal{O}(\mathsf{Dec}[k])$. In other words, the seNCE protocol proceeds as follows: the receiver generates the obfuscated programs pk, sk and then erases generation randomness, including the key k. Then it sends pk to the sender; the sender encrypts its message m, erases his encryption randomness, and sends back the resulting ciphertext c, which the receiver decrypts with sk. We present the detailed description of the seNCE protocol in Figure 5.

Theorem 2. *The scheme given on Fig. 5 is the same-public-key non-committing encryption scheme with erasures, assuming indistinguishability obfuscation for circuits and one way functions. In addition, it has ciphertexts (the second message in the protocol) of size* $\mathrm{poly}(\lambda) + |m|$. *The protocol is also perfectly correct.*

Proof. We show that the scheme from Figure 5. is a seNCE and has short ciphertexts.

Perfect correctness. The underlying secret key encryption scheme is perfectly correct, since $\mathsf{Dec}(\mathsf{Enc}(m, r)) = \mathsf{F}_k(c_1) \oplus (\mathsf{F}_k(c_1) \oplus m) = m$. Due to perfect correctness of $i\mathcal{O}$, our seNCE protocol is also perfectly correct.

Security with erasures: We need to show that real and simulated pk, c, sk are indistinguishable, even when the adversary can choose m adaptively after seeing pk.

Now we give the proof for the theorem:

1. **Real experiment.** In this experiment $\mathsf{P}_{\mathsf{Enc}}$ and $\mathsf{P}_{\mathsf{Dec}}$ are generated honestly using Gen, c^* is a ciphertext encrypting m^* with randomness r^*, i.e. $c_1^* = \mathsf{prg}(r^*)$, $c_2^* = F_k(c_1^*) \oplus m^*$.
2. **Hybrid 1.** In this experiment c_1^* is generated at random instead of $\mathsf{prg}(r^*)$.

   Indistinguishability from the previous hybrid follows by security of the PRG.

Simulation:

1. Generate a simulated public key $\mathsf{P}_{\mathsf{Enc}}$ as follows: choose a random PRF key k and randomness r, set $\mathsf{P}_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathsf{Enc}[k]; r)$.
2. Generate a simulated ciphertext $c^* = (c_1^*, c_2^*)$ for random c_1, c_2.
3. Generate a simulated receiver's internal state $\mathsf{P}_{\mathsf{Dec}}$ for message m^* as follows: $\mathsf{P}_{\mathsf{Dec}} \leftarrow i\mathcal{O}(\mathsf{SimDec}[k, c^*, m^*])$.

Program SimDec[k, c*, m*](c) // hardcoded PRF key k, dummy ciphertext c^*, challenge message m^*
Inputs: ciphertext c which consists of two parts (c_1, c_2)
Program Size: this program is padded to be of maximum size of Dec and SimDec.

1. If $c_1 = c_1^*$, output $c_2^* \oplus c_2 \oplus m^*$. Otherwise, output $\mathsf{F}_k(c_1) \oplus c_2$.

Figure 6: seNCE Simulator.

Program Enc:1$[\mathbf{k}\{\mathbf{c}_1^*\}](\mathbf{m}, \mathbf{r})$ // hardcoded punctured PRF key $k\{c_1^*\}$
Inputs: message m, randomness r
Program Size: this program is padded to be of maximum size of Enc and Enc:1

(a) Set $c_1 \leftarrow \mathsf{prg}(r)$ and $c_2 \leftarrow \mathsf{F}_{k\{c_1^*\}}(c_1) \oplus m$.
(b) Output $c = (c_1, c_2)$.

Figure 7: Program Enc:1 used in the proof.

3. **Hybrid 2.** In this experiment we puncture key k in both programs Enc and Dec, namely, we obfuscate programs $\mathsf{P}_{\mathsf{Enc}} = i\mathcal{O}(\mathsf{Enc{:}1}[k\{c_1^*\}])$, $\mathsf{P}_{\mathsf{Dec}} = i\mathcal{O}(\mathsf{SimDec}\ [k\{c_1^*\}, c^*, m^*])$. We claim that functionality of these programs is the same as that of Enc and Dec:

   Indeed, in Enc:1 (defined in Figure 7), c_1^* is random and thus with high probability it is outside the image of the PRG; therefore no input r results in evaluating F at the punctured point c_1^*, and we can puncture safely. In SimDec (defined in Figure 6), if $c_1 \neq c_1^*$, then the program behaves exactly like the original one (i.e. computes $F_k(c_1) \oplus c_2$); if $c_1 = c_1^*$, then SimDec outputs $c_2^* \oplus c_2 \oplus m = (F_k(c_1^*) \oplus m) \oplus c_2 \oplus m = F_k(c_1^*) \oplus c_2$, which is exactly what Dec outputs when $c_1 = c_1^*$. Note that c_1^* is random (and thus independent of m), therefore $\mathsf{pk} = \mathsf{Enc{:}1}[k\{c_1^*\}]$ can be generated *before* the message m^* is fixed.

   Indistinguishability from the previous hybrid follows by the security of $i\mathcal{O}$.

4. **Hybrid 3.** In this hybrid we switch c_2^* from $\mathsf{F}_k(c_1^*) \oplus m^*$ to random. This hybrid relies on the indistinguishability between punctured value $F_k(c_1^*)$ and a truly random value, even given a punctured key $k\{c_1^*\}$.

   Indeed, to reconstruct this hybrid, first choose random c_1^* and get $k\{c_1^*\}$ and val^* (which is either random or $F_k(c_1^*)$) from the PPRF challenger. Show obfuscated $\mathsf{Enc} : 1[k\{c_1^*\}]$ as a public key. When the adversary fixes message m^*, set $c_2^* = val^* \oplus m^*$ and upon corruption show obfuscated $\mathsf{SimDec}[k\{c_1^*\}, c^*, m^*]$. If val^* was truly random, then $c_2^* = val^* \oplus m^*$ is distributed uniformly and thus we are in hybrid 3. If val^* is the actual PRF value, then $c_2^* = F_k(c_1^*) \oplus m^*$ and we are in hybrid 2.

   Indistinguishability holds by security of a punctured PRF.

5. **Hybrid 4 (Simulation).** In this hybrid we unpuncture the key k in both programs and show $\mathsf{P}_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathsf{Enc}[k])$, $\mathsf{P}_{\mathsf{Dec}} \leftarrow i\mathcal{O}(\mathsf{SimDec}[k, c^*, m^*])$.

   This is without changing the functionality of the programs: Indeed, in Enc no random input r results in $\mathsf{prg}(r) = c_1^*$, thus we can remove the puncturing. In Dec:1 due to preceding "if" no input c causes evaluation of $F_{k\{c_1^*\}}$, thus we can unpuncture it as well.

   The indistinguishability from the previous hybrid follows by the security if the $i\mathcal{O}$.

We observe that the last hybrid is indeed the simulation experiment described in Figure 6: c^* is a simulated ciphertext since c_1^* is random, $c_2^* = \mathsf{F}_k(c_1^*)$, $\mathsf{P}_{\mathsf{Enc}}$ is honestly generated, and $\mathsf{P}_{\mathsf{Dec}}$ is a simulated key $\mathsf{SimDec}[k, c^*, m^*]$, which decrypts c^* to m^*. Thus, we have shown that this scheme is non-committing with erasures.

The same public key. Both real generation algorithm Gen and the simulator on randomness $r_{\mathsf{Gen}} = (r_1, r_2, r_3)$ produce exactly the same public key $pk = i\mathcal{O}(Enc[r_1]; r_2)$.

Efficiency: Our PRG should be length-doubling to ensure that its image is sparse. Thus $|c_1| = 2\lambda$, and $|c_2| = |m|$. Thus the size of our ciphertext is $2\lambda + |m|$.[7]

□

5.2 From seNCE to full NCE

In this subsection we show how to transform any seNCE (for instance, seNCE constructed in Section 5.1) into full non-committing encryption in the CRS model. We start with a brief overview of the construction:

Construction. Our CRS contains algorithms GenEnc and GenDec which share master secret key MSK. Both programs can internally generate the parameters for the underlying seNCE scheme using their MSK and then output an encryption program or a decryption key. More specifically, GenDec takes a random input, produces generation token t and then uses this token and MSK to generate randomness r_{NCE} for seNCE.Gen. Then the program samples seNCE keys pk, sk from r_{NCE}. It outputs the token t and the generated decryption key sk for a seNCE scheme. The receiver keeps sk for itself and sends the token t to the sender.

GenEnc, given a token t, can produce (the same) pair (pk, sk) and outputs an algorithm Enc_{pk}, which has pk hardwired. This algorithm takes a message m and outputs its encryption c, which the sender sends back to the receiver. Then receiver decrypts it using sk.

All three programs GenEnc, GenDec, Enc are augmented with sparse computationally extracting PRFs, which are used to obtain new randomness by hashing down all inputs to the program.

We present our full NCE protocol and its building block functions GenEnc, GenDec, Enc in Figure 8.

Theorem 3. *Assuming indistinguishability obfuscation for circuits, one way functions, and seNCE with a ciphertext size $O(\mathrm{poly}(\lambda)) + |m|$, the described construction is a constant-rate non-committing public key encryption scheme in a common reference string model. Assuming perfect correctness of underlying* seNCE*, our NCE scheme is also perfectly correct.*

6 Proof of security of theorem 2

6.1 Proof overview

Proof. We first show correctness of the scheme. Next we present a simulator and argue that the scheme is secure. Finally we argue that the scheme is constant-rate.

[7] In fact, the size of our public key is $\mathrm{poly}(\lambda)$, since so is the size of a key k punctured at c_1 and later obfuscated. If we could also simulate sender internal state, then we could obfuscate generation algorithm Gen, put it into a CRS, and the scheme "send $\mathsf{P}_{\mathsf{Enc}}$, send c" would be fully non-committing and constant rate. However, we don't know how to make sender state simulatable without making the size of $\mathsf{P}_{\mathsf{Enc}}$ at least $\lambda|m|$, which already has rate at least λ.

```
The NCE Protocol
CRS: obfuscated programs $\mathsf{P}_{\mathsf{GenEnc}} = i\mathcal{O}(\mathsf{GenEnc}), \mathsf{P}_{\mathsf{GenDec}} = i\mathcal{O}(\mathsf{GenDec})$
Inputs: sender's message $m$
  1. Round 1. The receiver chooses randomness $r_{\mathsf{GenDec}}$ and generates $(t, sk) \leftarrow \mathsf{P}_{\mathsf{GenDec}}(r_{\mathsf{GenDec}})$. The receiver
     sends $t$ to the sender.
  2. Round 2. The sender chooses randomness $r_{\mathsf{GenEnc}}$ and generates $\mathsf{P}_{\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{GenEnc}}(t, r_{\mathsf{GenEnc}})$. Then the sender
     chooses randomness $r_{\mathsf{Enc}}$ and encrypts $c \leftarrow \mathsf{P}_{\mathsf{Enc}}(m, r_{\mathsf{Enc}})$. The sender sends $c$ to the receiver.
  3. The receiver decrypts $m' \leftarrow \mathsf{seNCE.Dec}_{sk}(c)$ and outputs $m'$

Program GenEnc[MSK, $\mathsf{Ext}_{\mathsf{GenEnc}}$](t, $r_{\mathsf{GenEnc}}$)
// hardcoded values: master key MSK, key for sparse computationally extracting PRF $\mathsf{Ext}_{\mathsf{GenEnc}}$
Inputs: token $t$, randomness $r_{\mathsf{GenEnc}}$
  1. Set the randomness $r_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t)$, run $(pk, sk) \leftarrow \mathsf{seNCE.Gen}(r_{\mathsf{NCE}})$.
  2. Set the randomness $e \leftarrow \mathsf{F}_{\mathsf{Ext}_{\mathsf{GenEnc}}}(t, r_{\mathsf{GenEnc}})$, sample $\mathsf{Ext}_{\mathsf{Enc}}, r_{i\mathcal{O},\mathsf{Enc}} \leftarrow e$.
  3. Generate $\mathsf{P}_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathsf{Enc}[pk, \mathsf{Ext}_{\mathsf{Enc}}]; r_{i\mathcal{O},\mathsf{Enc}})$.
  4. Output the program $\mathsf{P}_{\mathsf{Enc}}$.

Program Enc[pk, $\mathsf{Ext}_{\mathsf{Enc}}$](m, $r_{\mathsf{Enc}}$)
// hardcoded values: seNCE public key pk, key for sparse computationally extracting PRF $\mathsf{Ext}_{\mathsf{Enc}}$
Inputs: message $m$, randomness $r_{\mathsf{Enc}}$
  1. Generate encryption randomness $u \leftarrow \mathsf{F}_{\mathsf{Ext}_{\mathsf{Enc}}}(m, r_{\mathsf{Enc}})$.
  2. Compute ciphertext $c \leftarrow \mathsf{seNCE.Enc}_{pk}(m; u)$.
  3. Output $c$.

Program GenDec[MSK, $\mathsf{Ext}_{\mathsf{GenDec}}$]($r_{\mathsf{GenDec}}$)
// hardcoded values: master key MSK , key for sparse extracting PRF $\mathsf{Ext}_{\mathsf{GenDec}}$,
Inputs: randomness $r_{\mathsf{GenDec}}$
  1. Generate token $t \leftarrow \mathsf{F}_{\mathsf{Ext}_{\mathsf{GenDec}}}(r_{\mathsf{GenDec}})$.
  2. Set the randomness $r_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t)$, run $(pk, sk) \leftarrow \mathsf{seNCE.Gen}(r_{\mathsf{NCE}})$.
  3. Output $(t, sk)$.
```

Figure 8: The NCE Protocol. The description of our NCE protocol. We write r_{Enc}, r_{GenEnc}, r_{GenDec} to denote randomness used in these programs. $\mathsf{Ext}_{\mathsf{Enc}}$, $\mathsf{Ext}_{\mathsf{GenEnc}}$, and $\mathsf{Ext}_{\mathsf{GenDec}}$ are keys for sparse computationally extracting PRF, which allows to "garble" randomness before using it in the program.

Correctness. The presented scheme is perfectly correct, as long as the underlying seNCE is perfectly correct: First, due to perfect correctness of $i\mathcal{O}$, using obfuscated programs GenEnc, GenDec, Enc is as good as using corresponding non-obfuscated programs. Next, both the sender and receiver generate public and secret seNCE keys as $(pk, sk) \leftarrow \mathsf{seNCE.Gen}(F_{\mathsf{MSK}}(t))$. The sender also generates c, which is an encryption of m under pk, which is decrypted under sk by receiver. Thus the scheme is as correct as the underlying seNCE scheme is.

Since the protocol for seNCE which we give in section 5.1 has perfect correctness, the overall NCE scheme, when instantiated with our seNCE protocol from section 5.1, also achieves perfect correctness.

6.1.1 Description of the simulator

In this subsection we first explain which variables the adversary sees and then describe our simulator.

The view of the adversary. The view of the adversary consists of the CRS (programs $\mathsf{P}^*_{\mathsf{GenEnc}}$, $\mathsf{P}^*_{\mathsf{GenDec}}$), as well as communications and internal state of several protocol instances. Namely, for each protocol instance the adversary sees the following variables:

1. The first protocol message t^*, after which the adversary assigns an input m for this protocol instance;
2. The second protocol message c^*;
3. The sender internal state $r^*_{\mathsf{Enc}}, r^*_{\mathsf{GenEnc}}$;
4. The receiver internal states r^*_{GenDec}.

Other values, such as $\mathsf{P}^*_{\mathsf{Enc}}$ and sk^*, can be obtained by the adversary by running programs in the CRS: $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathsf{P}^*_{\mathsf{GenEnc}}(t^*, r^*_{\mathsf{GenEnc}})$, $(sk^*, t^*) \leftarrow \mathsf{P}^*_{\mathsf{GenDec}}(r^*_{\mathsf{GenDec}})$.

Simulation. The simulator sets a simulated CRS to be a description of programs $\mathsf{GenEnc{:}Sim}$, $\mathsf{GenDec{:}Sim}$ (Figure 10). The difference from real-world programs is that these simulated programs have a trapdoor branch inside them, which allows the simulator to produce randomness such that a program outputs a desired output on this randomness. These programs are trapdoor versions of programs $\mathsf{GenEnc{:}clean}$, $\mathsf{GenDec{:}clean}$ (Figure 11).

- **CRS generation.** The simulator generates CRS programs GenEnc, GenDec, including keys f for trapdoor branch.

