Fully Deniable Interactive Encryption

Ran Canetti[1], Sunoo Park[2], and Oxana Poburinnaya[3]

[1] Boston University
[2] MIT and Harvard
[3] University of Rochester

Abstract. Deniable encryption (Canetti *et al.*, Crypto 1996) enhances secret communication over public channels, providing the additional guarantee that the secrecy of communication is protected even if the parties are later coerced (or willingly bribed) to expose their entire internal states: plaintexts, keys and randomness. To date, constructions of deniable encryption — and more generally, interactive deniable communication — only address restricted cases where only *one* party is compromised (Sahai and Waters, STOC 2014). The main question — whether deniable communication is at all possible if *both* parties are coerced at once — has remained open.

We resolve this question in the affirmative, presenting a communication protocol that is *fully deniable* under coercion of both parties. Our scheme has three rounds, assumes subexponentially secure indistinguishability obfuscation and one-way functions, and uses a short global reference string that is generated once at system set-up and suffices for an unbounded number of encryptions and decryptions.

Of independent interest, we introduce a new notion called *off-the-record deniability*, which protects parties even when their claimed internal states are inconsistent (a case not covered by prior definitions). Our scheme satisfies both standard deniability and off-the-record deniability.

1 Introduction

The ability to communicate secret information without having any prior shared secrets is a central pillar of modern cryptography [DH76, RSA78, GM84]. However, standard definitions and existing algorithms for secure communication only guarantee security if the parties' local randomness remains hidden. If the parties' secret keys or randomness are exposed, say as a result of coercion or bribery, secrecy is no longer guaranteed. Moreover, the transcript in common encryption and key exchange schemes often "commits" the sender to the plaintext, in that each transcript is consistent with only one plaintext and randomness.

To address this issue, Canetti, Dwork, Naor and Ostrovsky [CDNO96] introduced the notion of *deniable encryption*, which provides a mechanism for preserving the secrecy of communicated plaintexts even in the face of post-communication coercion or bribery.[4] Specifically, deniable encryption (or, more generally, deniable

[4] While our results address both bribery and coercion, bribery might be the better setting to keep in mind. Indeed, protecting against bribery is more challenging in

interactive communication) introduces additional algorithms, called *faking algorithms,* that are not present in standard secure communication definitions. The faking algorithms allow the communicating parties to present fake internal states (including keys and randomness) that make any communication transcript appear consistent with any plaintext of the parties' choice. Concretely, an adversary should not be able to tell whether the sender and the receiver gave it the true keys, randomness, and plaintext, or fake ones.

When the communicating parties have a secret key that was shared ahead of time, deniable encryption can be simple. The classic one-time-pad scheme is perfectly deniable: having sent $c = k \oplus m$, the parties can claim that they sent any plaintext m' by claiming that $k' = c \oplus m'$ is their true key. However, shared-key deniable schemes fail to address the crucial question of how to deniably agree on a shared key in the first place. Indeed, existing key exchange protocols are "committing" in a way that precludes deniability. For instance, in Diffie-Hellman key exchange, there exists only one key consistent with any given transcript, so it is impossible to equivocate a one-time pad key generated using Diffie-Hellman key exchange. Thus, the core question here is how to deniably transmit a value (or equivalently, to establish a shared key) *without any pre-shared secrets.*

This setting turns out to be much more challenging. Even the restricted case where only the *sender* is coerced (or bribed) was fully resolved only much later, assuming indistinguishability obfuscation, in the breakthrough work of Sahai and Waters [SW14].[5] The case where only the *receiver* is coerced or bribed follows from the sender-only case via a general transformation, at the cost of an additional message [CDNO96]; hence, the [SW14] scheme implies a 3-round receiver-deniable protocol. This transformation can also be extended to handle the case where the adversary may coerce either party, *but only one of the two.* Furthermore, as demonstrated by Bendlin, Nielsen, Nordholt, and Orlandi [BNNO11], *any* receiver-deniable encryption protocol must take at least three rounds of communication.

Constructing *bideniable* encryption protocols, namely encryption protocols that guarantee deniability in the unrestricted case where both the sender and the receiver can be simultaneously coerced or bribed, has remained open:

Do there exist bideniable encryption protocols, with any number of rounds?

Bideniability is a significantly stronger property than any of the restricted variants above, where the adversary only learns the internal state of either the sender

that the parties are *incentivized* to disclose all internal state, including all random choices.

[5] Prior to [SW14], we only had the partial solution of [CDNO96], where the adversary's distinguishing advantage decreases linearly with ciphertext size; in particular, to get indistinguishability with negligibly small advantage, one has to send super-polynomially long ciphertexts.

or the receiver.[6] Indeed, when both parties are coerced, the adversary obtains a complete transcript of an execution, including all the random choices, inputs and outputs of both parties. This means that the adversary can now fully run this execution, step by step, and compare it against the recorded communication. Even so, as long as the sender and receiver follow the protocol during the actual exchange of messages, bideniability guarantees that any (real or fake) internal state provided by the parties looks just as plausible as any other (real or fake) one.

Off-the-record deniability. When the attacker bribes or coerces *both* parties, a new concern emerges: what happens if the plaintext claimed by the sender is different from that claimed by the receiver? This could arise in various scenarios: the parties might simply not have the chance to coordinate a story in advance (e.g., if they are separated and interrogated); or the parties might be incentivized to tell different stories (e.g., to protect themselves or those close to them); or the parties might find themselves incentivized to "defect" on each other as in a prisoner's dilemma. Still, standard bideniability (as defined by [CDNO96]) provides no guarantees for these cases.

1.1 Our Contributions

Our first contribution is defining a security guarantee, called *off-the-record deniability*, that holds even in the above setting, where the coerced (or bribed) parties' responses are inconsistent with each other.[7] Off-the-record deniability achieves protection akin to an ideal, physically protected communication channel where the communication leaves no trace behind. That is, off-the-record deniability guarantees that the communication transcript does not help the attacker to determine which of the two parties is telling the truth, if any. This holds even if the parties deviate from the protocol —- as long as the deviation happens after the actual protocol execution completes. In other words, off-the-record deniability guarantees protection for each party *independently of the other party's actions.* This contrasts with standard bideniability where, for the security guarantee to hold, *both parties must lie, and their claims must be consistent.*

Our second and main contribution is the first encryption protocol that is both bideniable and off-the-record deniable. We call such protocols *fully deniable.* A fully deniable protocol provides protection akin to an ideal channel, in that after message transmission, each party can claim that the message was any value whatsoever (say, from a pre-specified domain) and the attacker has no way to tell which party, if any, is telling the truth.

We stress that, prior to this work, even the existence of bideniable encryption (without the additional off-the-record property) was an open question.

[6] We note that a related but different concept, *multi-distributional bideniability*, has previously been considered in a setting where both parties are attacked [OPW11]; see Section 1.4 and the full version of this paper [CPP18] for more details.

[7] The *off-the-record messaging protocol* [BGB04] is a messaging protocol that shares our motivation of enabling encrypted communications as close as possible to an ideal private channel, but is otherwise unrelated to our off-the-record deniability notion.

Theorem 1. *Assuming subexponentially secure indistinguishability obfuscation and subexponentially secure one-way functions, there exists a 3-message interactive bit encryption scheme that is* ***fully deniable*** *(i.e., both* ***bideniable*** *and* ***off-the-record-deniable****) in the common reference string model. In addition, the receiver's deniability is* public *(i.e., the true random coins of the receiver are not required to compute fake randomness of the receiver).*

Our common reference string (CRS) consists of six obfuscated programs: three for the sender (programs P1 and P3 for generating the first and third messages, respectively, and the sender faking program SFake) and three for the receiver (programs P2 for generating the second message, the decryption program Dec, and the receiver faking program RFake). The scheme instructs the parties to run the obfuscated programs on their relevant inputs and uniformly chosen random inputs.

Designing the scheme requires addressing two main classes of challenges. First, the operation of the six programs should follow a certain "internal logic" - which ends up being necessary even in an idealized model where parties only have *oracle access* to the encryption scheme programs. We make this logic explicit by constructing and proving security of a fully deniable encryption scheme in this idealized model. While this construction is not used directly in the full-fledged scheme, it highlights key design difficulties. Interestingly, while many cryptographic primitives are trivial to construct in this idealized model, deniable encryption is still highly non-trivial; indeed, *our technical overview (section 2) is fully devoted to building deniable encryption in the idealized model.*

The next challenge lies in translating this idealized protocol to one that is provably secure when the programs are (a) actual programs and (b) protected only by indistinguishability obfuscation (IO), not ideal obfuscation. Here, we use the sophisticated tools developed in [KLW15, CHJV14, BPR15, BPW16] that were developed for dealing with obfuscated programs that are designed to be repeatedly run on inputs that were generated earlier by the program itself. Our situation is however significantly more complex: We have several programs that are designed to take inputs from each other with a specific and context-dependent set of constraints. We thus develop additional tools and abstractions that allow us to deal with this more complicated setting.