  Next the simulator responds to requests of the adversary. The adversary can interactively ask to setup a new execution of the protocol (where the input m can be chosen based on what the adversary has already learn from other executions), or ask to deliver messages or corrupt parties in protocols which are already being executed. Below we describe what our simulator does in each case:

- **Simulation of the first message.** If the receiver is already corrupted, then the simulator generates the first message by choosing random r^*_{GenDec} and running $(t^*, sk^*) \leftarrow \mathsf{GenDec{:}Sim}(r^*_{\mathsf{GenDec}})$. Otherwise the simulator chooses random t^* as the first message.

- **Simulation of the second message.** If either the sender or the receiver is already corrupted, then the simulator learns m and therefore can generate the second message honestly. If neither the sender nor the receiver in this execution are corrupted by this moment, the simulator runs $(pk^*_f, c^*_f) \leftarrow \mathsf{seNCE.Sim}(\mathsf{F}_{\mathsf{MSK}}(t^*))$ and gives c^*_f to the adversary as the second message.

- **Simulation of the sender internal state.** If either the sender or the receiver had been corrupted before the second message was sent, then the simulator has generated the second message honestly and can thus show true sender randomness.

  Otherwise it first generates an obfuscation $\mathsf{P}^*_{\mathsf{Enc}}$ of the program $\mathsf{Enc{:}Sim}$ with simulated pk^*_f hardwired inside (Figure 10). Next it encodes m^*, c^*_f into sender encryption randomness, i.e. sets $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{Enc}}}(m^*, c^*_f, s^*_{\mathsf{Enc}}, \mathsf{prg}(\rho_3))$ for random ρ_3; so that P^*_{Enc} on input $(m^*, r^*_{f,\mathsf{Enc}})$ outputs c^*_f.

  Finally, it encodes $\mathsf{P}^*_{\mathsf{Enc}}$ into $r^*_{f,\mathsf{GenEnc}}$, i.e. sets the sender's generation randomness $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{GenEnc}}}(t^*, \mathsf{P}^*_{\mathsf{Enc}}, s^*_{\mathsf{GenEnc}}, \mathsf{prg}(\rho_2))$ for random ρ_2, so that $\mathsf{P}^*_{\mathsf{GenEnc}}$ outputs $\mathsf{P}^*_{\mathsf{Enc}}$ on input $(t^*, r^*_{f,\mathsf{GenEnc}})$.

  The pair $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ is set to be the sender internal state.

- **Simulation of the receiver internal state.** If the corruption happens before the first message is sent, then the simulator has generated the first message honestly and thus can show true receiver internal state.

  If corruption happens after the first message, but before the second, then the simulator has shown random t^* as the first message. It computes $sk^* \leftarrow \mathsf{seNCE.Gen}\ (F_{\mathsf{MSK}}(t^*))$. It encodes (t^*, sk^*) into receiver randomness, i.e. sets $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{GenDec}}}(t^*, sk^*, s^*_{\mathsf{GenDec}}, \mathsf{prg}(\rho_1))$ for random ρ_1, so that P^*_{GenDec} on input $r^*_{f,\mathsf{GenDec}}$ outputs (t^*, sk^*).

  If corruption happens after the second message, then the simulator runs seNCE simulator and gets fake secret key sk^*_f which decrypts dummy c^*_f to m^*, chosen by the adversary. Next it encodes (t^*, sk^*_f) into receiver randomness, i.e. sets $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{GenDec}}}(t^*, sk^*_f, s^*_{\mathsf{GenDec}}, \mathsf{prg}(\rho_1))$ for random ρ_1, so that P^*_{GenDec} on input $r^*_{f,\mathsf{GenDec}}$ outputs (t^*, sk^*_f).

Note that simulation of each protocol instance is independent of simulation of other protocol instances (except for the fact that they share the same CRS). Therefore in order to keep the description of the simulator simple enough, in Figure 9 we present a detailed description of the simulator for a single execution only; it can be trivially generalized to a multiple-execution case according to what is written above. In addition, the simulator is presented for a difficult case, i.e. when nobody is corrupted by the time the ciphertext is sent, and therefore the simulator has to present a dummy c and later open it to a correct m.

Next we outline the intuition for the security proof and after that provide the detailed description of hybrids.

Simulation

1. **Generate a CRS:**
   (a) Sample keys $\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}$. Also choose random $s^*_{\mathsf{GenEnc}}, s^*_{\mathsf{GenDec}}, s^*_{\mathsf{Enc}}$ and set $S^*_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{GenEnc}})$, $S^*_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s^*_{\mathsf{GenDec}})$, $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.
   (b) Generate obfuscations $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathsf{GenEnc{:}Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S^*_{\mathsf{GenEnc}}])$, $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathsf{GenDec{:}Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S^*_{\mathsf{GenDec}}])$.
   (c) Set the CRS to be $(\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) Choose a random t^* and generate $r^*_{\mathsf{NCE}} \leftarrow F_{\mathsf{MSK}}(t^*)$.
   (b) Run the seNCE simulator $(c^*_f, st) \leftarrow \mathsf{seNCE.Sim}(r^*_{\mathsf{NCE}})$ to generate a simulated ciphertext and state for a future opening.
   (c) Show (t^*, c^*_f) as communications in the protocol.
3. **Generate parties' internal state consistent with message m^* and communications:**
   (a) Run the seNCE simulator to create a simulated secret key: $sk^*_f \leftarrow \mathsf{seNCE.Sim}(st, m^*)$
   (b) Set the receiver's randomness $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{GenDec}}}(t^*, sk^*_f, s^*_{\mathsf{GenDec}}, \mathsf{prg}(\rho_1))$ for random ρ_1.
   (c) Sample keys $f_{\mathsf{Enc}}, \mathsf{Ext}_{\mathsf{Enc}}$ and generate $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathsf{Enc{:}Sim}[f_{\mathsf{Enc}}, \mathsf{Ext}_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}])$;
   (d) Set the sender's generation randomness $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{GenEnc}}}(t^*, \mathsf{P}^*_{\mathsf{Enc}}, s^*_{\mathsf{GenEnc}}, \mathsf{prg}(\rho_2))$ for random ρ_2.
   (e) Set the sender's encryption randomness $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{PDE.Enc}_{f_{\mathsf{Enc}}}(m^*, c^*_f, s^*_{\mathsf{Enc}}, \mathsf{prg}(\rho_3))$ for random ρ_3.
   (f) Show $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ as the sender's internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver's internal state.

Figure 9: Simulation.

6.1.2 Overview of the analysis of the simulator

Before presenting hybrids, let us give a roadmap of the proof: Starting from the real execution, we first add trapdoor branches to the programs in the CRS, in order to allow creating fake randomness which explains a given output. Next we change what the simulator shows as internal states of the parties: instead of showing their real randomness, the simulator shows fake randomness (which explains outputs of programs from a real execution, i.e. this randomness explains honestly generated sk^*, c^*, and P^*_{Enc}). Our next step is to puncture the key $\mathsf{MSK}\{t^*\}$ in both CRS programs. This allows us to switch seNCE generation randomness r^*_{NCE} from $\mathsf{F}_{\mathsf{MSK}}(t^*)$ to a random value; this means that seNCE parameters (pk^*, sk^*) are now freshly generated and do not depend on the rest of an experiment anymore. Therefore we can use security of seNCE and switch seNCE values (pk^*, c^*, sk^*) from real to simulated (in particular, the simulator hardwires these simulated c^*_f, sk^*_f into fake randomness, instead of hardwiring real-execution c^*, sk^*). Next we undo previous hybrids: we set r^*_{NCE} as the result of $\mathsf{F}_{\mathsf{MSK}}(t^*)$, and then unpuncture $\mathsf{MSK}\{t^*\}$ in both CRS programs.

We now briefly describe each hybrid. The full description with detailed security reductions is given in section 6.2.

- **Hybrid 0.** We start with a real execution of the protocol.
- **Hybrids 1a-1b.** We change CRS programs to be obfuscations of $\mathsf{GenEnc{:}Sim}$ and $\mathsf{GenDec{:}Sim}$, instead of

Programs used in a simulation.

Program GenEnc:Sim[MSK, $\mathbf{f}_{\mathsf{GenEnc}}$, $\mathbf{Ext}_{\mathsf{GenEnc}}$, $\mathbf{S}_{\mathsf{GenEnc}}$](t, $\mathbf{r}_{\mathsf{GenEnc}}$)

// hardcoded values: master key MSK, faking key f_{GenEnc}, key for sparse extracting PRF $\mathsf{Ext}_{\mathsf{GenEnc}}$, prg image S_{GenEnc}

Inputs: token t, randomness r_{GenEnc}

```
1. Trapdoor branch:
    (a) Decode $\mathsf{out} \leftarrow \mathsf{PDE.Dec}_{f_{\mathsf{GenEnc}}}(r_{\mathsf{GenEnc}})$; if $\mathsf{out} = \perp$, then execute the normal branch.
    (b) Else parse out as $(t', \mathsf{P}', s', \tilde{\rho})$. If $t' = t$ and $\mathsf{prg}(s') = S_{\mathsf{GenEnc}}$, then output $\mathsf{P}'$ and halt. Otherwise, execute the normal branch.
2. Normal branch:
    (a) Set the randomness $r_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t)$, run $(pk, sk) \leftarrow \mathsf{seNCE.Gen}(r_{\mathsf{NCE}})$.
    (b) Set the randomness $e \leftarrow \mathsf{F}_{\mathsf{Ext}_{\mathsf{GenEnc}}}(t, r_{\mathsf{GenEnc}})$, sample $\mathsf{Ext}_{\mathsf{Enc}}, r_{i\mathcal{O},\mathsf{Enc}} \leftarrow e$.
    (c) Generate $\mathsf{P}_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathsf{Enc}[pk, \mathsf{Ext}_{\mathsf{Enc}}]; r_{i\mathcal{O},\mathsf{Enc}})$.
    (d) Output the program $\mathsf{P}_{\mathsf{Enc}}$.
```

Program Enc:Sim[pk, $\mathbf{f}_{\mathsf{Enc}}$, $\mathbf{Ext}_{\mathsf{Enc}}$, $\mathbf{S}_{\mathsf{Enc}}$](m, $\mathbf{r}_{\mathsf{Enc}}$)

// hardcoded values: seNCE public key pk, sender faking key f_{Enc}, key for sparse extracting PRF $\mathsf{Ext}_{\mathsf{Enc}}$, prg image S_{Enc}

Inputs: message m, randomness r_{Enc}

```
1. Trapdoor branch:
    (a) Decrypt $\mathsf{out} \leftarrow \mathsf{PDE.Dec}_{f_{\mathsf{Enc}}}(r_{\mathsf{Enc}})$; if $\mathsf{out} = \perp$, then execute the normal branch.
    (b) Else parse out as $(m', c', s', \tilde{\rho})$. If $m' = m$ and $\mathsf{prg}(s') = S_{\mathsf{Enc}}$, then output $c'$ and halt. Otherwise, execute the normal branch.
2. Normal branch:
    (a) Generate encryption randomness $u \leftarrow \mathsf{F}_{\mathsf{Ext}_{\mathsf{Enc}}}(m, r_{\mathsf{Enc}})$.
    (b) Compute ciphertext $c \leftarrow \mathsf{seNCE.Enc}_{pk}(m; u)$.
    (c) Output $c$.
```

Program GenDec:Sim[MSK, $\mathbf{f}_{\mathsf{GenDec}}$, $\mathbf{Ext}_{\mathsf{GenDec}}$, $\mathbf{S}_{\mathsf{GenDec}}$]($\mathbf{r}_{\mathsf{GenDec}}$)

// hardcoded values: master key MSK , receiver faking key f_{GenDec}, key for sparse extracting PRF $\mathsf{Ext}_{\mathsf{GenDec}}$, prg image S_{GenDec}

Inputs: randomness r_{GenDec}

```
1. Trapdoor branch:
    (a) Decrypt $\mathsf{out} \leftarrow \mathsf{PDE.Dec}_{f_{\mathsf{GenDec}}}(r_{\mathsf{GenDec}})$; if $\mathsf{out} = \perp$, then execute normal branch.
    (b) Else parse out as $(t', sk', s', \tilde{\rho})$. If $\mathsf{prg}(s') = S_{\mathsf{GenDec}}$ then output $(t', sk')$ and halt; otherwise execute the normal branch.
2. Normal branch:
    (a) Generate token $t \leftarrow \mathsf{F}_{\mathsf{Ext}_{\mathsf{GenDec}}}(r_{\mathsf{GenDec}})$.
    (b) Set the randomness $r_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t)$, run $(pk, sk) \leftarrow \mathsf{seNCE.Gen}(r_{\mathsf{NCE}})$.
    (c) Output $(t, sk)$.
```

Figure 10: Programs used in a simulation. Note that GenEnc:Sim still produces normal program Enc, not Enc:Sim.

GenEnc and GenDec; in other words, we add trapdoor branch to CRS programs. Security holds by indistinguishability of programs with and without trapdoor branch (theorem 5).

Next for every execution i, in which the receiver is corrupted between the first and the second messages are sent, we run hybrids $2_i - 3_i$.

- **Hybrid 2$_i$.** Instead of showing the real r^*_{GenDec}, the simulator shows fake $r^*_{f,\mathsf{GenDec}}$, which encodes t^*, sk^*. These experiments are indistinguishable because of the indistinguishability of explanation: indeed, GenDec on both inputs r^*_{GenDec} and $r^*_{f,\mathsf{GenDec}}$ outputs t^*, sk^*, therefore true randomness r^*_{GenDec} is indistinguishable from randomness $r^*_{f,\mathsf{GenDec}}$, which explains the output t^*, sk^* on empty non-random input.

  Note that since there is no non-random input to our program $\mathsf{P}_{\mathsf{GenDec}}$, it is enough to use the selective indistinguishability of explanation.

- **Hybrid 3$_i$.** We choose t^* at random and then compute sk^* as $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(F_{\mathsf{MSK}}(t^*))$. In other words, when generating t^* (and therefore sk^*) we skip trapdoor branch and applying extracting PRF F_{Ext} in GenDec. Indistinguishability holds by indistinguishability of a source of the output for programs GenDec:Sim, GenDec:clean and output (t^*, sk^*).

  This is the simulation for the case when the receiver is corrupted between the first and the second message.

For every execution i, in which both corruptions happen after the second message is sent, we run hybrids $2_i - 5h_i$.

- **Hybrid 2$_i$.** Instead of showing the real r^*_{GenEnc}, the simulator shows fake $r^*_{f,\mathsf{GenEnc}}$, which encodes $t^*, \mathsf{P}^*_{\mathsf{Enc}}$. These experiments are indistinguishable because of the indistinguishability of explanation (theorem 5): indeed, GenEnc on both inputs $t^*, r^*_{\mathsf{GenEnc}}$ and $t^*, r^*_{f,\mathsf{GenEnc}}$ outputs P^*_{Enc}, and by the theorem true randomness r^*_{GenEnc} is indistinguishable from fake randomness $r^*_{f,\mathsf{GenEnc}}$ which explains P^*_{Enc} on input t^*. Note that non-random input to our program $\mathsf{P}_{\mathsf{GenEnc}}$ is t^*, obtained as $t^* \leftarrow F_{\mathsf{Ext}_{\mathsf{GenDec}}}(r^*_{\mathsf{GenDec}})$ for random r^*_{GenDec}, i.e., it can be generated before a CRS is shown to the adversary. Thus it is enough to use the selective indistinguishability of explanation (theorem 5).

- **Hybrid 3$_i$.** In the next step instead of showing the real r^*_{GenDec}, the simulator shows fake $r^*_{f,\mathsf{GenDec}}$, which encodes t^*, sk^*. These experiments are indistinguishable because of the indistinguishability of explanation: indeed, GenDec on both inputs r^*_{GenDec} and $r^*_{f,\mathsf{GenDec}}$ outputs t^*, sk^*, therefore true randomness r^*_{GenDec} is indistinguishable from randomness $r^*_{f,\mathsf{GenDec}}$, which explains the output t^*, sk^* on empty non-random input.

  Note that since there is no non-random input to our program $\mathsf{P}_{\mathsf{GenDec}}$, it is enough to use the selective indistinguishability of explanation.

- **Step 4$_i$.** Next global step is to switch random r^*_{Enc} to fake $r^*_{f,\mathsf{Enc}}$ which encodes (m^*, c^*). We do this in several steps:

  - **Hybrid 4a$_i$.** We obtain t^*, sk^* by running GenDec:clean on r^*_{GenDec} instead of running GenDec:Sim. In other words, when generating t^* (and therefore sk^*) we skip trapdoor branch in GenDec; in addition, we choose t^* at random instead of computing $\mathsf{F}_{\mathsf{ExtGenDec}}$ on r^*_{GenDec}. Indistinguishability holds by indistinguishability of a source of the output for programs GenDec:Sim, GenDec:clean and output (t^*, sk^*).

  - **Hybrid 4b$_i$.** We generate $\mathsf{P}^*_{\mathsf{Enc}}$ by running GenEnc:clean on t^* and random e^*, instead of running $\mathsf{GenEnc{:}Sim}(t^*, r^*_{\mathsf{GenEnc}})$. In other words, when generating P^*_{Enc} we skip trapdoor branch in GenEnc and use random e^*, instead of choosing e^* as a result of $\mathsf{F}_{\mathsf{Ext}_{\mathsf{GenEnc}}}(t^*, r^*_{\mathsf{GenEnc}})$ (recall that e^* is randomness used to create key $\mathsf{Ext}_{\mathsf{Enc}}$, put it on top of $\mathsf{seNCE.Enc}_{pk}()$ and obfuscate the whole program $\mathsf{P}^*_{\mathsf{Enc}} = i\mathcal{O}(\mathsf{Enc})$). In other words, we take generated pk^*, choose extractor keys and obfuscation randomness at random and generate $i\mathcal{O}(\mathsf{Enc})$. Later in the experiment we encode $\mathsf{P}^*_{\mathsf{Enc}}$ into randomness $r^*_{f,\mathsf{GenEnc}}$. Security holds by indistinguishability of source of the output (theorem 5) for program GenEnc.

– **Hybrid 4c$_i$.** We generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of program Enc:Sim instead of Enc. In other words, we add trapdoor branch to $\mathsf{P}^*_{\mathsf{Enc}}$; keys for these trapdoor branch are chosen at random. Security holds by indistinguishability of programs with and without trapdoor branch for program Enc (theorem 5).

– **Hybrid 4d$_i$.** In this step we finally change r^*_{Enc} to $r^*_{f,\mathsf{Enc}}$ as follows: we first create a CRS and give it to the adversary. Then we generate random t^* and show t^* to the adversary as the first message in the protocol. Next the adversary fixes an input m^*. Then we generate pk^*, sk^* as $\mathsf{seNCE.Gen}(F_{\mathsf{MSK}}(t^*))$ and give $\mathsf{Enc.clean}() = \mathsf{seNCE.Enc}_{pk^*}()$ to the explainability challenger as the underlying program. The challenger chooses random e^*, samples keys $\mathsf{Ext}_{\mathsf{Enc}}$, f_{Enc}, S_{Enc} and gives us either $(r^*_{\mathsf{Enc}}, m^*, c^*, \mathsf{P}^*_{\mathsf{Enc}})$ or $(r^*_{f,\mathsf{Enc}}, m^*, c^*, \mathsf{P}^*_{\mathsf{Enc}})$, where r^*_{Enc} is random, $\mathsf{P}^*_{\mathsf{Enc}} = i\mathcal{O}(\mathsf{Enc.Sim})$, $c^* = \mathsf{P}^*_{\mathsf{Enc}}$, and $r^*_{f,\mathsf{Enc}}$ encodes $m^*, c^*, s_{\mathsf{Enc}}$. We show the given c^* as the second message in the protocol. Once asked to open the internal state, we present the given r^*_{Enc} or $r^*_{f,\mathsf{Enc}}$, generate r^*_{GenEnc} explaining the given $\mathsf{P}^*_{\mathsf{Enc}}$, and generate r^*_{GenDec} explaining (t^*, sk^*).

  We can rely on the selective indistinguishability of explanation for program Enc:Sim (5) since at the moment when we need to see the challenge in explanation game (i.e., when we need to show c^* to the adversary), $\mathsf{P}^*_{\mathsf{Enc}}$'s input m^* is already fixed.

• **Step 5$_i$.** Our next global step is to change the underlying seNCE values to simulated. We proceed in several steps:

– **Hybrids 5a$_i$-5b$_i$.** We puncture MSK at t^*. In GenDec we can puncture immediately, since due to the sparseness of $\mathsf{F}_{\mathsf{Ext}_{\mathsf{GenDec}}}$ no input r results in t^*. In GenEnc we hardwire pk^* and use it whenever $t = t^*$; otherwise, we use the punctured key $\mathsf{MSK}\{t^*\}$ to generate r_{NCE} and then sample pk.

– **Hybrid 5c$_i$.** Once $\mathsf{MSK}\{t^*\}$ is punctured, we can choose the generation randomness for underlying seNCE scheme r^*_{NCE} at random.

– **Hybrids 5d$_i$.** We generate c^* as a result of running Enc.clean on m^* and random u^* instead of running $\mathsf{P}^*_{\mathsf{Enc}}(m^*, r^*_{\mathsf{Enc}})$. In other words, when generating c^* we skip trapdoor branch and use fresh randomness instead of using PRF $F_{\mathsf{Ext}_{\mathsf{Enc}}}$. We rely on indistinguishability of the source of the output for programs Enc:Sim, Enc:clean and output c^*.

– **Hybrid 5e$_i$.** Next we switch the seNCE values from real to simulated: namely, c^*_f is now simulated and sk^*_f is now a simulated key decrypting c^*_f to m^*. We rely on the security of the underlying seNCE. Here we crucially use the fact that in the underlying NCE scheme pk^* is shown *before* the adversary chooses a message, since we hardwire this pk^* into the CRS (in GenEnc).

– **Hybrid 5f$_i$.** We switch back r^*_{NCE} to be the result of $\mathsf{F}_{\mathsf{MSK}}(t^*)$.

– **Hybrid 5g$_i$-5h$_i$.** We unpuncture $\mathsf{MSK}\{t^*\}$ in GenEnc and GenDec and remove the hardwired pk^* from GenEnc. To remove hardwired pk^*, we crucially use the fact that pk^*, although simulated, is the same as real pk^*, generated from randomness $F_{MSK}(t^*)$, which is guaranteed by the same-public-key property of seNCE.

6.1.3 Sizes in our construction

Our construction has a lot of size dependencies. We present a size diagram on Figure 12. There all sizes are grouped in "complexity classes". Here we outline several main dependencies:

• if a fake randomness has values encoded, it should be longer than these values, but not much longer. Namely, if underlying encoded message has size l, then the size of the plaintext for PDE (which consists of encoded message, secret s and $\mathsf{prg}(\rho)$) is $l + 3\lambda$, and the size of PDE ciphertext should be at least 4 times bigger (the latter is because explainability compiler uses statistically injective PRF). Therefore randomness and encoded value are in the same "complexity class".

Programs used in hybrids

Program GenEnc:clean[MSK](t; e)

```
// hardcoded values: master key MSK
Inputs: token $t$, randomness $e$
  1. Set the randomness $r_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t)$, run $(pk, sk) \leftarrow \mathsf{seNCE.Gen}(r_{\mathsf{NCE}})$.
  2. Sample $\mathsf{Ext}_{\mathsf{Enc}}, r_{i\mathcal{O},\mathsf{Enc}} \leftarrow e$.
  3. Generate $\mathsf{P}_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathsf{Enc}[pk, \mathsf{Ext}_{\mathsf{Enc}}]; r_{i\mathcal{O},\mathsf{Enc}})$.
  4. Output the program $\mathsf{P}_{\mathsf{Enc}}$.
```

Program Enc:clean[pk](m; u)

```
// hardcoded values: seNCE public key pk
Inputs: message $m$, randomness $u$
  1. Compute ciphertext $c \leftarrow \mathsf{seNCE.Enc}_{pk}(m; u)$.
  2. Output $c$.
```

Program GenDec:clean[MSK](t)

```
// hardcoded values: master key MSK
Inputs: randomness $t$
  1. Set the randomness $r_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t)$, run $(pk, sk) \leftarrow \mathsf{seNCE.Gen}(r_{\mathsf{NCE}})$.
  2. Output $(t, sk)$.
```

Figure 11: Programs used in hybrids. Note that GenEnc:clean **still produces normal** Enc**, not** Enc:clean**.**

- if a key is punctured on some input, its size is at least $\lambda|input|$.
- if randomness is used as input for sparse extracting PRF, its length should be at least $O(\lambda)$ (since in this case we can construct such a PRF by theorem 4).
- size of an obfuscated program is significantly larger than the size of original program (polynomial in original size s and security parameter λ).

Note that all dependencies in the graph are due to the "hardwired values", i.e. due to the fact that some values should be hardcoded into programs, or messages should be encrypted into ciphertexts. In particular, the same length restrictions remain even when succinct $i\mathcal{O}$ for TM or RAM ([CHJV15, CH15, KLW15]) is used.

Note that the dependency graph is acyclic, and variables which we actually send over the channel - t and c - are in the very top of the graph. This means that we can set length of t and m to be a security parameter, and then set lengths of other variables as large as needed by following edges in dependency graph.

□

6.2 Full proof of security in Theorem 3

Theorem 2. Assuming indistinguishability obfuscation for circuits, one way functions, and same public key NCE with erasures, such that ciphertext has the size $\text{poly}(\lambda) + |m|$, the scheme given in Fig. 8 is a constant-rate non-committing public key encryption scheme in a common reference string model.

Proof. Assume the adversary decides to run N executions in total. In the proof we first switch programs in the CRS to their trapdoor versions and then start switching executions from real to simulated one by one.

In each hybrid we only describe variables specific to a target execution being changed, but we also comment on why other executions are still generated properly.

We consider the following sequence of hybrids that transforms the adversary's view in the real execution to the simulated execution and we argue that the view in each two consecutive hybrids are indistinguishable.

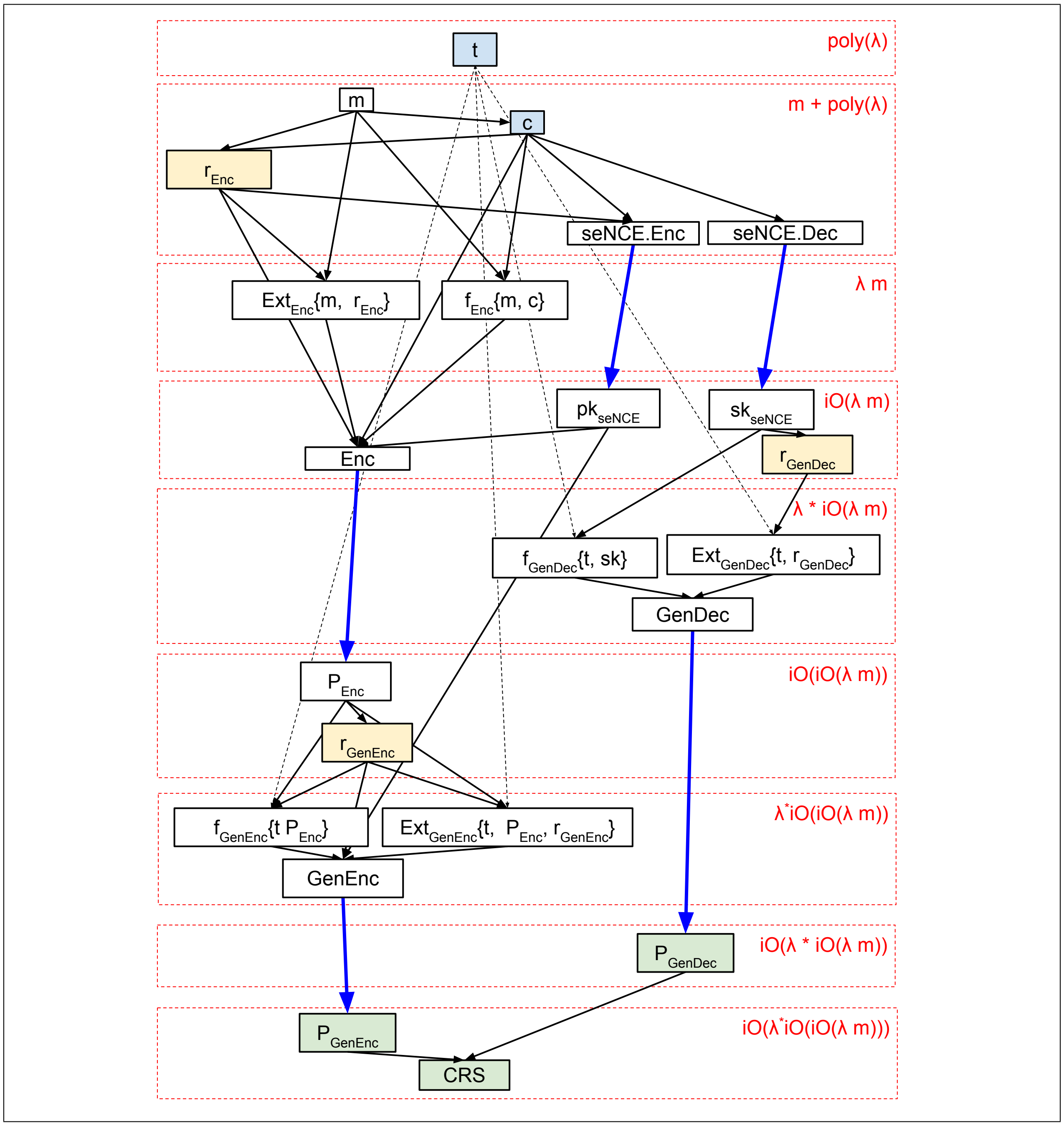

Figure 12: Size dependency graph between different variables, when underlying seNCE is instantiated with our construction from section 5.1. Notation: $i\mathcal{O}(s)$ for size s means the resulting size of an obfuscated program of the initial approximate size s. Dependencies due to obfuscation are drawn as fat blue arrows. Green boxes mark CRS, yellow boxes mark randomness used for extracting PRF, and blue denotes variables which are sent in the protocol. Arrows for t are shown dashed for easier tracking. Red dashed rectangles with complexity in the top right corner denote a complexity group.

Real execution.

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) Sample a key $\mathsf{Ext}_{\mathsf{GenEnc}}$ and obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathsf{GenEnc}[\mathsf{MSK}, \mathsf{Ext}_{\mathsf{GenEnc}}])$,
   (c) Sample a key $\mathsf{Ext}_{\mathsf{GenDec}}$ and obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathsf{GenDec}[\mathsf{MSK}, \mathsf{Ext}_{\mathsf{GenDec}}])$;
   (d) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) choose random r^*_{GenDec} and run $(t^*, sk^*) \leftarrow \mathsf{P}_{\mathsf{GenDec}}(r^*_{\mathsf{GenDec}})$
   (b) choose random $r^*_{\mathsf{GenEnc}}, r^*_{\mathsf{Enc}}$ and run $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{GenEnc}}(t^*, r^*_{\mathsf{GenEnc}})$, $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*, r^*_{\mathsf{Enc}})$
   (c) show (t^*, c^*) as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**
   (a) show $(r^*_{\mathsf{GenEnc}}, r^*_{\mathsf{Enc}})$ as sender internal state and r^*_{GenDec} as receiver internal state.

This hybrid corresponds to the real world. All executions are generated honestly, according to the protocol.

Hybrid 1a - switch GenDec in the CRS to its trapdoor version.

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) Sample a key $\mathsf{Ext}_{\mathsf{GenEnc}}$ and obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathsf{GenEnc}[\mathsf{MSK}, \mathsf{Ext}_{\mathsf{GenEnc}}])$,
   (c) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}, f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}$, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$; obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec}\!:\!\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   (d) Set $CRS = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) choose random $r^*_{\mathsf{GenEnc}}, r^*_{\mathsf{Enc}}$ and run $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{GenEnc}}(t^*, r^*_{\mathsf{GenEnc}})$, $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*, r^*_{\mathsf{Enc}})$
   (b) show (t^*, c^*) as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**
   (a) show $(r^*_{\mathsf{GenEnc}}, r^*_{\mathsf{Enc}})$ as sender internal state and r^*_{GenDec} as receiver internal state.

In this hybrid we add a trapdoor branch to the program GenDec. In other words, instead of generating $\mathsf{P}_{\mathsf{GenDec}}$ as $i\mathcal{O}(\mathsf{GenDec}[\mathsf{MSK}, \mathsf{Ext}_{\mathsf{GenDec}}])$, we generate it as
$i\mathcal{O}(\mathrm{GenDec}\!:\!\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$, where keys are chosen at random and $S_{\mathsf{GenDec}} = \mathsf{prg}(s_{\mathsf{GenDec}})$ for random s_{GenDec}.