We now turn to discussing our definitions and constructions in mor detail.

1.2 Fully Deniable Interactive Encryption: Definition in a Nutshell

Deniable interactive encryption is equipped with algorithms to generate protocol messages, to decrypt, and to generate fake randomness. We present the definition for the three-message case, since our protocol has three messages.

We start with syntax. A scheme consists of six programs: P1, P2, P3, Dec, SFake, and RFake. Program P1, run by the sender, takes as input a message m and sender random string s, and outputs a first message μ_1. Program P2, run by the receiver, takes as input a message μ_1 and receiver random string r, and outputs second message μ_2. Program $\mathsf{P3}$, run by the sender, takes s, m, μ_1, μ_2 and outputs a third message μ_3. Program Dec, run by the receiver, takes input r, μ_1, μ_2, μ_3 and outputs plaintext $\tilde{m}$. Program SFake takes as input the public transcript of the protocol (namely messages μ_1, μ_2, μ_3), the sender randomness s, the message m, a fake message m', and potentially some additional random input ρ_S, and outputs a fake random string $s_{m'}$ that is intended to explain the transcript as an encryption of m'. Program RFake takes as input the public transcript, the receiver randomness r, the message m, a fake message m', and potentially some additional random input ρ_R, and outputs a fake random string $r_{m'}$ that is intended to explain the transcript as decrypting to m'.

We define *correctness* in the natural way: if the sender runs $\mathsf{P1}, \mathsf{P3}$ with plaintext m and uniformly chosen s, and the receiver runs $\mathsf{P2}, \mathsf{Dec}$ with uniformly chosen r, the receiver must decrypt $\tilde{m} = m$ except with negligible probability.

Bideniability requires that no PPT adversary can distinguish the following two distributions: (1) a protocol transcript for plaintext m', and both parties' true random coins for that transcript; and (2) a protocol transcript for plaintext m and fake random coins which make that transcript decrypt to m'. That is,

$$(\mathsf{tr}(s, r, m'), s, r) \approx_c (\mathsf{tr}(s, r, m), s_{m'}, r_{m'}) \ , \tag{1}$$

where s, r are uniformly random, $\mathsf{tr}(s, r, m)$ is the transcript from running the protocol to transmit m with random inputs s for the sender and r for the receiver, $s_{m'} = \mathsf{SFake}(s, m, m', \mathsf{tr}(s, r, m); \rho_S), r_{m'} = \mathsf{RFake}(r, m, m', \mathsf{tr}(s, r, m); \rho_R)$, and $\approx_c$ denotes computational indistinguishability.

Off-the-record deniability requires that no PPT adversary can distinguish between the following three cases:

– **The sender tells the truth** and **the receiver lies**. That is, the adversary sees a transcript for plaintext m, the sender's true random coins, and fake random coins from the receiver consistent with m'.
– **The sender lies** and **the receiver tells the truth**. That is, the adversary sees a transcript for plaintext m', fake random coins from the sender consistent with m, and the receiver's true random coins.
– **Both the sender and the receiver lie**. That is, the adversary sees a transcript for plaintext m'', fake random coins from the sender consistent with m, and fake random coins from the receiver consistent with m'.

That is,

$$(\mathsf{tr}(s, r, m), s, r_{m'}) \approx_c (\mathsf{tr}(s, r, m'), s_m, r) \approx_c (\mathsf{tr}(s, r, m''), s_m, r_{m'}) \ , \tag{2}$$

where s, r, tr are defined as in (1), and $s_m, r_{m'}$ are fake coins produced by running faking algorithms on the corresponding transcript.

Observe that bideniability implies that $\mathsf{tr}(s, r, m) \approx_c \mathsf{tr}(s, r, m')$, so a bideniable scheme is also semantically secure. Similarly, off-the-record deniable schemes are also semantically secure.

Full deniability. A scheme is **fully deniable** if it is both bideniable and off-the-record deniable. Full deniability provides protection akin to an ideal secure channel in that the parties can freely claim any plaintext was sent or received, and which guarantees protection even when parties' claims do not match.

1.3 A Very Brief Overview of the Construction

Our starting point is an elegant technique from [SW14] that transforms any randomized algorithm A (with domain X and range Y) into a "deniable version" using IO. The technique creates two obfuscated programs A' and F, where: A' is the "deniable version" of A; and F is a "faking algorithm" that, for any input $(x,y) \in X \times Y$, outputs randomness ρ such that $A'(x;\rho) = y$. Using this technique, for any protocol, we can equip parties with a way to "explain" any given protocol message sent: that is, to produce fake randomness which makes that protocol message consistent with any plaintext of the parties' choice.

Based on this, a first attempt at a bideniable scheme might be to apply the [SW14] technique to an arbitrary public-key encryption scheme to create obfuscated programs for encryption, decryption, sender-fake and receiver-fake — and then use the sender-fake and receiver-fake programs to "explain" the protocol messages one by one. However, this does not yield a bideniable encryption scheme: the [SW14] technique is guaranteed to work only when applied to independent algorithm executions, but here the algorithms are run on the same keys and randomness, protocol messages are interrelated, and any convincing overall explanation must consist of a sequence of *consistent* explanations across the algorithms.[8] The problem in a nutshell is that although the [SW14] technique could create a deniable version of any single program, applying the technique separately to the key generation, encryption, and decryption programs fails to achieve deniability with respect to the programs' *joint* behavior.

More concretely, it is problematic that the adversary can manipulate any given transcript and randomness to generate certain "related" transcripts and randomness, and then try running the decryption algorithm on different combinations of them. Next, we give some intuition as to why this is a problem.

The Accumulating Attack. A fake r (i.e., receiver randomness) can be viewed as a string which "encodes" or "remembers", explicitly or implicitly, an instruction to decrypt a certain transcript to a certain fake plaintext. An adversary can run RFake iteratively on a given r (and a series of related transcripts) to successively obtain $r_1, r_2, \ldots$, hoping that each new application of RFake will add a new (ith) instruction into the "memory" of r_i in addition to all the preceding instructions. Since r_i is a bounded-length string which, information-theoretically, can carry only a fixed amount of information, sooner or later, one of the instructions will be lost from the "memory" of r_{i^*} for some i^*. It can be shown that, assuming r was fake, an adversary running RFake many times can obtain some r_i which does not carry the original r's instruction, and thus decrypts the original transcript

[8] Indeed, if this approach worked, it would yield two-message bideniable encryption, which is impossible [BNNO11].

honestly. (This attack first appeared in [BNNO11] in the two-message setting and was used to demonstrate impossibility of two-message receiver-deniable encryption.)

While the above attack does not carry over to the three-message case generically, it stills remains valid for many protocols: namely, for those protocols where it is easy, given the challenge transcript tr^*, to find transcripts "related" to tr^*. Here "related" means that these transcripts can be successfully decrypted using the same true randomness r that was used to generate tr^*. (In particular, in the two-message case, it is always easy to generate related transcripts (pk, c) by setting $pk = pk^*$ and setting c to be a fresh ciphertext with respect to pk.)

Therefore our approach, based on the above ideas, involves: (1) designing a protocol that prevents the adversary from computing related transcripts that force receiver randomness to "accumulate" information as described above, and then (2) applying the [SW14] technique to the algorithms for generating each message of this protocol. For the first step, we design such a protocol in the *oracle-access model*, where everyone (parties and adversaries) has only oracle access to the programs for computing protocol messages. Then, we adapt the construction to the setting where everyone has access to program code, obfuscated under IO.

Step 1 of our plan — designing a protocol resistant to the Accumulating Attack — itself consists of two key steps, further detailed below: (1a) design a "Base Protocol" that resists only some attacks, then (1b) augment the base protocol using the ideas of a ***level system*** and ***comparison-based decryption***, to obtain a protocol secure in the oracle-access model (the "Idealized Protocol").