Indistinguishability between this and previous hybrid holds by indistinguishability of programs with and without a trapdoor for $\mathsf{Alg} = \mathrm{GenDec}\!:\!\mathrm{clean}$, $\mathrm{Alg}:\mathrm{r} = \mathsf{GenDec}$, $\mathrm{Alg}:\mathrm{td} = \mathrm{GenDec}\!:\!\mathrm{Sim}$. Let us show a reduction. We generate MSK and set $\mathsf{Alg} = \mathrm{GenDec}\!:\!\mathrm{clean}[\mathsf{MSK}]$ as an algorithm for the game. The challenger runs the compiler and outputs either $\mathrm{Alg}:\mathrm{td} = i\mathcal{O}(\mathrm{GenDec}\!:\!\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$ or $\mathrm{Alg}:\mathrm{r} = i\mathcal{O}(\mathsf{GenDec}[\mathsf{MSK}, \mathsf{Ext}_{\mathsf{GenDec}}])$. We set a given program to be $\mathsf{P}^*_{\mathsf{GenDec}}$ and then run the rest of the hybrid.

Here all executions are generated honestly according to the protocol; the only difference is that instead of using program GenDec, GenDec:Sim is used to generate (t, sk).

Hybrid 1b - switch GenEnc in the CRS to its trapdoor version.

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}$ f_{GenEnc}, s_{GenEnc}, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc\!:\!Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}])$;
   (c) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}$, f_{GenDec}, s_{GenDec}, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec\!:\!Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   (d) Set $CRS = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) choose random r^*_{GenEnc}, r^*_{Enc} and run $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{GenEnc}}(t^*, r^*_{\mathsf{GenEnc}})$, $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*, r^*_{\mathsf{Enc}})$
   (b) show (t^*, c^*) as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**
   (a) show $(r^*_{\mathsf{GenEnc}}, r^*_{\mathsf{Enc}})$ as sender internal state and r^*_{GenDec} as receiver internal state.

In this hybrid we add a trapdoor branch to program GenEnc. In other words, instead of generating $\mathsf{P}_{\mathsf{GenEnc}}$ as $i\mathcal{O}(\mathsf{GenEnc}[\mathsf{MSK}, \mathsf{Ext}_{\mathsf{GenEnc}}])$, we generate it as $i\mathcal{O}(\mathrm{GenEnc\!:\!Sim}\ [\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}])$, where keys are chosen at random and $S_{\mathsf{GenEnc}} = \mathsf{prg}(s_{\mathsf{GenEnc}})$ for random s_{GenEnc}.

Indistinguishability between this and previous hybrid holds by indistinguishability of programs with and without a trapdoor for $\mathrm{Alg} = \mathrm{GenEnc\!:\!clean}$, $\mathrm{Alg:r} = \mathsf{GenEnc}$, $\mathrm{Alg:td} = \mathrm{GenEnc\!:\!Sim}$. Let us show a reduction. We generate MSK and set $\mathrm{Alg} = \mathrm{GenEnc\!:\!clean}[\mathsf{MSK}]$ as an algorithm for the game. The challenger runs the compiler and outputs either $\mathrm{Alg:td} = i\mathcal{O}(\mathrm{GenEnc\!:\!Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}])$ or $\mathrm{Alg:r} = i\mathcal{O}(\mathsf{GenEnc}[\mathsf{MSK}, \mathsf{Ext}_{\mathsf{GenEnc}}])$. We set a given program to be $\mathsf{P}^*_{\mathsf{GenEnc}}$ and then run the rest of the hybrid.

Here all executions are generated honestly according to the protocol; the only difference is that instead of using program GenEnc and GenDec, GenEnc:Sim and GenDec:Sim are used to generate $\mathsf{P}_{\mathsf{Enc}}$ and (t^*, sk^*), respectively.

Next we run the sequence hybrids $\mathrm{Hyb}2_l$ - $\mathrm{Hyb}5g_l$ for all executions $l = 1, \ldots, N$.

Hybrid 2_l - fake sender generation randomness.

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}$, f_{GenDec}, s_{GenDec}, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec\!:\!Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   (c) choose random r^*_{GenDec} and run $(t^*, sk^*) \leftarrow \mathsf{P}_{\mathsf{GenDec}}(r^*_{\mathsf{GenDec}})$;
   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}$ f_{GenEnc}, s_{GenEnc}, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc\!:\!Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.
   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) choose random r^*_{GenEnc}, r^*_{Enc} and run $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{GenEnc}}(t^*, r^*_{\mathsf{GenEnc}})$, $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*, r^*_{\mathsf{Enc}})$
   (b) show (t^*, c^*) as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**

(a) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, \mathsf{P}^*_{\mathsf{Enc}}; \rho)$ for some random ρ

(b) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{\mathsf{Enc}})$ as sender internal state and r^*_{GenDec} as receiver internal state.

In this hybrid we generate t^*, sk^* *before* we publish a CRS (they depend on $\mathsf{P}_{\mathsf{GenDec}}$ and random r^*_{GenDec}, so we can do this). We also show fake $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, \mathsf{P}^*_{\mathsf{Enc}}; \rho)$ instead of showing real random r^*_{GenEnc}.

Indistinguishability of this and previous hybrid holds by selective indistinguishability of explanations of algorithm $\mathrm{Alg} : \mathrm{td} = \mathsf{P}_{\mathsf{GenEnc}}$. Let's show a reduction. First we sample keys MSK, $f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}$ and create obfuscated $\mathsf{P}_{\mathsf{GenDec}}$. We choose random r^*_{GenDec} and generate $(t^*, sk^*) \leftarrow \mathsf{P}_{\mathsf{GenDec}}(r^*_{\mathsf{GenDec}})$. We fix t^* as a selective input for indistinguishability of explanations game. We set an algorithm for the game to be $\mathrm{Alg} = \mathrm{GenEnc}\!:\!\mathrm{clean}[\mathsf{MSK}]$.

GM runs the compiler and obtains programs $\mathrm{Alg} : \mathrm{r}, \mathrm{Alg} : \mathrm{td}, \mathsf{Explain}$. Note that $\mathrm{Alg} : \mathrm{td} = i\mathcal{O}(\mathrm{GenEnc}\!:\!\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}])$. The challenger chooses random r^*_{GenEnc}, runs $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathrm{Alg} : \mathrm{td}(t^*, r^*_{\mathsf{GenEnc}})$ on selective input t^*. Next it sets fake randomness $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{Explain}(t^*, \mathsf{P}^*_{\mathsf{Enc}}; \rho)$ for some random ρ. It outputs $\mathrm{Alg} : \mathrm{td}, \mathsf{Explain}, \mathsf{P}^*_{\mathsf{Enc}}$ and either r^*_{GenEnc} or $r^*_{f,\mathsf{GenEnc}}$.

We set the CRS to be $\mathsf{P}_{\mathsf{GenDec}}$ (generated by us) and $\mathrm{Alg} : \mathrm{td}$ from the challenge. We show this CRS to the adversary.

Next we show to the adversary the first message t^*. After it chooses input m^*, we choose random r^*_{Enc} and run challenge $\mathsf{P}^*_{\mathsf{Enc}}$ on $(m^*, r^*_{\mathsf{Enc}})$. Then we show c^* as the second message in the protocol.

To show parties' internal state, we output our own r^*_{Enc}, output challenge generation randomness (which is either r^*_{GenEnc} or $r^*_{f,\mathsf{GenEnc}}$), and output our own r^*_{GenDec}.

Generating other executions. In simulated executions we generate fake $r^j_{f,\mathsf{GenEnc}}$ using challenge program $\mathsf{Explain}$. Real executions are not changed.

Hybrid 3_l - fake receiver generation randomness.

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}, f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}$, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec}\!:\!\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;
   (c) choose random r^*_{GenDec} and run $(t^*, sk^*) \leftarrow \mathsf{P}_{\mathsf{GenDec}}(r^*_{\mathsf{GenDec}})$;
   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}$ $f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}$, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc}\!:\!\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.
   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) choose random $r^*_{\mathsf{GenEnc}}, r^*_{\mathsf{Enc}}$ and run $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{GenEnc}}(t^*, r^*_{\mathsf{GenEnc}})$, $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*, r^*_{\mathsf{Enc}})$
   (b) show (t^*, c^*) as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**
   (a) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1
   (b) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$ for some random ρ_2
   (c) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we show fake $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$ instead of showing real random r^*_{GenDec}.

Indistinguishability of this and previous hybrid holds by selective indistinguishability of explanations of algorithm $\mathrm{Alg} : \mathrm{td} = \mathsf{P}_{\mathsf{GenDec}}$. Let's show a reduction. First we sample keys MSK and fix an empty input for selective indistinguishability of explanations game. We set an algorithm for the game to be $\mathsf{Alg} = \mathrm{GenDec{:}clean}[\mathsf{MSK}]$.

GM runs the compiler and obtains programs $\mathrm{Alg} : \mathrm{r}, \mathrm{Alg} : \mathrm{td}, \mathsf{Explain}$. Note that $\mathrm{Alg} : \mathrm{td} = i\mathcal{O}(\mathrm{GenDec{:}Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$. The challenger chooses random r^*_{GenDec}, runs $(t^*, sk^*) \leftarrow \mathrm{Alg} : \mathrm{td}(r^*_{\mathsf{GenDec}})$. Next it sets fake randomness $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{Explain}(t^*, sk^*; \rho)$ for some random ρ. It outputs $\mathrm{Alg} : \mathrm{td}, \mathsf{Explain}, (t^*, sk^*)$ and either r^*_{GenDec} or $r^*_{f,\mathsf{GenDec}}$.

Next we choose necessary keys for $\mathrm{GenEnc{:}Sim}$ and set the CRS to be $\mathsf{P}_{\mathsf{GenEnc}}$ (generated by us) and $\mathrm{Alg} : \mathrm{td}$ from the challenge. We show this CRS to the adversary.

Next we show to the adversary the first message t^* from the challenge output. After it chooses input m^*, we choose random r^*_{GenEnc} and r^*_{Enc} and run $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{GenEnc}}(t^*; r^*_{\mathsf{GenEnc}})$, $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*, r^*_{\mathsf{Enc}})$. Then we show c^* as the second message in the protocol.

To show parties' internal state, we output our own r^*_{Enc}, output challenge generation randomness (which is either r^*_{GenDec} or $r^*_{f,\mathsf{GenDec}}$), and output our own $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$.

Generating other executions. In simulated executions we generate fake $r^j_{f,\mathsf{GenDec}}$ using challenge program $\mathsf{Explain}$. Real executions are not changed.

Hybrid 4a$_l$ - token t^* is generated at random

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}$, f_{GenDec}, s_{GenDec}, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec{:}Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;
   (c) choose random t^* , set $r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)$, $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$;
   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}$ f_{GenEnc}, s_{GenEnc}, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc{:}Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.
   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) choose random r^*_{GenEnc}, r^*_{Enc} and run $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{GenEnc}}(t^*, r^*_{\mathsf{GenEnc}})$, $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*, r^*_{\mathsf{Enc}})$
   (b) show $(\underline{t^*}, c^*)$ as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**
   (a) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1
   (b) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$ for some random ρ_2
   (c) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we generate (t^*, sk^*) by running $\mathrm{GenDec{:}clean}(t^*)$ instead of running $\mathrm{GenDec{:}Sim}(r^*_{\mathsf{GenDec}})$.

Indistinguishability holds because of indistinguishability of the source of the output for the program GenDec. Let us show the reduction:

We choose MSK and set Alg $=$ $\mathrm{GenDec:clean}[\mathsf{MSK}]$ as an algorithm and and an empty input for indistinguishability of the source of the output game. The challenger runs the compiler to obtain programs $\mathrm{Alg:r}, \mathrm{Alg:td}, \mathsf{Explain}$. Note that $\mathrm{Alg:td} = i\mathcal{O}(\mathrm{GenDec:Sim})$. Then the challenger chooses random r^*_{GenDec} and random t^* and generates (t^*, sk^*) either as $\mathrm{Alg:td}(r^*_{\mathsf{GenDec}})$ or $\mathsf{Alg}(t^*)$ (note that $\mathsf{Alg} = \mathrm{GenDec:clean}[\mathsf{MSK}]$ indeed outputs its input t). The challenger gives us $(\mathrm{Alg:td}, \mathsf{Explain}, (t^*, sk^*))$.

We choose the keys for $\mathrm{GenEnc:Sim}$ and create $\mathsf{P}_{\mathsf{GenEnc}}$ and $\mathsf{P}_{\mathsf{ExplainGenEnc}}$. We set the CRS to be $\mathrm{Alg:td}$ and $\mathsf{P}_{\mathsf{GenEnc}}$.

We show t^* as the first message in the protocol. When the adversary fixes an input m^*, we choose random r^*_{GenEnc}, r^*_{Enc} and generate $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{GenEnc}}(t^*; r^*_{\mathsf{GenEnc}})$ using challenge t^*. Next we set $c^* \leftarrow P^*_{\mathsf{Enc}}(m^*; r^*_{\mathsf{Enc}})$.

Later we generate $r^*_{f,\mathsf{GenDec}}$ using challenge $\mathsf{Explain}(t^*, sk^*; \rho_2)$ for random ρ_2 and generate $r^*_{f,\mathsf{GenEnc}}$ using $\mathsf{P}_{\mathsf{ExplainGenEnc}}$.

Generating other executions. Other executions are generated in the same way as in the previous hybrid. Changing how t in execution l was generated doesn't affect other executions.

Hybrid 4b$_l$ - keys for Enc are generated at random

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}, f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}$, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec:Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;
   (c) choose random t^*, set $r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)$, $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$;
   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}\ f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}$, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc:Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.
   (e) Set $CRS = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) generate $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow \mathrm{GenEnc:clean}[\mathsf{MSK}](t^*; e^*)$ for random e^*:
      i. $\underline{\text{choose random } e^*\text{, sample } \mathsf{Ext}^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*}$
      ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathsf{Enc}[pk^*, \underline{\mathsf{Ext}^*_{\mathsf{Enc}}}]; \underline{r^*_{i\mathcal{O},\mathsf{Enc}}})$
   (b) choose random r^*_{Enc} and generate $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*; r^*_{\mathsf{Enc}})$:
   (c) show (t^*, c^*) as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**
   (a) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1
   (b) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$ for some random ρ_2
   (c) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we generate P^*_{Enc} by running $\mathrm{GenEnc:clean}$ instead of GenEnc.

Indistinguishability holds because of indistinguishability of the source of output for program GenEnc. Let us show the reduction:

We choose MSK, generate the keys for GenDec and obfuscate it to get $\mathsf{P}_{\mathsf{GenDec}}, \mathsf{P}_{\mathsf{ExplainGenDec}}$. Next we choose random t^*, set $r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)$, $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$;

We set $\mathsf{Alg} = \mathrm{GenEnc}{:}\mathrm{clean}[\mathsf{MSK}]$ as an algorithm and t^* as an input for indistinguishability of the source of the output game. The challenger runs the compiler to obtain programs $\mathrm{Alg} : \mathrm{r}, \mathrm{Alg} : \mathrm{td}, \mathsf{Explain}$. Note that $\mathrm{Alg} : \mathrm{td} = i\mathcal{O}(\mathrm{GenEnc}{:}\mathrm{Sim})$. Then the challenger chooses random r^*_{GenEnc} and random e^* and generates $\mathsf{P}^*_{\mathsf{Enc}}$ either as $\mathrm{Alg} : \mathrm{td}(t^*; r^*_{\mathsf{GenEnc}})$ or $\mathsf{Alg}(t^*; e^*)$. The challenger gives us $(\mathrm{Alg} : \mathrm{td}, \mathsf{Explain}, \mathsf{P}^*_{\mathsf{Enc}})$.

We set the CRS to be $\mathrm{Alg} : \mathrm{td}$ and $\mathsf{P}^*_{\mathsf{GenDec}}$.

We show t^* as the first message in the protocol. When the adversary fixes an input m^*, we choose r^*_{Enc} and generate $c^* \leftarrow P^*_{\mathsf{Enc}}(m^*; r^*_{\mathsf{Enc}})$ using the challenge program $\mathsf{P}^*_{\mathsf{Enc}}$.

Later we generate $r^*_{f,\mathsf{GenDec}}$ using $\mathsf{P}_{\mathsf{ExplainGenDec}}$ and generate $r^*_{f,\mathsf{GenEnc}}$ as challenge $\mathsf{Explain}(t^*, P^*_{\mathsf{Enc}}; \rho_2)$.

Generating other executions. Other executions are generated in the same way as in the previous hybrid. Changing how keys for $\mathsf{P}_{\mathsf{Enc}}$ in execution l were generated doesn't affect other executions.

Hybrid 4c$_l$ - $\mathsf{P}^*_{\mathsf{Enc}}$ is an obfuscation of $\mathrm{Enc}{:}\mathrm{Sim}$

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}, f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}$, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec}{:}\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;
   (c) choose random t^* and set $r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)$, $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$;
   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}\ f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}$, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc}{:}\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.
   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of $\mathrm{Enc}{:}\mathrm{Sim}$:
      i. choose random e^*, sample $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*$. Set $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.
      ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathrm{Enc}{:}\mathrm{Sim}[pk^*, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]; r^*_{i\mathcal{O},\mathsf{Enc}})$
   (b) choose random r^*_{Enc} and generate $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*; r^*_{\mathsf{Enc}})$:
   (c) show (t^*, c^*) as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**
   (a) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1
   (b) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$ for some random ρ_2
   (c) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we generate $\mathsf{P}^*_{\mathsf{Enc}}$ as obfuscation of $\mathrm{Enc}{:}\mathrm{Sim}[pk^*, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]$ instead of $\mathsf{Enc}[pk^*, \mathsf{Ext}^*_{\mathsf{Enc}}]$.

Indistinguishability holds because of indistinguishability of programs with and without a trapdoor for program Enc. Let us show a reduction.

We first generate the CRS programs (together with corresponding Explain programs), choose random t^* and set $r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)$, $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$. We show the CRS, as well as the first message t^*, to the adversary.

We set $\mathsf{Alg} = \mathrm{Enc{:}clean}[pk^*]$ as a program for indistinguishability game. The challenger runs its compiler to obtain $\mathrm{Alg:r}, \mathrm{Alg:td}, \mathsf{Explain}$. It gives us a challenge program P which is either $\mathrm{Alg:r}$ or $\mathrm{Alg:td}$.

We choose random r^*_{Enc} and generate $c^* \leftarrow \mathsf{P}(m^*; r^*_{\mathsf{Enc}})$ using challenge program P. We show c^* to the adversary.

Finally we complete the hybrid by generating fake randomness $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, \mathsf{P}; \rho_1)$, encoding challenge P, and $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$. We show $r^*_{f,\mathsf{GenEnc}}$, $r^*_{f,\mathsf{GenDec}}$, and r^*_{Enc} to the adversary.

Generating other executions. Other executions are generated in the same way as in the previous hybrid. Changing how $\mathsf{P}_{\mathsf{Enc}}$ in execution l was generated doesn't affect other executions.

Hybrid 4d$_l$ - fake encryption randomness

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}$, f_{GenDec}, s_{GenDec}, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec{:}Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;
   (c) choose random t^* and set $r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)$, $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$;
   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}$ f_{GenEnc}, s_{GenEnc}, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc{:}Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.
   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of $\mathrm{Enc{:}Sim}$:
      i. choose random e^*, sample $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*$. Set $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.
      ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathrm{Enc{:}Sim}[pk^*, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]; r^*_{i\mathcal{O},\mathsf{Enc}})$
   (b) choose random r^*_{Enc} and generate $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*; r^*_{\mathsf{Enc}})$:
   (c) show (t^*, c^*) as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**
   (a) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1
   (b) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$ for some random ρ_2
   (c) set $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{ExplainEnc}}(m^*, c^*; \rho_3)$ for random ρ_3;
   (d) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we show fake $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{ExplainEnc}}(m^*, c^*; \rho_3)$ instead of showing real random r^*_{Enc}.

Indistinguishability of this and previous hybrid holds by selective indistinguishability of explanations of algorithm $\mathrm{Alg:td} = \mathrm{Enc{:}Sim}$. Let's show a reduction. First we generate a CRS, choose random t^* and compute (pk^*, sk^*). We show the CRS and the first message, t^*, to the adversary. The adversary chooses m^*.

We fix input m^* and program $\mathsf{Alg} = \mathrm{Enc{:}clean}[pk^*]$ for selective indistinguishability of explanations game.

GM runs the compiler and obtains programs $\mathrm{Alg:r}, \mathrm{Alg:td}, \mathsf{Explain}$. Note that $\mathrm{Alg:td} = i\mathcal{O}(\mathrm{Enc{:}Sim}[pk^*, f_{\mathsf{Enc}}, \mathsf{Ext}_{\mathsf{Enc}}, S_{\mathsf{Enc}}])$. The challenger chooses random r^*_{Enc}, runs $c^* \leftarrow \mathrm{Alg:td}(r^*_{\mathsf{Enc}})$. Next it sets fake randomness $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{Explain}(m^*, c^*; \rho)$ for some random ρ. It outputs $\mathrm{Alg:td}, \mathsf{Explain}, c^*$ and either r^*_{Enc} or $r^*_{f,\mathsf{Enc}}$.

We show challenge c^* as the second message in the protocol to the adversary.

To show parties' internal state, we output $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, \text{Alg} : \text{td}; \rho_1)$, output $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$, and output challenge randomness (r^*_{Enc} or $r^*_{f,\mathsf{Enc}}$),

Generating other executions. Other executions are generated in the same way as in the previous hybrid. Changing the generation of r^*_{Enc} in execution l was generated doesn't affect other executions.

Hybrid 5a$_l$ - puncture $\mathsf{MSK}\{t^*\}$ in $\mathrm{GenEnc}\!:\!\mathrm{Sim}$

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}, f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}$, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec}\!:\!\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;
   (c) choose random t^* and set $r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)$, $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$;
   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}$ $f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}$, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc}\!:\!\mathrm{Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}, t^*, pk^*])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.
   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of $\mathrm{Enc}\!:\!\mathrm{Sim}$:
      i. choose random e^*, sample $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*$. Set $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.
      ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathrm{Enc}\!:\!\mathrm{Sim}[pk^*, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]; r^*_{i\mathcal{O},\mathsf{Enc}})$
   (b) choose random r^*_{Enc} and generate $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*; r^*_{\mathsf{Enc}})$:
   (c) show (t^*, c^*) as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**
   (a) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1
   (b) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$ for some random ρ_2
   (c) set $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{ExplainEnc}}(m^*, c^*; \rho_3)$ for random ρ_3;
   (d) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we use a punctured key $\mathsf{MSK}\{t^*\}$ in $\mathsf{P}_{\mathsf{GenEnc}}$, i.e. $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc}\!:\!\mathrm{Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S^*_{\mathsf{GenEnc}}, t^*, pk^*])$ (Fig. 13), with a punctured $\mathsf{MSK}\{t^*\}$ inside and hardwired t^* and pk^*. In $\mathrm{GenEnc}\!:\!\mathrm{Sim1}$ normal branch first checks if $t = t^*$; in this case it uses hardwired public key pk^* to generate program $\mathsf{P}_{\mathsf{Enc}}$. If $t \neq t^*$, then it proceeds as before, computing $r_{\mathsf{NCE}} \leftarrow F_{\mathsf{MSK}\{t^*\}}(t)$, $pk, sk \leftarrow \mathsf{NCE.Gen}(r_{\mathsf{NCE}})$, and using this pk to generate $\mathsf{P}_{\mathsf{Enc}}$. Indistinguishability of this and previous hybrid holds by $i\mathcal{O}$:

Note that we didn't change a trapdoor branch, thus the set of inputs on which the trapdoor branch is executed (and therefore also the set on which the normal branch is executed) is the same for both programs. Consider three cases:

1. If input (t, r) results in executing trapdoor branch, then both programs output the same answer, since their trapdoor branch code is the same.
2. If input (t, r) results in executing normal branch, and $t \neq t^*$, then both programs execute the same steps to compute $\mathsf{P}_{\mathsf{Enc}}$, with the only difference that $\mathrm{GenEnc}\!:\!\mathrm{Sim}$ uses full key MSK and $\mathrm{GenEnc}\!:\!\mathrm{Sim1}$ uses punctured $\mathsf{MSK}\{t^*\}$. Since we assumed that $t \neq t^*$, the output is the same in both programs.

3. If input (t, r) results in executing normal branch, and $t = t^*$, then $\mathrm{GenEnc:Sim}$ computes $r^*_{\mathsf{NCE}} \leftarrow F_{\mathsf{MSK}}(t^*)$, sets $pk^*, sk^* \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$, and uses pk^* to generate $\mathsf{P}_{\mathsf{Enc}}$; $\mathrm{GenEnc:Sim1}$ on input $t = t^*$ directly sets pk^* to be used in generating $\mathsf{P}_{\mathsf{Enc}}$. Therefore they are using the same public key pk^*, and other variables needed to generate $\mathsf{P}_{\mathsf{Enc}}$ (keys $f_{\mathsf{Enc}}, \mathsf{Ext}_{\mathsf{Enc}}, S_{\mathsf{Enc}}$, obfuscation randomness $r_{i\mathcal{O},\mathsf{Enc}}$) are all sampled from the same $e = F_{\mathsf{Ext}_{\mathsf{GenEnc}}}(t^*, r)$; thus they generate exactly the same obfuscated program $\mathsf{P}_{\mathsf{Enc}} = i\mathcal{O}(\mathsf{Enc}[pk^*, f_{\mathsf{Enc}}, \mathsf{Ext}_{\mathsf{Enc}}]; r_{i\mathcal{O},\mathsf{Enc}}])$.

Generating other executions. The difference is that $\mathrm{GenEnc:Sim1}$ is now used to generate $\mathsf{P}_{\mathsf{Enc}}$ in all other executions. In simulated executions t is chosen at random and thus with overwhelming probability $t \neq t^*$; this means that other simulated executions are still generated normally, as if original program $\mathrm{GenEnc:Sim}$ was used.

Now consider the first message in a real execution $t = \mathrm{GenDec:Sim}(r_{\mathsf{GenDec}})$. Since r_{GenDec} is random, with overwhelming probability normal branch is executed and thus $t = F_{\mathsf{Ext}_{\mathsf{GenDec}}}(r_{\mathsf{GenDec}})$. But random t^* is outside the image of $F_{\mathsf{Ext}_{\mathsf{GenDec}}}$ due to sparseness of this PRF, thus $t^* \neq t$, and therefore with overwhelming probability over the choice of t^* other executions are not affected by the change.

Hybrid 5b$_l$ - puncture $\mathsf{MSK}\{t^*\}$ **in** $\mathrm{GenDec:Sim}$

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) choose random t^* and set $r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)$, $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$;
   (c) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}, f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}$, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec:Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;
   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}\ f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}$, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc:Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}, t^*, pk^*])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.
   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.
2. **Generate communications in the protocol:**
   (a) generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of $\mathrm{Enc:Sim}$:
      i. choose random e^*, sample $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*$. Set $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.
      ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathrm{Enc:Sim}[pk^*, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]; r^*_{i\mathcal{O},\mathsf{Enc}})$
   (b) choose random r^*_{Enc} and generate $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*; r^*_{\mathsf{Enc}})$:
   (c) show (t^*, c^*) as communications in the protocol
3. **Show parties' internal state consistent with message** m^* **and communications:**
   (a) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1
   (b) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$ for some random ρ_2
   (c) set $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{ExplainEnc}}(m^*, c^*; \rho_3)$ for random ρ_3;
   (d) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we puncture $\mathsf{MSK}\{t^*\}$ in program $\mathsf{P}_{\mathsf{GenDec}}$, i.e. $\mathsf{P}_{\mathsf{GenDec}} \leftarrow$
$i\mathcal{O}(\mathrm{GenDec:Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$ (Fig. 14); the use of the punctured key instead of the full one is the only difference between programs.

Indistinguishability of this and previous hybrid holds by $i\mathcal{O}$; note that we didn't change a trapdoor branch, thus the set of inputs on which the trapdoor branch is executed (and therefore also the set on which the normal branch is executed) is the same for both programs. Consider two cases:

```
Program GenEnc:Sim1

Program GenEnc:Sim1[$\mathsf{MSK}\{\mathbf{t}^*\}, \mathbf{f}_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, \mathbf{S}_{\mathsf{GenEnc}}, \mathbf{t}^*, \mathbf{pk}^*](\mathbf{t}, \mathbf{r}_{\mathsf{GenEnc}})$
Constants: punctured master key $\mathsf{MSK}\{t^*\}$, faking key $f_{\mathsf{GenEnc}}$, key for sparse extracting prf $\mathsf{Ext}_{\mathsf{GenEnc}}$, prg image $S_{\mathsf{GenEnc}}$, a point $t^*$, seNCE public key $pk^*$
Inputs: token $t$, randomness $r_{\mathsf{GenEnc}}$
  1. Trapdoor branch:
     (a) decode out $\leftarrow \mathsf{PDE.Dec}_{f_{\mathsf{GenEnc}}}(r_{\mathsf{GenEnc}})$; if out $= \perp$ then goto normal branch;
     (b) parse out as $t', \mathsf{P}', s', \tilde{\rho}$. If $t' = t$ and $\mathsf{prg}(s') = S_{\mathsf{GenEnc}}$ then output $\mathsf{P}'$ and halt, else goto normal branch;
  2. Normal branch:
     (a) if $t = t^*$ then set $pk = pk^*$;
     (b) else set $r_{\mathsf{NCE}} \leftarrow F_{\mathsf{MSK}\{t^*\}}(t)$, set $pk, sk \leftarrow \mathsf{NCE.Gen}(r_{\mathsf{NCE}})$
     (c) set randomness $e \leftarrow F_{\mathsf{Ext}_{\mathsf{GenEnc}}}(t, r_{\mathsf{GenEnc}})$, sample $f_{\mathsf{Enc}}, \mathsf{Ext}_{\mathsf{Enc}}, r_{i\mathcal{O},\mathsf{Enc}}, s_{\mathsf{Enc}} \leftarrow e$. Set $S_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s_{\mathsf{Enc}})$.
     (d) create $\mathsf{P}_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathsf{Enc}[pk, \mathsf{Ext}_{\mathsf{Enc}}, f_{\mathsf{Enc}}, S_{\mathsf{Enc}}]; r_{i\mathcal{O},\mathsf{Enc}})$
     (e) output $\mathsf{P}_{\mathsf{Enc}}$
```

Figure 13: Program GenEnc:Sim1**.**

```
Program GenDec:Sim1

Program GenDec:Sim1[$\mathsf{MSK}\{\mathbf{t}^*\}, \mathbf{f}_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, \mathbf{S}_{\mathsf{GenDec}}](\mathbf{r}_{\mathsf{GenDec}})$
Constants: punctured master key $\mathsf{MSK}\{t^*\}$ , receiver faking key $f_{\mathsf{GenDec}}$, key for sparse extracting prf $\mathsf{Ext}_{\mathsf{GenDec}}$, prg image $S_{\mathsf{GenDec}}$
Inputs: randomness $r_{\mathsf{GenDec}}$
  1. Trapdoor branch:
     (a) decode out $\leftarrow \mathsf{PDE.Dec}_{f_{\mathsf{GenDec}}}(r_{\mathsf{GenDec}})$; if out $= \perp$ then goto normal branch;
     (b) parse out as $(t', sk', s', \tilde{\rho})$. If $\mathsf{prg}(s') = S_{\mathsf{GenDec}}$ then output $(t', sk')$ and halt; else goto normal branch.
  2. Normal branch:
     (a) generate token $t \leftarrow F_{\mathsf{Ext}_{\mathsf{GenDec}}}(r_{\mathsf{GenDec}})$
     (b) set randomness $r_{\mathsf{NCE}} \leftarrow F_{\mathsf{MSK}\{t^*\}}(t)$, run $(pk, sk) \leftarrow \mathsf{NCE.Gen}(r_{\mathsf{NCE}})$
     (c) output $(t, sk)$
```

Figure 14: Program GenDec:Sim1**.**

1. If input (t, r) results in executing the trapdoor branch, then both programs output the same answer, since their trapdoor branch code is the same.

2. If input (t, r) results in executing the normal branch, then, since t^* is randomly chosen, with overwhelming probability over the choice of t^* it is outside the image of $F_{\mathsf{Ext}_{\mathsf{GenDec}}}$ due to sparseness of this PRF. Thus no input r result in $t = t^*$, and we can safely puncture the key without changing the functionality.

Generating other executions. The difference is that we use GenDec:Sim1 to generate t, sk in other executions. In simulated executions t is chosen at random and thus with overwhelming probability $t \neq t^*$; this means that other simulated executions are still generated normally.