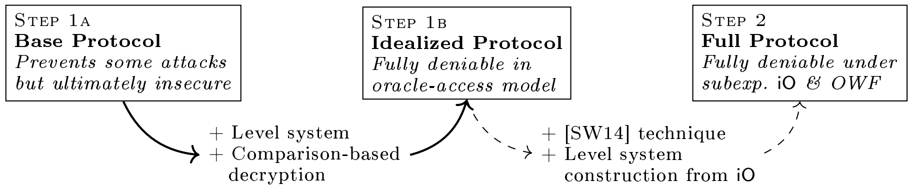

STEP 1A
Base Protocol
Prevents some attacks but ultimately insecure
STEP 1B
Idealized Protocol
Fully deniable in oracle-access model
STEP 2
Full Protocol
Fully deniable under subexp. iO & OWF
+ Level system
+ Comparison-based decryption
+ [SW14] technique
+ Level system construction from iO

Fig. 1: ***The construction, step by step.*** *The second arrow is dashed because while* conceptually *the Idealized Protocol is a stepping-stone to the Full Protocol,* technically *the Full Protocol requires very different techniques, and must be proven from scratch rather than "building on" the Idealized Protocol.*

STEP 1A: We design the Base Protocol in the oracle-access model as follows. The first message μ_1 is a PRF output for input (s, m) where s is the sender randomness s and m is the plaintext. The second message μ_2 is a PRF output for input (r, μ_1) where r is the receiver randomness r. The third message μ_3 is an encryption of (m, μ_1, μ_2). All keys for PRFs and encryption are hidden inside these programs (and not known to anyone, including parties). After exchanging μ_1, μ_2, μ_3 with the sender, the receiver runs Dec, which decrypts the ciphertext

μ_3 and outputs m. In addition, the programs contain certain consistency checks: Dec returns an output only if it gets the correct r (i.e., consistent with μ_2), and $\mathsf{P3}$ only returns an output if it gets the correct s (i.e., consistent with μ_1).

The intuition for this design is as follows. The first two messages serve as "hashes" of the parties' internal state so far, and the next two programs — $\mathsf{P3}$ and Dec — produce output only if the parties "prove" to the programs (by giving randomness consistent with these "hashes") that they are continuing to execute the protocol on the same inputs used to generate these hashes. This design aims to prevent the adversary from computing related transcripts (and thus prevent the Accumulating Attack): for instance, an adversary must not be able to reuse μ_1, μ_2 from a transcript (μ_1, μ_2, μ_3) to compute a new $\mu_3{}'$ such that $(\mu_1, \mu_2, \mu_3{}')$ is also a valid transcript with respect to the same r. Section 2 gives more intuition about this.

Step 1b: Unfortunately, the intuition from Step 1a is only partially correct: it turns out that it is still possible to generate related transcripts, although the design above indeed protects against "most" ways of generating them. Concretely, we describe a method Ω (detailed in Section 2.1) to compute a series of related transcripts differing only in the third message. Importantly, Ω is generic: it works for *any* three-message bideniable encryption scheme. Ω takes any transcript (μ_1, μ_2, μ_3) and, applied iteratively, produces a "chain" of valid transcripts $\mathsf{tr}_1 = (\mu_1, \mu_2, \mu_3{}^{(1)}), \mathsf{tr}_2 = (\mu_1, \mu_2, \mu_3{}^{(2)})$, and so on. However, the scheme from Step 1a importantly ensures that Ω is the *only* way to compute valid related transcripts: this is crucial for the security proof.

It remains to ensure that the adversary cannot learn the true plaintext from the chain of related transcripts produced using Ω (e.g., by performing the Accumulating Attack). To do this, we augment the Base Protocol with a *level system*, under which each $\mu_3{}^{(i)}$, generated using Ω, encodes a number which we call a *level*, which is set to that transcript's own *index* i.[9] Concretely, $\mu_3{}^{(i)}$ is an encryption of (m, μ_1, μ_2, i). Additionally, any fake randomness r_i — generated by running RFake on $(\mu_1, \mu_2, \mu_3{}^{(i)})$ — also encodes the level i of the transcript used to generate this r_i. The level i is encrypted, and so hidden from parties and the adversary, but the programs can decrypt and learn i using their internal keys. To complete the Idealized Protocol, we modify the decryption algorithm such that any fake r_i associated with level i may be used to decrypt transcripts with $\mu_3{}^{(j)}$ where $j > i$ ("correctness forward"), but decryption will fail (i.e., output $\perp$) if attempted with respect to r_i and $\mu_3{}^{(j)}$ where $j < i$ ("oblivious past"). We call this *comparison-based decryption behavior*.

The Idealized Protocol, just described, is fully deniable in the oracle-access model. In particular, it prevents the Accumulating Attack: intuitively, this is because comparison-based decryption ensures that an iteratively faked r only

[9] Since Ω is inherently applied sequentially, the index i of each transcript produced by Ω is well defined.

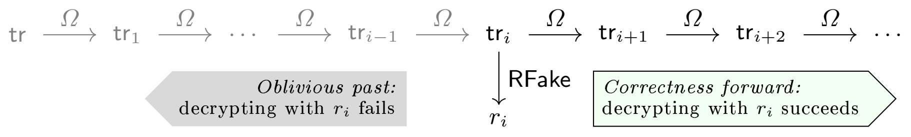

tr
Ω
tr$_1$
$\cdots$
tr$_{i-1}$
tr$_i$
tr$_{i+1}$
tr$_{i+2}$
RFake
r_i
Oblivious past: decrypting with r_i fails
Correctness forward: decrypting with r_i succeeds

Fig. 2: Comparison-based decryption behavior

encodes the most recent faked plaintext, rather than accumulating a sequence of past fake plaintexts.

Step 2: We obtain the Full Protocol by applying the [SW14] technique to the Idealized Protocol, which enables the parties to use *obfuscated programs* (not oracle access) to compute protocol messages and to generate fake randomness. Proving security of the resulting protocol based on IO presents a number of challenges. To start with, the security argument in the oracle-access model relies heavily on certain outputs of programs being hard to find provided the corresponding inputs are hard to find. To make the analogous argument with respect to IO, we need to show that such inputs *don't exist* (rather than being hard to find). Furthermore, as part of our construction, we introduce and construct a special primitive that could be called "deterministic order-revealing encryption": a variant of deterministic encryption where $\mathsf{Enc}(0)$ and $\mathsf{Enc}(1)$ must be indistinguishable, even given programs which homomorphically increment ciphertexts (producing $\mathsf{Enc}(2), \mathsf{Enc}(3)$ and so on up to some superpolynomial bound) *and* homomorphically compare them. (Intuitively, homomorphic comparison enables the comparison-based decryption behavior; see section 2). To argue security of this special deterministic encryption, we employ different primitives and techniques from the literature, including the *asymmetrically contrained encryption* from [CHJV14], and the proof techniques from [BPR15] to argue unreachability of the end of a superpolynomially-long chain.

This concludes the brief overview of our scheme. An in-depth technical overview of the scheme, including the intuition for the construction and the proof, can be found in section 2 (technical overview). Impatient readers may wish to jump ahead to the Idealized Protocol program descriptions in figures 6 and 7 or refer to the Full Protocol in the full version of this paper [CPP18].

1.4 Variants of Deniable Encryption and Other Related Concepts

Next, we overview some variants of deniable encryption, deniable communication, and surrounding concepts. While they are not directly relevant to this work, clarifying these concepts may prevent confusion.

- **Post-execution vs. adaptive coercion.** This paper considers coercion that happens after protocol execution. A broader definition, *adaptive* coercion, would capture coercion at some (arbitrary) point during the protocol execution (with uncoerced parties possibly unaware of the coercion).
- **Private vs. public deniability.** The deniability of the sender (or receiver, or both) is *public* [SW14] if the corresponding faking algorithm does not require the true randomness or the true plaintext as input. Our scheme has

public receiver deniability (our RFake has syntax $\mathsf{RFake}(m', \mathsf{tr}; \rho_R)$). This means that anyone, not just the receiver, can produce fake random coins for the receiver. Note that any publicly deniable faking algorithm must be randomized: otherwise, the adversary could easily check if a claimed r is fake by comparing it to $\mathsf{RFake}(m', \mathsf{tr})$.

– **"Coordinated" schemes.** One can also consider "coordinated" schemes [OPW11] where a single faking algorithm takes as input the true coins of *both the sender and the receiver* at the same time. Such schemes require coordination between the sender and the receiver in order to compute fake randomness. Our scheme does not require coordination, but we note that prior to this work, even coordinated fully bideniable schemes were not known.