Consider the first message in a real execution $t = \mathrm{GenDec:Sim}(r_{\mathsf{GenDec}})$. Since r_{GenDec} is random, with overwhelming probability normal branch is executed and thus $t = F_{\mathsf{Ext}_{\mathsf{GenDec}}}(r_{\mathsf{GenDec}})$. But random t^* is outside the image of $F_{\mathsf{Ext}_{\mathsf{GenDec}}}$ due to sparseness of this PRF, thus $t^* \neq t$, and other real executions are generated normally, as if original program GenDec:Sim was used.

Hybrid 5c$_l$ - choose at random punctured value r^*_{NCE}

1. **Generate a CRS:**

   (a) Sample MSK;

(b) choose random t^*, choose random r^*_{NCE}. Set $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$;

(c) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}$, f_{GenDec}, s_{GenDec}, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec}:\mathrm{Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;

(d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}$ f_{GenEnc}, s_{GenEnc}, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc}:\mathrm{Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}, t^*, pk^*])$,
obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.

(e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.

2. **Generate communications in the protocol:**

   (a) generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of $\mathrm{Enc}:\mathrm{Sim}$:

   i. choose random e^*, sample $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*$. Set $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.

   ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathrm{Enc}:\mathrm{Sim}[pk^*, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]; r^*_{i\mathcal{O},\mathsf{Enc}})$

   (b) choose random r^*_{Enc} and generate $c^* \leftarrow \mathsf{P}^*_{\mathsf{Enc}}(m^*; r^*_{\mathsf{Enc}})$:

   (c) show (t^*, c^*) as communications in the protocol

3. **Show parties' internal state consistent with message m^* and communications:**

   (a) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1

   (b) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$ for some random ρ_2

   (c) set $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{ExplainEnc}}(m^*, c^*; \rho_3)$ for random ρ_3;

   (d) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we choose r^*_{NCE} at random instead of $F_{\mathsf{MSK}}(t^*)$.

Indistinguishability holds by selective security of puncturable PRF F_{MSK}: we first choose random t^* and give it to puncturable PRF challenger as a point to puncture at. We get back $\mathsf{MSK}\{t^*\}$ and either $r^*_{\mathsf{NCE}} = F_{\mathsf{MSK}}(t^*)$ or random r^*_{NCE}. We use a given key to generate programs, and use given r^*_{NCE} to generate $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$. Depending on whether we got random r^*_{NCE} or not, we are either in this hybrid or in the previous hybrid.

Generating other executions. Similar to previous two hybrids, we can assume that for real and simulated t it holds that $t \neq t^*$, and changing $r^*_{\mathsf{NCE}} = F_{\mathsf{MSK}}(t^*)$ to a random value doesn't affect other executions.

Hybrid 5d$_l$ - encrypt m^* using fresh randomness u

1. **Generate a CRS:**

   (a) Sample MSK;

   (b) choose random t^*, choose random r^*_{NCE}. Set $(pk^*, sk^*) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$;

   (c) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}$, f_{GenDec}, s_{GenDec}, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec}:\mathrm{Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;

   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}$ f_{GenEnc}, s_{GenEnc}, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc}:\mathrm{Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}, t^*, pk^*])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.

   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish a CRS.

2. **Generate communications in the protocol:**

(a) generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of $\mathrm{Enc}\!:\!\mathrm{Sim}$:

i. choose random e^*, sample $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*$. Set $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.

ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathrm{Enc}\!:\!\mathrm{Sim}[pk^*, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]; r^*_{i\mathcal{O},\mathsf{Enc}})$

(b) choose random u^* and generate $c^* \leftarrow \mathsf{seNCE.Enc}(pk^*; m^*; u^*)$

(c) show (t^*, c^*) as communications in the protocol

3. **Show parties' internal state consistent with message m^* and communications:**

(a) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1

(b) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*; \rho_2)$ for some random ρ_2

(c) set $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{ExplainEnc}}(m^*, c^*; \rho_3)$ for random ρ_3;

(d) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we generate $c^* \leftarrow \mathrm{Enc}\!:\!\mathrm{clean}[pk^*](m^*; u^*)$ for random u^*, instead of running P^*_{Enc}.

Indistinuishability holds by indistinguishability of source of the output for algorithm $\mathsf{Alg} = \mathrm{Enc}\!:\!\mathrm{clean}$. Let us show a reduction.

We start by generating random t^*, r^*_{NCE} and necessary keys for the CRS, obfuscating programs and showing this CRS to the adversary. We also show t^*.

We set $\mathsf{Alg} = \mathrm{Enc}\!:\!\mathrm{clean}[pk^*]$ to be an algorithm and m^* to be an input for the game. The challenger runs the compiler and gets programs $\mathrm{Alg} : \mathrm{r}, \mathrm{Alg} : \mathrm{td}, \mathsf{Explain}$. Note that $\mathrm{Alg} : \mathrm{td} = \mathsf{P}^*_{\mathsf{Enc}}$. The challenger chooses r^*_{Enc} and u^* and generates c^* either as $\mathrm{Alg} : \mathrm{td}(m^*; r^*_{\mathsf{Enc}})$ or $\mathsf{Alg}(m^*; u^*)$. It gives us $(\mathrm{Alg} : \mathrm{td}, \mathsf{Explain}, c^*)$.

We show challenge c^* to the adversary as ciphertext. Upon request to show internal state we encode challenge $\mathrm{Alg} : \mathrm{td}$ into $r^*_{f,\mathsf{GenEnc}}$. $r^*_{f,\mathsf{GenDec}}$ is generated as described in the hybrid. $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{Explain}(m^*, c^*; \rho_3)$.

Generating other executions. Other executions are not affected by the changes - we generate them as in the previous hybrid.

Hybrid 5e$_l$ - switch pk^*, sk^*, c^* to simulated.

1. **Generate a CRS:**

(a) Sample MSK;

(b) choose random t^*, choose random r^*_{NCE}. Set $(pk^*_f, c^*_f, \mathsf{st}) \leftarrow \mathsf{seNCE.Sim}(r^*_{\mathsf{NCE}})$;

(c) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}, f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}$, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec}\!:\!\mathrm{Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;

(d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}\ f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}$, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc}\!:\!\mathrm{Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}, t^*, pk^*_f])$,
obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.

(e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.

2. **Generate communications in the protocol:**

(a) generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of $\mathrm{Enc}\!:\!\mathrm{Sim}$:

i. choose random e^*, sample $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*$. Set $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.

ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathrm{Enc}\!:\!\mathrm{Sim}[pk^*_f, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]; r^*_{i\mathcal{O},\mathsf{Enc}})$

   (b) show (t^*, c_f^*) as communications in the protocol

3. **Show parties' internal state consistent with message m^* and communications:**

   (a) $\underline{\text{set } sk_f^* \leftarrow \mathsf{Sim}(\mathsf{st}, m^*)}$

   (b) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1

   (c) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk_f^*; \rho_2)$ for some random ρ_2

   (d) set $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{ExplainEnc}}(m^*, c_f^*; \rho_3)$ for random ρ_3;

   (e) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we switch real pk^*, c^*, sk^* to simulated pk_f^*, c_f^*, sk_f^*, where $pk^*, c_f^*, \mathsf{st} \leftarrow \mathsf{seNCE.Sim}(r^*_{\mathsf{NCE}})$, and $sk_f^* \leftarrow \mathsf{seNCE.Sim}(\mathsf{st}, m^*)$ (st is the state of a simulator).

Indistinguishability holds by security of underlying seNCE. Let us show the reduction. First we generate CRS keys, random t^* and get pk^* or pk_f^* from NCE challenger. We obfuscate CRS programs (in particular, we hardwire given pk^* or pk_f^* into GenEnc) and publish the CRS.

We show t^* to the adversary as the first message in our protocol. The adversary fixes input m^*. We get either c^* (encrypting m^*) or c_f^* (dummy) from seNCE challenger and output it as the second message in the protocol.

When the adversary asks us to show internal state for message m^*, we first ask the challenger to provide us with a secret key for m^* and get back either sk^* or sk_f^*. Then we sample necessary keys and obfuscation randomness $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}}$ and generate $\mathsf{P}^*_{\mathsf{Enc}}$ with pk^* or pk_f^* hardwired. Then we create internal state by encoding $(t^*, \mathsf{P}^*_{\mathsf{Enc}})$ into $r^*_{f,\mathsf{GenEnc}}$; t^* and challenge secret key (sk^* or sk_f^*) into $r^*_{f,\mathsf{GenDec}}$; and challenge ciphertext (c^* or c_f^*) into $r^*_{f,\mathsf{Enc}}$. We show these three random values as internal state of the parties.

Generating other executions. We can assume that $r_{\mathsf{NCE}} \neq r^*_{\mathsf{NCE}}$ for all other executions and therefore switching seNCE values, obtained from r^*_{NCE}, to simulated doesn't affect other executions.

Hybrid 5f$_l$ - switch r^*_{NCE} back to the result of the PRF.

1. **Generate a CRS:**

   (a) Sample MSK;

   (b) choose random t^*, $\underline{\text{set } r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)}$. Set $(pk_f^*, c_f^*, \mathsf{st}) \leftarrow \mathsf{seNCE.Sim}(r^*_{\mathsf{NCE}})$;

   (c) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}, f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}$, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec{:}Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;

   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}$ $f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}$, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc{:}Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}, t^*, pk_f^*])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.

   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.

2. **Generate communications in the protocol:**

   (a) generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of $\mathrm{Enc{:}Sim}$:

      i. choose random e^*, sample $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*$. Set $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.

      ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathrm{Enc{:}Sim}[pk_f^*, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]; r^*_{i\mathcal{O},\mathsf{Enc}})$

   (b) show (t^*, c_f^*) as communications in the protocol

3. **Show parties' internal state consistent with message m^* and communications:**

(a) set $sk_f^* \leftarrow \mathsf{Sim}(\mathsf{st}, m^*)$

(b) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1

(c) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk_f^*; \rho_2)$ for some random ρ_2

(d) set $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{ExplainEnc}}(m^*, c_f^*; \rho_3)$ for random ρ_3;

(e) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we set $r^*_{\mathsf{NCE}} \leftarrow F_{\mathsf{MSK}}(t^*)$ instead of choosing it at random.

Indistinguishability holds because of selective security of a puncturable PRF $F_{\mathsf{MSK}\{t^*\}}$. Let us show a reduction. We first choose random t^* and give it to PRF challenger as a point to puncture at. The challenger gives back $\mathsf{MSK}\{t^*\}$ and value r^*_{NCE} which is either random or $F_{\mathsf{MSK}}(t^*)$. We proceed with using this punctured key and $(pk_f^*, c_f^*, \mathsf{st}) \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$ in generating the rest of the hybrid.

Generating other executions. We can assume that real and simulated $t \neq t^*$ and changing random r^*_{NCE} to $F_{\mathsf{MSK}}(t^*)$ doesn't affect other executions.

Hybrid 5g$_l$ - Unpuncture $\mathsf{MSK}\{t^*\}$ in $\mathrm{GenDec{:}Sim}$.

1. **Generate a CRS:**

   (a) Sample MSK;

   (b) choose random t^*, set $r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)$. Set $(pk_f^*, c_f^*, \mathsf{st}) \leftarrow \mathsf{seNCE.Sim}(r^*_{\mathsf{NCE}})$;

   (c) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}, f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}$, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   $\underline{\text{obfuscate } \mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec{:}Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}]);}$
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;

   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}\ f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}$, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc{:}Sim1}[\mathsf{MSK}\{t^*\}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}, t^*, pk_f^*])$,
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.

   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish the CRS.

2. **Generate communications in the protocol:**

   (a) generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of $\mathrm{Enc{:}Sim}$:

      i. choose random e^*, sample $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*$. Set $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.

      ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathrm{Enc{:}Sim}[pk_f^*, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]; r^*_{i\mathcal{O},\mathsf{Enc}})$

   (b) show (t^*, c_f^*) as communications in the protocol

3. **Show parties' internal state consistent with message m^* and communications:**

   (a) set $sk_f^* \leftarrow \mathsf{Sim}(\mathsf{st}, m^*)$

   (b) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1

   (c) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk_f^*; \rho_2)$ for some random ρ_2

   (d) set $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{ExplainEnc}}(m^*, c_f^*; \rho_3)$ for random ρ_3;

   (e) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we show original program $\mathrm{GenDec{:}Sim}$ instead of $\mathrm{GenDec{:}Sim1}$, i.e. we unpuncture $\mathsf{MSK}\{t^*\}$ in GenDec.

Indistinguishability holds because of $i\mathcal{O}$. Indeed, since t^* is random, with overwhelming probability it is outside the image of sparse PRF $F_{\mathsf{Ext}_{\mathsf{GenDec}}}$, thus, no input r result in evaluating this PRF on t^*, and we can bring the punctured value back.

Generating other executions. Since in other executions with overwhelming probability $t \neq t^*$, this doesn't affect other executions.

Hybrid 5h$_l$ - Unpuncture $\mathsf{MSK}\{t^*\}$ in $\mathrm{GenEnc}:\mathrm{Sim}$.

1. **Generate a CRS:**
   (a) Sample MSK;
   (b) choose random t^*, set $r^*_{\mathsf{NCE}} \leftarrow \mathsf{F}_{\mathsf{MSK}}(t^*)$. Set $(pk^*_f, c^*_f, \mathsf{st}) \leftarrow \mathsf{seNCE.Sim}(r^*_{\mathsf{NCE}})$;
   (c) Sample keys $\mathsf{Ext}_{\mathsf{GenDec}}$, f_{GenDec}, s_{GenDec}, set $S_{\mathsf{GenDec}} \leftarrow \mathsf{prg}(s_{\mathsf{GenDec}})$;
   obfuscate $\mathsf{P}_{\mathsf{GenDec}} \leftarrow i\mathcal{O}(\mathrm{GenDec}:\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenDec}}, \mathsf{Ext}_{\mathsf{GenDec}}, S_{\mathsf{GenDec}}])$;
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenDec}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenDec}}, s_{\mathsf{GenDec}}])$;
   (d) Sample keys $\mathsf{Ext}_{\mathsf{GenEnc}}$ f_{GenEnc}, s_{GenEnc}, set $S_{\mathsf{GenEnc}} \leftarrow \mathsf{prg}(s_{\mathsf{GenEnc}})$;
   $\underline{\text{obfuscate } \mathsf{P}_{\mathsf{GenEnc}} \leftarrow i\mathcal{O}(\mathrm{GenEnc}:\mathrm{Sim}[\mathsf{MSK}, f_{\mathsf{GenEnc}}, \mathsf{Ext}_{\mathsf{GenEnc}}, S_{\mathsf{GenEnc}}]),}$
   obfuscate $\mathsf{P}_{\mathsf{ExplainGenEnc}} \leftarrow i\mathcal{O}(\mathsf{Explain}[f_{\mathsf{GenEnc}}, s_{\mathsf{GenEnc}}])$.
   (e) Set $\mathsf{CRS} = (\mathsf{P}_{\mathsf{GenEnc}}, \mathsf{P}_{\mathsf{GenDec}})$. Publish a CRS.
2. **Generate communications in the protocol:**
   (a) generate $\mathsf{P}^*_{\mathsf{Enc}}$ as an obfuscation of $\mathrm{Enc}:\mathrm{Sim}$:
      i. choose random e^*, sample $\mathsf{Ext}^*_{\mathsf{Enc}}, f^*_{\mathsf{Enc}}, s^*_{\mathsf{Enc}}, r^*_{i\mathcal{O},\mathsf{Enc}} \leftarrow e^*$. Set $S^*_{\mathsf{Enc}} \leftarrow \mathsf{prg}(s^*_{\mathsf{Enc}})$.
      ii. create $\mathsf{P}^*_{\mathsf{Enc}} \leftarrow i\mathcal{O}(\mathrm{Enc}:\mathrm{Sim}[pk^*_f, f^*_{\mathsf{Enc}}, \mathsf{Ext}^*_{\mathsf{Enc}}, S^*_{\mathsf{Enc}}]; r^*_{i\mathcal{O},\mathsf{Enc}})$
   (b) show (t^*, c^*_f) as communications in the protocol
3. **Show parties' internal state consistent with message m^* and communications:**
   (a) set $sk^*_f \leftarrow \mathsf{Sim}(\mathsf{st}, m^*)$
   (b) set $r^*_{f,\mathsf{GenEnc}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenEnc}}(t^*, P^*_{\mathsf{Enc}}; \rho_1)$ for some random ρ_1
   (c) set $r^*_{f,\mathsf{GenDec}} \leftarrow \mathsf{P}_{\mathsf{ExplainGenDec}}(t^*, sk^*_f; \rho_2)$ for some random ρ_2
   (d) set $r^*_{f,\mathsf{Enc}} \leftarrow \mathsf{P}_{\mathsf{ExplainEnc}}(m^*, c^*_f; \rho_3)$ for random ρ_3;
   (e) show $(r^*_{f,\mathsf{GenEnc}}, r^*_{f,\mathsf{Enc}})$ as sender internal state and $r^*_{f,\mathsf{GenDec}}$ as receiver internal state.