Deniable encryption is related to a number of other cryptographic concepts:

– **Incoercible key exchange** is equivalent to deniable encryption. The former can be used to establish deniable one-time-pad keys for encryption. The latter enables a sender to pick a random key and transmit it deniably to a receiver.
– **Flexible deniability.** [CDNO96] also proposed a weaker deniability notion, variously called *flexible deniability*, *multi-distributional deniability* ([OPW11, BNNO11, Dac12, AFL16, CIO16]), or *dual-scheme deniability* ([GKW17]). In a nutshell, this notion considers a setting where the coercer does not know which scheme is actually in use, and the coerced party has the freedom to "lie" in an undetectable way regarding the scheme that was actually used. (Equivalently, this notion assumes that the coercer does not expect to see some of the randomness used by the coerced party.) We note that none of the schemes in [OPW11, BNNO11, Dac12, AFL16, CIO16] are deniable in a setting where the coercer knows the scheme used in full and expects to see all the random coins of the coerced party.
– **Non-committing (adaptively secure) encryption (NCE, [CFGN96])** is weaker than deniable encryption, and designed for a different purpose. NCE requires that a simulator can generate dummy ciphertexts that can later be opened to any plaintext. The differences with deniable encryption are twofold. First, deniable encryption enables faking of real ciphertexts (that carry plaintexts), while NCE ciphertexts can *either* be faked (if simulated) *or* carry a plaintext (if real). Thus, in NCE, parties cannot fake; only the simulator can. Secondly, fake opening on behalf of all parties in NCE is done by the same entity, the simulator, while in deniable encryption the sender and the receiver fake independently of each other.
Bideniable encryption is strictly stronger than NCE: any bideniable encryption is also an NCE [CDNO96], but two-message NCE (e.g., [CDMW09]) is provably not bideniable, due to the three-message lower bound of [BNNO11].
– **Deniable authentication.** Deniable encryption is incomparable to deniable authentication. Deniable authentication allows the receiver of a message to authenticate the message's origin and contents, while preventing the receiver from convincing a third party who did not directly witness the communication that the message came from the sender (see, e.g., [DKSW09]). In contrast, in deniable encryption, the third party (adversary) may directly witness the

communicated ciphertext and learn whether the parties have communicated with each other. The goal of deniable encryption is not to hide *whether* a party participated in a communication, but rather to preserve secrecy of the communication *contents* — even when parties are coerced (separately or jointly) to reveal their internal secrets.

1.5 Prior Work on Deniable Encryption

The definition of deniable encryption was introduced in 1996 by [CDNO96]. However, the techniques of that time fell short of achieving deniability: in fact, [CDNO96] presented a construction where the distinguishing advantage between real and fake opening was inversely proportional to the length of the ciphertext, thus requiring superpolynomially long ciphertexts in order to achieve cryptographic deniability. It was not until 2014 that Sahai and Waters presented the first (and, to date, the only) construction of sender-deniable encryption [SW14]. Their construction is based on indistinguishability obfuscation.

The [SW14] scheme can be transformed into a three-message *receiver*-deniable protocol using a generic transformation from sender- to receiver-deniable encryption (due to [CDNO96]) at the cost of one additional round, as follows: the *receiver* first deniably sends a random bit b to the *sender* deniably using the sender-deniable protocol, then the sender sends $b \oplus m$ to the receiver in the final round. Furthermore, if the sender sends $b \oplus m$ using the sender-deniable protocol rather than in the clear, the resulting scheme will be *sender-or-receiver*-deniable: that is, deniable against adversaries that coerce either one but not both of the parties. This final step incurs no additional rounds if (as in [SW14]) the message needs not be decided until the last round of the sender-deniable protocol. However, all these constructions rely heavily on the fact one of the parties' internal states remains hidden, and therefore fail to achieve bideniability.

Several prior works have focused on proving lower bounds for deniable encryption. [CDNO96] showed that a certain class of schemes cannot achieve better distinguishing advantage than inverse polynomial. [Dac12] extended this result to a broader class of constructions, showing that the same holds for *any* black-box construction of sender-deniable encryption from simulatable encryption. [Nie02] showed that any non-committing encryption, including bideniable encryption, can only reuse its public key an *a priori* bounded number of times; and therefore deniable communication must be interactive, even if two messages. Using different techniques, [BNNO11] showed that two-message receiver-deniable schemes, and hence also bideniable schemes, do not exist.

1.6 Organization of the Paper

The rest of the paper is organized as follows. Section 2 gives an informal yet almost complete description of the scheme, and outlines the main proof steps. Due to space constraints, the rest of the paper (definitions, the level system, full description of the deniable encryption scheme both in oracle-access and the CRS model, and formal proofs) is in the full version [CPP18].

2 Towards the Scheme: Technical Overview

This section provides an informal yet almost complete overview of our construction. The primary purpose of this section is to guide the reader through the process of designing the scheme, outlining concrete attacks and corresponding protection mechanisms. This should be helpful for readers who want to gain some intuition about the scheme and its security, but are not willing to read the whole 250-page full version [CPP18], and for readers seeking to design a scheme from weaker assumptions (several issues described in this overview inhere in any 3-round deniable encryption, and could arise in schemes with more rounds too).

In this overview we describe the scheme in the oracle-access model. That is, we assume that all parties and the adversary have oracle access to programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}$ (which generate the three messages of the protocol), decryption program Dec, and faking programs $\mathsf{SFake}, \mathsf{RFake}$.

We build our scheme in two main steps. As a first attempt, we try to avoid the known attacks on the 2-message case by considering a 3-message scheme. Next, we discuss some attacks and augment our scheme with levels and comparison-based decryption behavior, which yields our final scheme.

2.1 A First Attempt

Recall the [SW14] technique, mentioned above, that transforms any algorithm into a deniable version using indistinguishability obfuscation. Given this, a natural attempt to build deniable encryption is to take any (2-message) public-key encryption scheme and use the [SW14] technique to make each of its algorithms Gen, Enc, and Dec deniable. Concretely, the [SW14] technique takes any randomized algorithm A (with domain X and range Y) and outputs two obfuscated programs A' and F, where: A' is the "deniable version" of A; and F is a "faking algorithm" that, for any input $(x, y) \in X \times Y$, outputs randomness ρ such that $A'(x; \rho) = y$. Using this technique, we can take any protocol and equip parties with a way to "explain" any given protocol message they send: that is, to produce fake randomness which makes that protocol message consistent with any plaintext of the parties' choice.

This approach would allow, for example, the receiver to create a fake sk' decrypting a given ciphertext c to any plaintext of its choice. This sk' would even be indistinguishable from the real sk, to an adversary that only sees the purported secret key. But the problem is that the adversary sees other related information: e.g., it has the public key, so can run the encryption algorithm and generate outputs related to sk. The [SW14] technique does not work when applied to multiple programs with interrelated outputs: such as Gen, Enc and Dec.

Let us now outline the result of [BNNO11]: impossibility of bideniable encryption in 2 messages. This will give us insight on how to construct a 3-message bideniable encryption scheme while "avoiding" the impossibility. Also, the [BNNO11] result yields a concrete attack on the first-attempt scheme outlined above.

Impossibility of the 2-message case ([BNNO11]). [BNNO11] shows that even receiver-deniable (as opposed to bideniable) schemes are impossible with two messages. Their result is unconditional. Their proof shows that any 2-message receiver-deniable encryption scheme, even for a single-bit plaintext, can be used to deniably send any polynomial number of plaintexts, simply by reusing the first message (pk) and sending multiple second messages $c_1, \ldots, c_N$ (where N is arbitrarily, but polynomially, large); then they show that all these ciphertexts can be faked *simultaneously* using a *single* fake decryption key. This implies a method for compressing an arbitrary string beyond what is information-theoretically possible, as follows. To compress a string $b_1, \ldots, b_N$ from N bits (where N is larger than $|\mathsf{sk}|$) to $|\mathsf{sk}|$ bits: (1) prepare N encryptions of 0 under a single pk (call them $c_1, \ldots, c_N$);[10] (2) compute $\mathsf{sk}^{(1)} \leftarrow \mathsf{RFake}(\mathsf{sk}, c_1, b_1)$, $\mathsf{sk}^{(2)} \leftarrow \mathsf{RFake}(\mathsf{sk}^{(1)}, c_2, b_2), \ldots, \mathsf{sk}^{(N)} \leftarrow \mathsf{RFake}(\mathsf{sk}^{(N-1)}, c_N, b_N)$. The final string $\mathsf{sk}^{(N)}$ is a compressed description of $b_1, \ldots, b_N$, since it is shorter than N and since the original string can be recovered by decrypting each b_i as $\mathsf{Dec}(sk^{(N)}, c_i)$. Since most strings cannot be compressed, we have a contradiction.