In this hybrid we show original program $\mathrm{GenEnc}:\mathrm{Sim}$ instead of $\mathrm{GenEnc}:\mathrm{Sim1}$, i.e. we remove the line "if $t = t^*$ then set $pk = pk^*_f$" and unpuncture $\mathsf{MSK}\{t^*\}$.

Indistinguishability holds because of $i\mathcal{O}$. Note that we didn't change the trapdoor branch, thus the set of inputs on which the trapdoor branch is executed (and therefore also the set on which the normal branch is executed) is the same for both programs. Consider three cases:

1. If input (t, r) results in executing the trapdoor branch, then both programs output the same answer, since their trapdoor branch code is the same.
2. If input (t, r) results in executing the normal branch, and $t \neq t^*$, then both programs execute the same steps to compute $\mathsf{P}_{\mathsf{Enc}}$, with the only difference that $\mathrm{GenEnc}:\mathrm{Sim}$ uses full key MSK and GenEnc : 1 uses punctured $\mathsf{MSK}\{t^*\}$. Since we assumed that $t \neq t^*$, the output is the same in both programs.

3. If input (t, r) results in executing the normal branch, and $t = t^*$, then $\mathrm{GenEnc\!:\!Sim}$ computes $r^*_{\mathsf{NCE}} \leftarrow F_{\mathsf{MSK}}(t^*)$, sets $pk^*, sk^* \leftarrow \mathsf{seNCE.Gen}(r^*_{\mathsf{NCE}})$, and uses pk^* to generate $\mathsf{P}_{\mathsf{Enc}}$. $\mathrm{GenEnc\!:\!Sim1}$ on input $t = t^*$ directly sets pk^*_f to be used in generating $\mathsf{P}_{\mathsf{Enc}}$, where $(pk^*_f, c^*_f, st) \leftarrow \mathsf{seNCE.Sim}(r^*_{\mathsf{NCE}})$, $r^*_{\mathsf{NCE}} \leftarrow F_{\mathsf{MSK}}(t^*)$. Importantly, pk^* obtained by running $\mathsf{NCE.Gen}$ on r^*_{NCE} is exactly the same as pk^*_f obtained by running $\mathsf{NCE.Sim}$ on r^*_{NCE} by the "fixed-public-key" property of underlying seNCE. Therefore they are using the same public key $pk^* = pk^*_f$, and other variables needed to generate $\mathsf{P}_{\mathsf{Enc}}$ (keys $f_{\mathsf{Enc}}, \mathsf{Ext}_{\mathsf{Enc}}$, obfuscation randomness $r_{i\mathcal{O},\mathsf{Enc}}$) are all sampled from the same $e = F_{\mathsf{Ext}_{\mathsf{GenEnc}}}(t^*, r)$; thus they generate exactly the same obfuscated program $\mathsf{P}_{\mathsf{Enc}} = i\mathcal{O}(\mathsf{Enc}[pk^*, f_{\mathsf{Enc}}, \mathsf{Ext}_{\mathsf{Enc}}]; r_{i\mathcal{O},\mathsf{Enc}}])$.

Generating other executions. Since for other executions $t \neq t^*$, this change doesn't affect other executions.

This is hybrid is equivalent to simulation. □

References

[Bea97] Donald Beaver. Plug and play encryption. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, pages 75–89, 1997. 1

[BGI$^+$01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, pages 1–18, 2001. 44

[BGI$^+$12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012. 44

[BGL$^+$15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In Ronitt Rubinfeld, editor, *Symposium on the Theory of Computing (STOC)*, 2015. 5

[BH92] Donald Beaver and Stuart Haber. Cryptographic protocols provably secure against dynamic adversaries. In *Advances in Cryptology - EUROCRYPT '92, Workshop on the Theory and Application of of Cryptographic Techniques, Balatonfüred, Hungary, May 24-28, 1992, Proceedings*, pages 307–323, 1992. 1

[CDMW09] Seung Geol Choi, Dana Dachman-Soled, Tal Malkin, and Hoeteck Wee. Simple, black-box constructions of adaptively secure protocols. In *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, pages 387–402, 2009. 1

[CFGN96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 639–648, 1996. 1

[CGP15] Ran Canetti, Shafi Goldwasser, and Oxana Poburinnaya. Adaptively secure two-party computation from indistinguishability obfuscation. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, pages 557–585, 2015. 1

[CH15] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. *IACR Cryptology ePrint Archive*, 2015:388, 2015. 5, 25

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for RAM programs. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 429–437, 2015. 5, 25

[CHK05] Ran Canetti, Shai Halevi, and Jonathan Katz. Adaptively-secure, non-interactive public-key encryption. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, pages 150–168, 2005. 1, 2, 11

[DKR15] Dana Dachman-Soled, Jonathan Katz, and Vanishree Rao. Adaptively secure, universally composable, multiparty computation in constant rounds. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, pages 586–613, 2015. 1, 2, 5, 9, 46, 47, 49

[DN00] Ivan Damgård and Jesper Buus Nielsen. Improved non-committing encryption schemes based on a general complexity assumption. In *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, pages 432–450, 2000. 1

[DNRS99] Cynthia Dwork, Moni Naor, Omer Reingold, and Larry J. Stockmeyer. Magic functions. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 523–534, 1999. 1

[GGH$^+$13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, 2013. 44

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pages 464–479, 1984. 44

[GP15] Sanjam Garg and Antigoni Polychroniadou. Two-round adaptively secure MPC from indistinguishability obfuscation. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, pages 614–637, 2015. 1

[Had00] Satoshi Hada. Zero-knowledge and code obfuscation. In *Advances in Cryptology - ASIACRYPT'00*, volume 1976 of *Lecture Notes in Computer Science*, pages 443–457. Springer, 2000. 44

[HOR15] Brett Hemenway, Rafail Ostrovsky, and Alon Rosen. Non-committing encryption from Φ-hiding. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part I*, pages 591–608, 2015. 1

[HORR16] Brett Hemenway, Rafail Ostrovsky, Silas Richelson, and Alon Rosen. Adaptive security with quasi-optimal rate. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*, pages 525–541, 2016. 1

[JL00] Stanislaw Jarecki and Anna Lysyanskaya. Adaptively secure threshold cryptography: Introducing concurrency, removing erasures. In *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, pages 221–242, 2000. 1, 2, 11

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In Ronitt Rubinfeld, editor, *Symposium on the Theory of Computing (STOC)*, 2015. 5, 25

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, pages 111–126, 2002. 1, 7

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 475–484, 2014. 2, 3, 4, 5, 9, 15, 45, 46, 47, 49

[Wat14] Brent Waters. A punctured programming approach to adaptively secure functional encryption. *IACR Cryptology ePrint Archive*, 2014:588, 2014. 45, 46

A Preliminaries

A.1 Indistinguishability Obfuscation for Circuits

The goal of obfuscation is to provide a method that transforms any program in an obfuscated version that hides the details of the implemented functionality but still provides the capability to evaluate it on any input. The study of obfuscation was initiated by Hada [Had00] and Barak et al. [BGI+01, BGI+12].

There are several different security notions for obfuscation. In our constructions we will be using *indistinguishability obfuscation*($i\mathcal{O}$) which we define next. Indistinguishability obfuscation was introduced by Barak et al. [BGI+01] and the first candidate construction for $i\mathcal{O}$ for any polynomial size circuit was presented by the work of Garg et a. [GGH+13].

Definition 4 (Indistinguishability Obfuscation ($i\mathcal{O}$))**.** *A uniform PPT machine $i\mathcal{O}$ is called an* indistinguishability obfuscator *if the following conditions are satisfied:*

- *For all security parameters $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$, for all inputs x, we have that*
$$\mathsf{Pr}[C'(x) = C(x) : C' \leftarrow i\mathcal{O}(1^\lambda, C)] = 1$$
- *There is a polynomial p such that for every circuit $C \in \mathcal{C}_\lambda$, it holds that $|i\mathcal{O}(c)| \leq p(|C|)$.*
- *For any (not necessarily uniform) PPT distinguisher D, there exists a negligible function α such that the following holds: For all security parameters $\lambda \in \mathbb{N}$, for all circuit families $C_0 = \{C^0_\lambda\}_{\lambda\in\mathbb{N}}, C_1 = \{C^1_\lambda\}_{\lambda\in\mathbb{N}}$ of size $|C^0_\lambda| = |C^1_\lambda|$, we have that if $C^0_\lambda(x) = C^1_\lambda(x)$ for all inputs x, then*
$$\Big|\mathsf{Pr}\big[D(i\mathcal{O}(1^\lambda, C^0_\lambda)) = 1\big] - \mathsf{Pr}\big[D(i\mathcal{O}(1^\lambda, C^1_\lambda)) = 1\big]\Big| \leq \mathsf{negl}(\lambda)$$

A.2 Puncturable Pseudorandom Functions and their variants

Puncturable PRFs. In puncrurable PRFs it is possible to create a key that is punctured at a set S of polynomial size. A key k punctured at S (denoted $k\{S\}$) allows evaluating the PRF at all points not in S. Furthermore, the function values at points in S remain pseudorandom even given $k\{S\}$.

Definition 5. *A puncturable pseudorandom function family for input size $n(\lambda)$ and output size $m(\lambda)$ is a tuple of algorithms $\{\mathsf{Sample}, \mathsf{Puncture}, \mathsf{Eval}\}$ such that the following properties hold:*

- **Functionality preserved under puncturing:** *For any PPT adversary A which outputs a set $S \subset \{0,1\}^n$, for any $x \notin S$,*
$$\mathsf{Pr}[F_k(x) = F_{k\{S\}}(x) : k \leftarrow \mathsf{Sample}(1^\lambda), k\{S\} \leftarrow \mathsf{Puncture}(k, S)] = 1.$$
- **Pseudorandomness at punctured points:** *For any PPT adversaries A_1, A_2, define a set S and state* state *as $(S, \mathsf{state}) \leftarrow A_1(1^\lambda)$. Then*
$$\mathsf{Pr}[A_2(\mathsf{state}, S, k\{S\}, F_k(S))] - \mathsf{Pr}[A_2(\mathsf{state}, S, k\{S\}, U_{|S|\cdot m(\lambda)})] < \mathsf{negl}(\lambda),$$
where $F_k(S)$ denotes concatenated PRF values on inputs from S, i.e. $F_k(S) = \{F_k(x_i) : x_i \in S\}$.

The GGM PRF [GGM84] satisfies this definition.

Statistically injective puncturable PRFs. Such PRFs are injective with overwhelming probability over the choice of a key. Sahai and Waters [SW14] show that if F is a puncturable PRF where the output length is large enough compared to the input length, and h is 2-universal hash function, then $\mathsf{F}'_{k,h} = \mathsf{F}_k(x) \oplus h(x)$ is a statistically injective puncturable PRF.

Extracting puncturable PRFs. Such PRFs have a property of a strong extractor: even when a full key is known, the output of the PRF is statistically close to uniform, as long as there is enough min-entropy in the input. Sahai and Waters [SW14] showed that if the input length is large enough compared to the output length, then such PRF can be constructed from any puncturable PRF F as $\mathsf{F}'_{k,h} = h(\mathsf{F}_k(x))$, where h is 2-universal hash function.

Sparse computationally extracting puncturable PRFs. We need a slightly modified version of extracting PRFs: we relax the extracting requirement from statistical to computational, but require our PRF to have a sparse image.

Definition 6. *A PRF family with a key* Ext *mapping a* $a\{0,1\}^{n(\lambda)}$ *to* $\{0,1\}^{l(\lambda)}$ *is a sparse computationally extracting family for min-entropy* $k(\lambda)$*, if the following two conditions hold:*

- **Sparseness:** $\mathsf{Pr}[r \in \mathsf{Im}(\mathsf{F}_{\mathsf{Ext}}) : \mathsf{Ext} \leftarrow \mathsf{Sample}(1^\lambda), r \leftarrow U_l] < \nu(\lambda)$ *for some negligible function* ν*;*
- **Computational extractor:** *If distribution X has min-entropy at least* $k(\lambda)$*, then with overwhelming probability over the choice of key* Ext *for any PPT adversary* $\mathcal{A}$

$$|\,\mathsf{Pr}\,[\,\mathcal{A}(\mathsf{Ext}, \mathsf{F}_{\mathsf{Ext}}(x)) = 1 \mid x \leftarrow X\,] - \mathsf{Pr}\,[\,\mathcal{A}(\mathsf{Ext}, r) = 1 \mid r \leftarrow U_I\,]\,\,| < \mathsf{negl}(\lambda).$$

Theorem 4. *Assuming one way functions exist, if* $n(\lambda) \geq k(\lambda) \geq 3\lambda + 2$ *and* $l(\lambda) \geq 2\lambda$*, then there exists a sparse extracting puncturable PRF family for min-entropy* $k(\lambda)$ *mapping* $n(\lambda)$*-bit inputs to* $l(\lambda)$*-bit outputs.*

Proof. [SW14] show that if $n(\lambda) \geq k(\lambda) \geq m(\lambda) + 2e(\lambda) + 2$, then there exist an extracting PRF family mapping $n(\lambda)$-bit strings to $m(\lambda)$-bit strings such that their output is $2^{-e(\lambda)}$-close to uniform even with the key given, as long as an input has min-entropy at least $k(\lambda)$. By setting $m = \lambda, e = \lambda$ we obtain an extracting family $\{F_K\}$ with outputs $2^{-\lambda}$-close to uniform, as long as $k(\lambda) \geq 3\lambda + 2$.

Let us set our sparse computationally extracting PRF to be $\mathsf{F}'_K = \mathsf{prg}(\mathsf{F}_K)$, where prg is at least length-doubling, i.e., it maps λ bits to l bits, and $l \geq 2\lambda$. F'_K is still a puncturable PRF, and it has inputs of size $n \geq k \geq 3\lambda + 2$ and outputs of size $l \geq 2\lambda$. For inputs chosen from a distribution X with min-entropy $k(\lambda)$ it holds that

$$\{(K, \mathsf{F}'_K(x)) : x \leftarrow X\} \approx_c \{(K, r) : r \leftarrow U_l\}\,.$$

Indeed, $\{(K, \mathsf{F}'_K(x)) : x \leftarrow X\} \approx \{(K, \mathsf{prg}(s)) : s \leftarrow U_\lambda\} \approx_c \{(K, r) : r \leftarrow U_l\}$, where the first indistinguishability holds by statistical closeness of underlying PRF output to random and the second by security of the pseudorandom generator.

Our PRF is sparse due to the fact that the PRG is at least length-doubling and therefore has a sparse image.

□

A.3 Puncturable Deterministic Encryption (PDE).

Puncturable deterministic encryption was first introduced as a building block for deniable encryption by Sahai and Waters [SW14] (called there a hidden sparse trigger mechanism) and then was defined as a primitive by Waters [Wat14]. In this scheme the ciphertext of a message m is computed as $(A = \mathsf{F}_{k1}(m), B = \mathsf{F}_{k2}(A) \oplus m)$, and decryption on input (A, B) outputs $m \leftarrow F_{k2}(A) \oplus B$ if $F_{k1}(m) = A$; otherwise if outputs $\perp$. An important property of this

PDE scheme is that it is iO-friendly[8] and has ciphertexts with pseudorandom properties. This allows us to encode information inside "randomness" by encrypting this information using PDE and then claiming that this ciphertext is true randomness. For a full definition and properties of PDE we refer the reader to [Wat14].

B Augmented Explainability Compiler

In this section we describe a variant of an explainability compiler of [DKR15]. This compiler is used in our construction of NCE, as discussed in the introduction.

Roughly speaking, explainability compiler modifies a randomized program such that it becomes possible, for those who know faking keys, to create fake randomness r_f which is consistent with a given input-output pair. Explainability techniques were first introduced by Sahai and Waters [SW14] as a method to obtain deniability for encryption (there they were called "a hidden sparse trigger meachanism"). Later Dachman-Soled, Katz and Rao [DKR15] generalized these ideas and introduced a notion of explainability compiler.