Stated differently, this impossibility says that a secret key which was faked multiple times to lie about different ciphertexts has to "remember" or store information about each lie; but information-theoretically, it cannot remember more information than its length allows. Thus, at some point, such a secret key has to "forget" previous lies, and then it can be used to decrypt the original ciphertext to its real plaintext. That is, there is always an attack on any 2-message scheme, which roughly goes as follows. Assume the adversary sees a ciphertext c, claimed to encrypt plaintext m', together with a fake sk' that decrypts c to m'; but in reality, c encrypts m. The adversary can generate $N > |\mathsf{sk}|$ ciphertexts $c_1, \ldots, c_N$ as above, and run RFake iteratively to compute $\mathsf{sk}^{(N)}$ as above, and then compute $\mathsf{Dec}(\mathsf{sk}^{(N)}; c) = m$ to learn the true plaintext.

In summary, the core issue with the 2-message schemes is that for a single receiver message (i.e., pk) it is possible to efficiently generate many different sender messages (i.e., ciphertexts), such that all these ciphertexts are valid ciphertexts with respect to the same receiver key; this, in turn, means we must be able to use a single secret key to fake all the ciphertexts at once, which is information-theoretically impossible.

What would be the analogous argument in the 3-message case? Consider a 3-message scheme with messages (μ_1, μ_2, μ_3). If the scheme has the property that, given a receiver message μ_2, one can efficiently generate many different sender messages ${\mu_1}^{(i)}, {\mu_3}^{(i)}$ yielding valid transcripts $({\mu_1}^{(i)}, \mu_2, {\mu_3}^{(i)})$, then the scheme is subject to the [BNNO11] impossibility. For example, consider a 3-message scheme where the third message is a fresh encryption under freshly sampled random coins: this enables generating many third messages ${\mu_3}^{(i)}$ for any given μ_1, μ_2, and applying the [BNNO11] argument shows that any fake receiver randomness must remember a lie for each ${\mu_3}^{(i)}$, so this scheme is susceptible to the same attack as two-message schemes.

[10] These ciphertexts do not depend on the string to be compressed and thus can be thought of as public parameters of the compression protocol.

Base Protocol. Now we present our Base Protocol, which is insecure but will be augmented later to achieve a secure version. The scheme has parties first exchange two PRF values, then has the sender encrypt its plaintext m into a ciphertext μ_3 using program $\mathsf{P3}$, which the receiver can decrypt using program Dec. Before presenting the scheme formally, we give motivation for the design.

With the [BNNO11] impossibility in mind, a natural approach to building a 3-message scheme is to ensure that for any given first two messages μ_1, μ_2, only one consistent third message μ_3 can be efficiently computed. The Base Protocol achieves this using the following ideas.

1. The first message μ_1 "commits" to the sender's coins s and message m.
2. The third message μ_3 is a deterministic, symmetric-key encryption of m under a key K that is hardwired in programs $\mathsf{P3}$ and Dec and is unknown to parties.
3. $\mathsf{P3}(s, m, \mu_1, \mu_2)$ does a validity check before its output: if μ_1 is indeed a "commitment" to s and m, $\mathsf{P3}$ outputs μ_3; otherwise, it outputs $\perp$.

In other words, the only way for the sender to generate a valid μ_3 is to "prove" to $\mathsf{P3}$ that it is running $\mathsf{P3}$ on the same s, m used to compute μ_1. Thus, as long as K remains secret and the ciphertexts are sufficiently sparse, for any μ_1, μ_2, there is only one (efficiently computable) consistent μ_3.

So far, since μ_3 is computed under the same key K in each execution and it is not randomized, all executions with the same m yield the same μ_3, which is clearly insecure. To fix this, we let μ_3 encrypt not only m, but the first two messages μ_1, μ_2 as well, forcing different executions to have different third messages.

We have not yet discussed how the second message μ_2 should be computed, which actually depends on an extension of the attack based on [BNNO11], described above. Recall that we wanted it to be hard to compute multiple transcripts with the same μ_2: say, $({\mu_1}^{(i)}, \mu_2, {\mu_3}^{(i)})$. In fact, we also want it to be hard to convert a transcript (μ_1, μ_2, μ_3) with receiver randomness r into a different transcript $({\mu_1}', {\mu_2}', {\mu_3}')$ consistent with the same r, since it is possible to extend the attack to this case as well. With this in mind, we design the protocol as follows.

1. The second message μ_2 is a pseudorandom function output $\mathsf{PRF}(r, \mu_1)$, for a PRF key that is hardwired into $\mathsf{P2}$ and Dec and not known to the parties.[11] The PRF inputs are the receiver randomness r and the first message μ_1.
2. $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$ does a validity check before decryption: if μ_2 is the correct PRF output for input (r, μ_1), Dec outputs m; otherwise, it outputs $\perp$.

Thus, the only way for the receiver to decrypt is to "prove" to Dec that it is running Dec on a valid r (consistent with μ_2). This ensures that it is hard to transform a transcript (μ_1, μ_2, μ_3) into a different $({\mu_1}', {\mu_2}', {\mu_3}')$ consistent with the same receiver randomness r, since that would require finding ${\mu_1}', {\mu_2}'$ such that ${\mu_2}' = \mathsf{PRF}(r, {\mu_1}')$, for an unknown r and an unknown PRF key.

[11] In this high-level description we omit PRF keys to simplify notation.

We conclude this protocol design with a couple of final notes. First, we instantiate our "commitment" using a PRF as well, with its key hardwired into programs $\mathsf{P1}, \mathsf{P3}$ and not known to parties (thus, both μ_1 and μ_2 are PRF outputs). Secondly, we augment each program $\mathsf{P1}$, $\mathsf{P2}$, $\mathsf{P3}$, Dec with a "trapdoor step" which makes each of these programs separately deniable, in the spirit of the [SW14] technique. Finally, we make the validity check inside Dec accept if $\mathsf{P2}(r, \mu_1) = \mu_2$, rather than if $\mathsf{PRF}(r, \mu_1) = \mu_2$; the difference is that $\mathsf{P2}$ also accepts "fake" values which are not real preimages of the PRF. We make a similar modification to $\mathsf{P3}$: its validity check verifies that $\mathsf{P1}(s, m) = \mu_1$ and therefore would also accept fake s which is not a real opening of the "commitment". These changes are necessary because without them, an adversary could use the validity check to test whether a given s is a real (PRF) preimage of μ_1 or a fake one.

We present the programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake}$ as described so far, in fig. 3. For readability, the program includes comments to explain what the code is doing. Despite the somewhat dense code, the programs are very structured, and in a nutshell they behave as follows.

- Each program has a main step which is triggered when the program is run on uniformly random s or r (which is the case during an honest execution);
- Programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}$ each have a trapdoor step which is triggered when the programs are given fake randomness (which has a special format recognizable to the programs). The set of fake randomness is sufficiently sparse that the trapdoor steps are almost never triggered on uniform s or r. Fake randomness contains an "instruction" of how the program should behave.
- Programs $\mathsf{P3}$ and Dec run validity checks, as described and motivated above.
- Programs SFake and RFake generate fake randomness which can be recognized by other programs.

In particular, during an honest execution with uniformly random s and r and plaintext m, the parties exchange messages μ_1, μ_2, μ_3 (computed by programs $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}$, respectively), as follows: $\mu_1 = \mathsf{PRF}(s, m)$, $\mu_2 = \mathsf{PRF}(r, \mu_1)$, $\mu_3 = \mathsf{Enc}_K(m, \mu_1, \mu_2)$.[12] The receiver decrypts (μ_1, μ_2, μ_3) by running $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$, which verifies that $\mathsf{PRF}(r, \mu_1) = \mu_2$, then decrypts μ_3 and outputs m.

If the parties want to claim to a coercing adversary that they transmitted a different plaintext $\hat{m}$, they can use SFake, RFake to compute fake s' and r', which are random-looking strings with $\hat{m}$, μ_1, μ_2, and μ_3 encrypted inside. If the adversary decrypts the transcript (μ_1, μ_2, μ_3) with fake $r' = \mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \rho_R)$, it will get $\hat{m}$ as a result (via the trapdoor step of the decryption program). Similarly, the other programs, when given fake s' or r' as input, employ their trapdoor steps as well, making each protocol message appear consistent with $\hat{m}$.

[12] Note that s, m (and r, μ_1) are both *inputs* to the PRF, not keys; we omit PRF keys for simplicity of notation.

[13] We treat s, r as non-random inputs, even though they are supposed to be uniformly chosen, since they are reused across different programs.

Base Protocol programs: first attempt at deniable encryption.

```
Program $\mathsf{P1}(s, m)$
  1. Trapdoor step: if $\mathsf{Dec}_{K_S}(s) = (m', \mu_1', \mu_2', \mu_3')$ and $m' = m$, then return $\mu_1'$ //if $s$ is fake and encodes $m$, output encoded $\mu_1'$
  2. Main step: Return $\mu_1 \leftarrow \mathsf{PRF}(s, m)$. //otherwise output $\mathsf{PRF}(s, m)$

Program $\mathsf{P2}(r, \mu_1)$
  1. Trapdoor step: if $\mathsf{Dec}_{K_R}(r) = (m', \mu_1', \mu_2', \mu_3')$ and $\mu_1' = \mu_1$, then return $\mu_2'$. //if $r$ is fake and encodes $\mu_1$, output encoded $\mu_2'$
  2. Normal step: else return $\mathsf{PRF}(r, \mu_1)$. //otherwise output $\mathsf{PRF}(r, \mu_1)$

Program $\mathsf{P3}(s, m, \mu_1, \mu_2)$
  1. Validity check: if $\mathsf{P1}(s, m) \neq \mu_1$, then abort;
  2. Trapdoor step: if $\mathsf{Dec}_{K_S}(s) = (m', \mu_1', \mu_2', \mu_3')$ and $(m', \mu_1', \mu_2') = (m, \mu_1, \mu_2)$, then return $\mu_3'$. //if $s$ is fake and encodes correct $(m, \mu_1, \mu_2)$, output encoded $\mu_3'$
  3. Normal step: else return $\mathsf{Enc}_K(m, \mu_1, \mu_2)$.//otherwise encrypt $m$

Program $\mathsf{Dec}(r, \mu_1, \mu_2, \mu_3)$
  1. Validity check: if $\mathsf{P2}(r, \mu_1) \neq \mu_2$, then abort;
  2. Trapdoor step: if $\mathsf{Dec}_{K_R}(r) = (m', \mu_1', \mu_2', \mu_3')$ and $(\mu_1', \mu_2', \mu_3') = (\mu_1, \mu_2, \mu_3)$, then return $m'$. //if $r$ is fake and encodes correct $(\mu_1, \mu_2, \mu_3)$, output encoded $m'$
  3. Normal step: else decrypt $(m'', \mu_1'', \mu_2'') \leftarrow \mathsf{Dec}_K(\mu_3)$. If $(\mu_1'', \mu_2'' = \mu_1, \mu_2)$ then output $m''$, else abort. //otherwise decrypt honestly

Program $\mathsf{SFake}(s, m, \hat{m}, \mu_1, \mu_2, \mu_3; \rho_S)$
  1. Validity check: if $\mathsf{P1}(s, m) \neq \mu_1$, then abort;
  2. Normal step: else return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \rho_S)$ // output fake $s$ with fake plaintext and the transcript inside.

Program $\mathsf{RFake}(\hat{m}, \mu_1, \mu_2, \mu_3; \rho_R)$
  1. Normal step: return $\mathsf{Enc}_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \rho_R)$ // output fake $r$ with fake plaintext and the transcript inside
```

Fig. 3: Base Protocol programs: first attempt at deniable encryption. $\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}$ *are deterministic;*[13] SFake*,* RFake *are randomized.*

The problem with the Base Protocol. We designed our scheme above with specific attacks in mind, but is it secure against all attacks? The answer is "almost": it is relatively easy to show security of the scheme in an idealized model where parties (and the adversary) have only oracle access to the programs, *but only as long as the adversary cannot query the* SFake *oracle.* Concretely, the adversary can use SFake to mount a certain attack Ω on the scheme, but this turns out to be the *only* possible type of attack. In section 2.2, we describe a special protection mechanism — comparison-based decryption behavior — which, when added to the protocol, prevents this type of attack and yields a scheme that is fully deniable even if the adversary has an access to *all* oracles including SFake. (And in the full version [CPP18], we prove this result even when the adversary can see *the code* of all programs, obfuscated under IO).

Let's unpack why the protocol described so far is insecure. Recall that we wanted μ_1 to serve as a "commitment", and we wanted $\mathsf{P3}$ to output μ_3 only if the sender uses the same s and m in the commitment and as input to $\mathsf{P3}$. This was important to make sure that for any μ_1, μ_2, at most one consistent μ_3 is efficiently computable. Then, however, we said that $\mathsf{P3}$ should perform its validity check with respect to the whole program $\mathsf{P1}$ and not just the commitment; in particular, the validity check in $\mathsf{P3}$ accepts not only the true opening of the commitment, but also fake s. The problem is that $\mathsf{P1}$, due to its trapdoor step, is not binding: given any ${\mu_1}^* = \mathsf{PRF}(s^*, m_0)$ and $m_1 \neq m_0$, it is easy to generate a different s_1 that passes the verification check. In fact, SFake does exactly that: given $(s^*, m_0, m_1, {\mu_1}^*, \mu_2, \mu_3)$ for some μ_2, μ_3, it outputs s_1 such that $\mathsf{P1}(s_1, m_1) = {\mu_1}^*$.

While this is not yet a concrete attack, it exposes a problem with our initial hope of a committing first message: sender deniability guarantees the first message is easily invertible, potentially with respect to inconsistent plaintexts m, so μ_1 cannot be committing. Thus, it is easy to create many fake s_i consistent with μ_1, and therefore many third messages ${\mu_3}^{(1)}, {\mu_3}^{(2)}, \ldots$, all consistent with a given $({\mu_1}^*, {\mu_2}^*)$. A procedure Ω that does this is detailed in fig. 4. For our purposes, the key features of the attack Ω are as follows.

- To generate such a ${\mu_3}^{(i)}$, encrypting some m_1 for a given $({\mu_1}^*, {\mu_2}^*)$, one has to run $\mathsf{P3}$ on certain fake sender randomness s_i.
- $\mathsf{P3}$ can recognize when it is being used to generate such a ${\mu_3}^{(i)}$. (This is because $\mathsf{P3}$ will be run on a "mixed input": that is, $\mathsf{P3}$ should be run on $s, m, {\mu_1}^*, {\mu_2}^*$, and a fake s_i that encodes the same ${\mu_1}^*$ but different $\tilde{\mu_2} \neq {\mu_2}^*$.)
- The only way to generate such fake s_i efficiently is to run SFake (on a transcript *different* from the one being attacked: specifically, with a different second message).

A procedure Ω to generate a new third message encrypting m_1 and consistent with given first and second messages μ_1, μ_2.

Inputs to $\Omega({\mu_1}^*, {\mu_2}^*, {\mu_3}^*, s^*, m^*, m_1)$ are: transcript $({\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$, sender randomness s^* (which could be real or fake), plaintext m^*, and new desired plaintext m_1:

1. Compute an auxiliary transcript $\widetilde{\mathsf{tr}} = ({\mu_1}^*, \tilde{\mu_2}, \tilde{\mu_3})$ with the same first message ${\mu_1}^*$, but different second message $\tilde{\mu_2}$, by choosing fresh receiver randomness $\tilde{r}$ and setting $\widetilde{\mathsf{tr}} \leftarrow \mathsf{tr}(s^*, \tilde{r}, m^*)$. Note that the first message of this transcript is $\mathsf{P1}(s^*, m^*) = {\mu_1}^*$.
2. Compute $s_1 \leftarrow \mathsf{SFake}(s^*, m^*, m_1, {\mu_1}^*, \tilde{\mu_2}, \tilde{\mu_3})$. Note that s_1 is fake randomness which remembers $m_1, {\mu_1}^*$ and a new $\tilde{\mu_2} \neq {\mu_2}^*$.
3. Compute ${\mu_3}^{(1)} \leftarrow \mathsf{P3}(s_1, m_1, {\mu_1}^*, {\mu_2}^*)$.

Ω can now be repeated on input ${\mu_1}^*, {\mu_2}^*, {\mu_3}^{(1)}, s_1, m_1, m_2$ to generate ${\mu_3}^{(2)}$, and so on.

Fig. 4: Procedure Ω to compute many third messages consistent with given μ_1, μ_2.