We modify this primitive for our construction and call it an "augmented explainability compiler". Before giving a formal definition, we briefly describe it here. Such a compiler Comp takes a randomized algorithm $\mathsf{Alg}(input; u)$ with input $input$ and randomness u and outputs three new algorithms:

- $\mathrm{Alg}:\mathrm{r}(input; r)$ is a "rerandomized" version of Alg. Namely, this algorithm first creates fresh randomness u using a PRF and then runs Alg with this fresh randomness.
- $\mathrm{Alg}:\mathrm{td}(input; r)$ is a "trapdoored" version of $\mathrm{Alg}:\mathrm{r}$, which allows to create randomness consistent with a given $output$: namely, before executing $\mathrm{Alg}:\mathrm{r}$, $\mathrm{Alg}:\mathrm{td}$ interprets its randomness r s a PDE ciphertext and tries to decrypt it. If it succeeds and r encrypts an instruction to output $output$, then $\mathrm{Alg}:\mathrm{td}$ complies. Otherwise it runs $\mathrm{Alg}:\mathrm{r}$.
- $\mathsf{Explain}(input, output)$ outputs randomness for algorithm $\mathrm{Alg}:\mathrm{td}$ consistent with given $input$ and $output$. It uses PDE to encrypt an instruction to output $output$ on an input $input$, and outputs the resulting ciphertext.

Definition 7. *An augmented explainability compiler* Comp *is an algorithm which takes as input algorithm* Alg *and random coins and outputs programs* $\mathsf{P}_{\mathrm{Alg:r}}, \mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}$*, such that the following properties hold:*

- **Indistinguishability of the source of the output.** *For any $input$ it holds that*

$$\{(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output) : r \leftarrow U, output \leftarrow \mathsf{Alg}(input; r)\}$$

  and

$$\{(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output) : r \leftarrow U, output \leftarrow \mathsf{P}_{\mathrm{Alg:td}}(input; r)\}$$

  are indistinguishable.

- **Indistinguishability of programs with and without a trapdoor.** $\mathsf{P}_{\mathrm{Alg:r}}$ *and* $\mathsf{P}_{\mathrm{Alg:td}}$ *are indistinguishable.*
- **Selective explainability.** *Any PPT adversary has only negligible advantage in winning the following game:*
  1. *Adv fixes an input $input^*$;*
  2. *The challenger runs* $\mathsf{P}_{\mathrm{Alg:r}}, \mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}} \leftarrow \mathsf{Comp}(\mathsf{Alg})$*;*
  3. *The challenger chooses random r^* and computes $output^* \leftarrow \mathsf{P}_{\mathrm{Alg:td}}(input^*; r^*)$;*
  4. *The challenger chooses random ρ and computes fake $r_f^* \leftarrow \mathsf{P}_{\mathsf{Explain}}(input^*, output^*; \rho)$*

[8]Namely, there is one-to-one correspondence between ciphertexts and plaintexts. This allows to puncture a program at a certain message m, but at the same time preserve its functionality by hardwiring *only a sinlge* (m, c) pair into the program.

5. *The challenger chooses random bit b. If* $b = 0$*, it shows* $(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output^*, r^*)$*, else it shows* $(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output^*, r^*_f)$

6. *Adv outputs* b' *and wins if* $b = b'$

Differences between [DKR15] compiler and our construction. For the reader familiar with [SW14], [DKR15], we briefly describe the differences.

First, we split compiling procedure into two parts: the first part, rerandomization, adds a PRF to the program Alg, such that the program uses randomness $\mathsf{F}(input, r)$ instead of r. The second part adds a trapdoor branch to rerandomized program. This is done for a cleaner presentation of the proof.

Second, we slightly change a trapdoor branch activation mechanism: together with faking keys we hardwire an image S of a pseudorandom generator into the program. Whenever this program decrypts fake r, it follows instructions inside r only if these instructions contain a correct preimage of S. This trick allows us to first change S to random and then to indistinguishably "delete" the whole trapdoor branch from the program. Thus it becomes possible to use a program without a trapdoor in the protocol (and only in the proof change it to its trapdoor version), which is crucial for achieving perfect correctness.

Another difference is that our extracting PRF is only computationally extracting, but instead has a sparse image. This sparseness is not directly needed for the compiler, but we still need it in the proof of our NCE scheme; thus we crucially use the exact implementation of the compiler.

Construction. Our explainability compiler is described in Figure 15. It takes as input algorithm Alg and randomness r. It uses r to sample keys Ext (for sparse computationally extracting PRF), f (for PDE), as well as random s, and randomness for $i\mathcal{O}$. It sets $S = \mathsf{prg}(s)$. Then it obfuscates programs $\mathrm{Alg} : \mathrm{r}[\mathsf{Alg}, \mathsf{Ext}]$, $\mathrm{Alg} : \mathrm{td}[\mathsf{Alg}, \mathsf{Ext}, f, S]$, and $\mathsf{Explain}[f, s]$. It outputs these programs.
Theorem 5. *Algorithm* Comp *presented in Figure 15 is an augmented explainability compiler.*

Proof. We need to show indistinguishability of the source of the output, indistinguishability of programs with and without a trapdoor, and selective explainability.

Indistinguishability of the source of the output. We show that for any program Alg, for any $input$ two distributions

$$\{(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output) : r \leftarrow U, output \leftarrow \mathsf{Alg}(input; r)\}$$

and

$$\{(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output) : r \leftarrow U, output \leftarrow \mathsf{P}_{\mathrm{Alg:td}}(input; r)\}$$

are indistinguishable. We show this by a sequence of hybrids:

1. We start with a distribution $\{(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output) : r \leftarrow U, output \leftarrow \mathsf{P}_{\mathrm{Alg:td}}(input; r)\}$

2. In hybrid 1 we show a distribution $\{(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output) : r \leftarrow U, output \leftarrow \mathsf{P}_{\mathrm{Alg:r}}(input; r)\}$. In other words, when $output$ is computed, trapdoor branch in $\mathrm{Alg} : \mathrm{td}$ is omitted.

   This hybrid is statistically close to the previous one, since randomly chosen r is not a valid PDE encryption with overwhelming probability, and therefore the trapdoor check doesn't pass.

3. Next we show a distribution $\{(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output) : r \leftarrow U, output \leftarrow \mathsf{Alg}(input; r)\}$. In other words, we use given randomness r from the input instead of first applying extracting PRF $\mathsf{F}_{\mathsf{Ext}}$ to $(input, r)$.

   Indistinguishability holds by computationally extracting property of a PRF $\mathsf{F}_{\mathsf{Ext}}$. Indeed, given (Ext, val) as a challenge, where val is either truly random or $\mathsf{F}_{\mathsf{Ext}}(input, r)$ for random r and fixed $input$, we can reconstruct the hybrid as follows: first we create $\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}$ ourselves (for these we need to use given Ext and sample other needed keys ourselves), and then compute $output$ as $\mathsf{Alg}(input; val)$.

Explainability compiler Comp.

```
Program Comp(Alg; r)
Inputs: Algorithm Alg, randomness $r$
  1. Use $r$ to sample keys Ext (for sparse computationally extracting PRF), $f$ (for PDE), as well as random $s$ and
     randomness for $i\mathcal{O}$ $r_1, r_2, r_3$.
  2. Set $S \leftarrow \mathsf{prg}(s)$;
  3. Set $\mathsf{P}_{\mathrm{Alg:r}} \leftarrow i\mathcal{O}(\mathrm{Alg:r}[\mathsf{Alg}, \mathsf{Ext}]; r_1)$, $\mathsf{P}_{\mathrm{Alg:td}} \leftarrow i\mathcal{O}(\mathrm{Alg:td}[\mathsf{Alg}, \mathsf{Ext}, f, S]; r_2)$, and $\mathsf{P}_{\mathsf{Explain}} \leftarrow$
     $i\mathcal{O}(\mathsf{Explain}[f, s]; r_3)$.
  4. Output $\mathsf{P}_{\mathrm{Alg:r}}$, $\mathsf{P}_{\mathrm{Alg:td}}$, and $\mathsf{P}_{\mathsf{Explain}}$.
```

Program Alg : r

```
Program Alg : r[Alg, Ext](input; r)
Constants: underlying randomized algorithm $\mathsf{Alg}(input; u)$, a key for sparse computationally extracting prf Ext
Inputs: input $input$, randomness $r$
  1. Create randomness $u \leftarrow \mathsf{F}_{\mathsf{Ext}}(input, r)$;
  2. output $output \leftarrow \mathsf{Alg}(input; u)$
```

Program Alg : td

```
Program Alg : td[Alg, f, Ext, S](input; r)
Constants: underlying randomized algorithm $\mathsf{Alg}(input; u)$, a faking key $f$, a key for sparse computationally ex-
tracting prf Ext, prg image $S$
Inputs: input $input$, randomness $r$
  1. Trapdoor branch:
     (a) decode out $\leftarrow \mathsf{PDE.Dec}_f(r)$; if out $= \perp$ then goto normal branch;
     (b) parse out as $(input', output', s', \tilde{\rho})$. If $input = input'$ and $\mathsf{prg}(s') = S$ then output $output'$ and halt, else
         goto normal branch;
  2. Normal branch:
     (a) Create randomness $u \leftarrow \mathsf{F}_{\mathsf{Ext}}(input, r)$;
     (b) output $output \leftarrow \mathsf{Alg}(input; u)$
```

Program Explain

```
Program Explain[f, s](input, output; $\rho$)
Constants: a faking key $f$, secret $s$, which is a prg preimage of S
Inputs: input and output $(input, output)$, randomness $\rho$
  1. output $r \leftarrow \mathsf{PDE.Enc}_f(input, output, s, \mathsf{prg}(\rho))$
```

Figure 15: Explainability compiler and programs used.

Since r has the size of at least λ, the input to the PRF $(input, r)$ has enough min-entropy, and by theorem 4 (Ext, val) for random val and for PRF value val are indeed indistinguishable.

Indistinguishability of programs with and without a trapdoor. Here we show that $\mathsf{P}_{\mathrm{Alg:td}}$ and $\mathsf{P}_{\mathrm{Alg:r}}$ are indistinguishable. Consider the following hybrids:

1. We start by showing $\mathsf{P}_{\mathrm{Alg:td}} = i\mathcal{O}(\mathrm{Alg} : \mathrm{td}[\mathsf{Alg}, f, \mathsf{Ext}, S])$, where $S \leftarrow \mathsf{prg}(s)$.
2. In the next hybrid we show $\mathsf{P}_{\mathrm{Alg:td}} = i\mathcal{O}(\mathrm{Alg} : \mathrm{td}[\mathsf{Alg}, f, \mathsf{Ext}, S])$, where S is chosen at random.

   Indistinguishability holds by security of a prg.
3. Next we show $\mathsf{P}_{\mathrm{Alg:r}}$; in other words, we delete trapdoor branch from the program.

   Indistinguishability holds by $i\mathcal{O}$. Indeed, with overwhelming probability over the coins of reduction, random S is outside prg image, thus, there is no input $(input; r)$, which results in executing trapdoor branch due to the check $S = \mathsf{prg}(s')$.

Selective indistinguishability of explanations. We show that any PPT adversary has only negligible advantage in winning the following game:

1. Adv fixes input $input^*$;
2. The challenger runs $\mathsf{P}_{\mathrm{Alg:r}}, \mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}} \leftarrow \mathsf{Comp}(\mathsf{Alg})$;
3. The challenger chooses random r^* and computes $output^* \leftarrow \mathsf{P}_{\mathrm{Alg:td}}(input^*; r^*)$;
4. The challenger chooses random ρ and computes fake $r_f^* \leftarrow \mathsf{P}_{\mathsf{Explain}}(input^*, output^*; \rho)$
5. The challenger chooses random bit b. If $b = 0$, it shows $(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output^*, r^*)$, else it shows $(\mathsf{P}_{\mathrm{Alg:td}}, \mathsf{P}_{\mathsf{Explain}}, output^*, r_f^*)$
6. Adv outputs b' and wins if $b = b'$

We show this in a sequence of hybrids $0^b, \ldots, 4^b$, for bit $b = 0, 1$, and then show that hybrids 4^0 and 4^1 are indistinguishable. This proof is very close to the original proof in [SW14], [DKR15], and therefore we only briefly sketch it here.

1. **Hybrid** 0^b**.** We start with running experiment in explainability game for bit b.
2. **Hybrid** 1^b**.** We generate $r_f^* \leftarrow \mathsf{PDE}_f(input^*, output^*, s^*, \tilde{\rho}^*)$, where $\tilde{\rho}^*$ is random instead of $\mathsf{prg}(\rho^*)$; security holds by security of a prg.
3. **Hybrid** 2^b**.** In the next hybrid we show programs $\mathsf{P}_{\mathrm{Alg:td}} = i\mathcal{O}(\mathrm{Alg} : \mathrm{td} : 1), \mathsf{P}_{\mathsf{Explain}} = i\mathcal{O}(\mathsf{Explain} : 1)$ (Figure 16). These programs have hardwired values and punctured keys. Indistinguishability holds by $i\mathcal{O}$, since programs have the same functionality (for the proof of this fact we refer the reader to [SW14].
4. **Hybrid** 3^b**.** In the next hybrid we choose $r1_f^*$ to be a random value instead of $\mathsf{F}_{f_1}(input^*, output^*, s^*, \tilde{\rho}^*)$; we also choose $r2_f^*$ to be a random value instead of $\mathsf{F}_{f_2}(r1_f^*) \oplus (input^*, output^*, s^*, \tilde{\rho}^*)$. Indistinguishability holds by pseudorandomness of punctured points for PRFs $\mathsf{F}_{f_1}, \mathsf{F}_{f_2}$.
5. **Hybrid** 4^b**.** In this hybrid we choose u^* to be random instead of $\mathsf{F}_{\mathsf{Ext}}(input^*, r^*)$. Indistinguishability holds by pseudorandomness of punctured points for PRF $\mathsf{F}_{\mathsf{Ext}}$.

Hybrids 4^0 and 4^1 are the same, since both r^* and r_f^* are chosen at random and are treated in the same manner. Therefore selective explainability holds.

Program $\mathrm{Alg:td:1}$

```
Program Alg : td : $\mathbf{1}[\mathsf{Alg}, \mathbf{f}, \mathsf{Ext}, \mathbf{S}](\mathbf{input}; \mathbf{r})$
Constants: underlying randomized algorithm $\mathsf{Alg}(input; u)$, a faking key $f = f_1, f_2$, a key for sparse computationally extracting prf Ext, prg image $S$
Inputs: input $input$, randomness $r = r1, r2$
  1. Trapdoor branch:
     (a) if $(input, r) = (input^*, r^*)$ or $(input, r) = (input^*, r_f^*)$, then output $output^*$ and halt;
     (b) if $r1 = r1^*$ or $r1 = r1_f^*$ then goto normal branch;
     (c) set out $\leftarrow \mathsf{F}_{f_2\{r1^*\}\{r1_f^*\}}(r1) \oplus r2$
     (d) if out $= (input^*, output^*, s^*, \tilde{\rho}^*)$ then goto normal branch;
     (e) if $\mathsf{F}_{f_1\{input^*, output^*, s^*, \tilde{\rho}^*\}}(\mathsf{out}) \neq r1$ then goto normal branch;
     (f) parse out as $(input', output', s', \tilde{\rho})$. If $input = input'$ and $\mathsf{prg}(s') = S$ then output $output'$ and halt, else goto normal branch;
  2. Normal branch:
     (a) Create randomness $u \leftarrow \mathsf{F}_{\mathsf{Ext}\{input^*, r^*\}\{input^*, r_f^*\}}(input, r)$;
     (b) output $output \leftarrow \mathsf{Alg}(input; u)$
```

Program $\mathsf{Explain}:1$

```
Program Explain : $\mathbf{1}[\mathbf{f}, \mathbf{s}](\mathbf{input}, \mathbf{output}; \rho)$
Constants: a faking key $f$, secret $s$, which is a prg preimage of S
Inputs: input and output $(input, output)$, randomness $\rho$
  1. set $r1 \leftarrow F_{f_1\{input^*, output^*, s^*, \tilde{\rho}^*\}}(input, output, s, \mathsf{prg}(\rho))$
  2. set $r2 \leftarrow F_{f_2\{r1^*, r1_f^*\}}(r1) \oplus (input^*, output^*, s^*, \tilde{\rho}^*)$
  3. output $r = r1, r2$
```

Figure 16: Programs $\mathrm{Alg:td:1}$ and $\mathsf{Explain}:1$, used in the proof of explainability compiler.

□