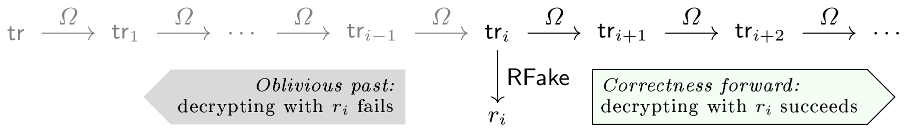
Since it is easy to generate many third messages, our scheme is subject to the same attack as all 2-message schemes: namely, the adversary can generate many ciphertexts ${\mu_3}^{(i)}$, fake each of them to compute an N-times-faked $r^{(N)}$, and then use it to correctly decrypt the original ${\mu_3}^*$. While this attack inheres in all 2-message schemes [BNNO11], in the 3-message case we can fix it. We do so by introducing *levels* and *comparison-based decryption behavior*, which specify how the decryption program should behave when the adversary tries to use such $r^{(N)}$ to decrypt a transcript $({\mu_1}^*, {\mu_2}^*, {\mu_3}^{(i)})$ or a challenge transcript $({\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$.

2.2 Levels, Comparison-Based Decryption, and the Final Scheme

Comparison-based decryption behavior. Let $({\mu_1}^*, {\mu_2}^*, {\mu_3}^*)$ be a challenge transcript. For any superpolynomial T and $j \in \{0, \dots, T\}$, let r_j be the output of RFake on transcript $({\mu_1}^*, {\mu_2}^*, {\mu_3}^{(j)})$, where ${\mu_3}^{(j)}$ is computed by j iterations of Ω. Let ${\mu_3}^{(0)}$ denote the challenge ${\mu_3}^*$. When Dec is run on r_j and ${\mu_3}^{(i)}$, for $i, j \in \{0, \dots, T\}$, *comparison-based decryption behavior* requires the following.

1. *Oblivious past:* When $j > i$, Dec outputs $\perp$.
2. *Correctness forward:* When $j < i$, Dec decrypts ${\mu_3}^{(i)}$ correctly (as long as consistency checks pass).
3. When $j = i$, Dec should decrypt ${\mu_3}^{(i)}$ according to the instruction in fake r_j.

That is, if an adversary creates fake r_j using ${\mu_3}^{(j)}$, the jth in the sequence of ciphertexts, this r_j can be used to honestly decrypt ciphertexts "after" ${\mu_3}^{(j)}$, but cannot be used to decrypt ciphertexts "before" ${\mu_3}^{(j)}$; and naturally, r_j decrypts ${\mu_3}^{(j)}$ itself according to the instruction inside fake r_j.



$$\begin{tikzcd} \mathsf{tr} \arrow[r, "\Omega"] & \mathsf{tr}_1 \arrow[r, "\Omega"] & \cdots \arrow[r, "\Omega"] & \mathsf{tr}_{i-1} \arrow[r, "\Omega"] & \mathsf{tr}_i \arrow[r, "\Omega"] \arrow[d, "\mathsf{RFake}"] & \mathsf{tr}_{i+1} \arrow[r, "\Omega"] & \mathsf{tr}_{i+2} \arrow[r, "\Omega"] & \cdots \\ & & & & r_i & & & \end{tikzcd}$$

Fig. 5: Comparison-based decryption behavior

Comparison-based decryption behavior prevents the attack described above, despite the fact that Ω enables the adversary to generate many third messages. Next, we give some intuition as to why. Recall that the attack had the adversary generate a fake r_j (by faking a ciphertext sequence ${\mu_3}^{(1)}, {\mu_3}^{(2)}, \dots$) and then return to the challenge ${\mu_3}^*$ and decrypt it. Thus, a natural idea to mitigate this attack is to make Dec output $\perp$ whenever fake r_j is used to try to decrypt the initial ${\mu_3}^* = {\mu_3}^{(0)}$.[14] This simple modification indeed stops the attack, but introducing it alone would break security. To maintain security, we need to make sure that Dec on inputs r_j, ${\mu_3}^{(i)}$ should output $\perp$ for *all* $j > i$, and not just $j > i = 0$.[15] In other words, the "oblivious past" rule is the "minimum" modifica-

[14] Such a restriction is not possible in the 2-message case, in contrast to the 3-message case. This relates to the fact that our procedure Ω which generates ${\mu_3}^{(i)}$ is "one way", i.e., it is easy to generate ${\mu_3}^{(i+1)}$ from ${\mu_3}^{(i)}$, but it could be hard — and *is* hard, in our scheme — to generate ${\mu_3}^{(i)}$ from ${\mu_3}^{(i+1)}$. In contrast, in any 2-message scheme, there is no order on the ciphertexts; they are always easy to generate.

[15] To see this, suppose Dec outputs $\perp$ whenever $r_j, j > 0$ is used to decrypt ${\mu_3}^* = {\mu_3}^{(0)}$. Now consider trying to decrypt some ${\mu_3}^{(i)}$ with, say, r_{i+3}. r_3 does not decrypt ${\mu_3}^{(0)}$,

tion which prevents fake r_j from decrypting ${\mu_3}^* = {\mu_3}^{(0)}$ *and* maintains security of the scheme.

Finally, the "correctness forward" rule *must* hold as well, since it is implied by sender-deniability. As a result, the behavior of the decryption program depends on the comparison of "indices" of the transcript and the receiver randomness; therefore, we call this comparison-based decryption behavior.

Implementing comparison-based decryption behavior: levels.

Next, we consider how to construct our programs such that comparison-based decryption behavior holds. When we run Dec on some μ_3 and some r, how does it know whether μ_3 is "forward" of r in the chain (meaning Dec should decrypt honestly), or "in the past" with respect to r (meaning Dec should output $\perp$)?

To this end, we introduce *levels*. That is, we have all fake sender randomness, all fake receiver randomness, and all third message ${\mu_3}^{(i)}$ also encrypt a number ℓ between 0 and some superpolynomial T, as follows.

- Fake sender randomness encrypts, among other things, a level ℓ which is *how many times this randomness was faked*. (E.g., to compute fake randomness, the sender would normally run SFake once, so the level ℓ of the resulting fake randomness is 1. If it runs SFake on the resulting randomness again, its level ℓ will be 2, and so on).
- Each potential third message ${\mu_3}^{(i)}$ also encrypts, in addition to m and μ_1, μ_2, its level, which is *its index i in the chain*. Note that the algorithm Ω which computes ${\mu_3}^{(i)}$ outputs ${\mu_3}^{(1)}$, ${\mu_3}^{(2)}, \ldots$ sequentially, and therefore their index i is well defined. In an honest execution, the level of μ_3 is always set to 0.
- Fake receiver randomness encrypts, among other things, a level ℓ which is *the level of its "parent" transcript* (i.e., the transcript which was used as input to RFake). (E.g., to compute fake randomness, the receiver would normally run RFake on the honest transcript, which has level 0, so the resulting fake randomness would have level 0 too).

We claim that storing this "level" information in fake randomness and third messages is enough for the scheme to maintain the level information accurately and follow comparison-based decryption behavior. For instance, Dec can decide its output behavior by comparing the levels inside r and μ_3. RFake can record the correct level of r by copying the level of its parent ciphertext. SFake can maintain the correct number of times something was faked, by reading the level in its input s and incrementing it. P3, as discussed above, can detect when it is being run within Ω, and it can put inside its output third message the level it copied from input s; since generating each new μ_3 requires once-more fake s, the level in s — i.e., the number of times it was faked — corresponds to the index of μ_3 in the chain.

and the difference between $({\mu_3}^{(0)}, r_3)$ and $({\mu_3}^{(i)}, r_{i+3})$ is that ${\mu_3}^{(0)}$ was generated with truly random s whereas ${\mu_3}^{(i)}$ used s_i which was faked i times. Sender deniability requires these two cases be indistinguishable, so ${\mu_3}^{(i)}$ must not be decrypted by r_{i+3}.

Our final protocol in the oracle-access model. We present our final protocol (albeit still in the oracle model) in figs. 6–7. This scheme is a provably secure deniable encryption scheme in the oracle access model. A proof outline follows the program descriptions, and a complete proof is given in the full version [CPP18].

The structure of the final protocol programs is summarized below.

– Each program has a main step which is triggered when the program is run on uniformly random s or r, which is the case during an honest execution.
– Programs $\mathsf{P1}$, $\mathsf{P2}$, $\mathsf{P3}$, and Dec also have a trapdoor step which is triggered when the programs receive fake randomness (which has a special format recognizable to the programs). The set of fake randomness is sufficiently sparse that the trapdoor step is almost never triggered on uniformly chosen s or r. Fake randomness contains an "instruction" of how the program should behave on some particular input.
– Programs $\mathsf{P3}$ and Dec also have a "mixed input" step which serves to prevent attacks using Ω to generate many third messages μ_3. $\mathsf{P3}$'s mixed input step copies the level from its input s into the third message μ_3, ensuring μ_3 encrypts its own index in the sequence. Dec's mixed input step implements comparison-based decryption behavior by comparing the levels of μ_3 and r. The mixed input steps are triggered when the programs receive fake s (or r) as input, but the program's other inputs do not match the inputs in the instruction inside s (or r). $\mathsf{P3}$ enters its mixed input step when its input and fake s contain the same μ_1 but different second messages, and Dec enters its mixed input step when its input and fake r contain the same μ_1, μ_2 but different third messages.
– Programs $\mathsf{P3}$ and Dec's output behavior depends on validity checks, as in the Base Protocol (and for the same reasons as in the Base Protocol).
– Programs SFake and RFake generate fake randomness that is recognizable to the other programs, and maintain accurate level information inside the fake randomness as follows: SFake increments the level of sender randomness with respect to its input sender randomness (unless the latter is honest, in which case SFake sets the level to 0); and RFake copies the level from the parent transcript into fake randomness.

The interesting cases of protocol execution are summarized next.

– **Normal protocol execution.** Executing the programs on randomly chosen s^*, r^* and plaintext m_0^* triggers the main step, yielding outputs ${\mu_1}^* = \mathsf{PRF}(s^*, m_0^*)$, ${\mu_2}^* = \mathsf{PRF}(r^*, {\mu_1}^*)$, and ${\mu_3}^* = \mathsf{Enc}_K(m_0^*, {\mu_1}^*, {\mu_2}^*, 0)$, where the last 0 is the level. Dec, given the resulting transcript as input, outputs m_0^* via its main step.
– **Fake randomness of parties.** A sender wishing to claim that it sent plaintext $m_1^* \neq m_0^*$ can run SFake to obtain fake s' encoding $(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 1)$, where the last 1 is the level. A receiver wishing to claim that it received $m_1^* \neq m_0^*$ can run RFake to obtain fake r' encoding $(m_1^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, 0)$, where the last 0 is the level. Executing programs on fake s' or fake r' and m_1^*

Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

```
Program P1$(s, m)$
1. Trapdoor step:
   (a) out $\leftarrow \mathsf{Dec}_{K_S}(s)$; if out = 'fail' goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m = m'$ then return $\mu_1'$; //if $s$ is fake and encodes $m$, output encoded $\mu_1'$
2. Main step:
   (a) Return $\mu_1 \leftarrow \mathsf{PRF}_{k_S}(s, m)$. //otherwise output $\mathsf{PRF}(s, m)$
Program P3$(s, m, \mu_1, \mu_2)$
1. Validity check: if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. Trapdoor step:
   (a) out $\leftarrow \mathsf{Dec}_{K_S}(s)$; if out = 'fail' goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1, \mu_2 = m', \mu_1', \mu_2'$ then return $\mu_3'$; //if $s$ is fake and encodes correct $(m, \mu_1, \mu_2)$, output encoded $\mu_3'$
3. Mixed input step: If $m, \mu_1 = m', \mu_1'$ but $\mu_2 \neq \mu_2'$ then return $\mu_3 \leftarrow \mathsf{Enc}_K(m, \mu_1, \mu_2, \ell')$; //if $s$ is fake and encodes correct $(m, \mu_1)$ but incorrect $\mu_2'$, encrypt $m$ with level copied from $s$
4. Main step:
   (a) Return $\mu_3 \leftarrow \mathsf{Enc}_K(m, \mu_1, \mu_2, 0)$. //otherwise encrypt $m$ with level 0
Program SFake$(s, m, \hat{m}, \mu_1, \mu_2, \mu_3)$
1. Validity check: if $\mathsf{P1}(s, m) \neq \mu_1$ then abort;
2. Trapdoor step:
   (a) out $\leftarrow \mathsf{Dec}_{K_S}(s)$; if out = 'fail' goto main step, else parse out as $(m', \mu_1', \mu_2', \mu_3', \ell')$;
   (b) If $m, \mu_1 = m', \mu_1'$ then
       i. If $\ell \geq T$ then abort;
       ii. Return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell + 1)$. //if input $s$ is already fake then output new fake $s$ with fake plaintext, the transcript, and incremented level
3. Main step:
   (a) Return $\mathsf{Enc}_{K_S}(\hat{m}, \mu_1, \mu_2, \mu_3, 1)$. //otherwise output fake $s$ with fake plaintext, the transcript, and level 1
```

Fig. 6: Programs $\mathsf{P1}, \mathsf{P3}, \mathsf{SFake}$.

triggers the trapdoor step, so programs will output the values hardwired into the fake s' or r'. Thus, $\mathsf{P1}$ will output $\mu_1{}^*$, $\mathsf{P2}$ will output $\mu_2{}^*$, $\mathsf{P3}$ will output $\mu_3{}^*$, and Dec will output m_1^* via their trapdoor steps, making the transcript, originally for plaintext m_0^*, appear consistent with m_1^*.

– **Efficiently computable related transcripts.** It is only possible to compute related transcripts of the form $(\mu_1{}^*, \mu_2{}^*, \mu_3)$, where $\mu_3 = \mathsf{Enc}_K(m, \mu_1{}^*, \mu_2{}^*, \ell)$, $\ell \geq 1$; moreover, the only way of doing so is to use the procedure Ω described above (which invokes SFake). Trying to compute μ_3 a for such transcript will cause program $\mathsf{P3}$ to execute its "mixed input step", ensuring that such μ_3 receives level $\ell \geq 1$; for this, it is important that SFake increments the level inside s. Trying to decrypt such a related transcript $(\mu_1{}^*, \mu_2{}^*, \mu_3)$ will cause program Dec to execute its "mixed input step",

```
Programs P2, Dec, RFake.
Program P2$(r, \mu_1)$
 1. Trapdoor step:
    (a) out $\leftarrow$ Dec$_{K_R}(r)$; if out = 'fail' then goto main step, else parse out as
        $(m', \mu_1', \mu_2', \mu_3', L', \hat{\rho})$;
    (b) If $\mu_1 = \mu_1'$ then return $\mu_2'$; //if $r$ is fake and encodes $\mu_1$, output encoded $\mu_2'$
 2. Main step:
    (a) Return $\mu_2 \leftarrow$ PRF$_{k_R}(r, \mu_1)$. //otherwise output PRF$(r, \mu_1)$
Program Dec$(r, \mu_1, \mu_2, \mu_3)$
 1. Validity check: if P2$(r, \mu_1) \neq \mu_2$ then abort;
 2. Trapdoor step:
    (a) out $\leftarrow$ Dec$_{K_R}(r)$; if out' = 'fail' then goto main step; else parse out' as
        $(m', \mu_1', \mu_2', \mu_3', \ell', \hat{\rho})$;
    (b) if $\mu_1, \mu_2, \mu_3 = \mu_1', \mu_2', \mu_3'$ then return $m'$; //if $r$ is fake and encodes correct
        $(\mu_1, \mu_2, \mu_3)$, output encoded $m'$
    (c) out $\leftarrow$ Dec$_K(\mu_3)$; if out'' = 'fail' then abort, else parse out'' as $(m'', \mu_1'', \mu_2'', \ell'')$;
 3. Mixed input step: If $\mu_1, \mu_2 = \mu_1', \mu_2'$ but $\mu_3 \neq \mu_3'$ then
    (a) If $(\mu_1', \mu_2') = (\mu_1'', \mu_2'')$ and $\ell' < \ell''$ then return $m''$; //if $r$ is fake and en-
        codes correct $(\mu_1, \mu_2)$ but incorrect $\mu_3'$, decrypt honestly or abort, depending
        on whether the level in $r$ is smaller than in $\mu_3$ or not
    (b) Else abort.
 4. Main step:
    (a) out $\leftarrow$ Dec$_K(\mu_3)$; if out = 'fail' then abort, else parse out as $(m'', \mu_1'', \mu_2'', \ell'')$;
    (b) If $(\mu_1, \mu_2) = (\mu_1'', \mu_2'')$ then return $m''$; //otherwise decrypt honestly
    (c) Else abort.
Program RFake$(\hat{m}, \mu_1, \mu_2, \mu_3; \rho)$
 1. out $\leftarrow$ Dec$_K(\mu_3)$; if out = 'fail' then abort, else parse out as $(m'', \mu_1'', \mu_2'', \ell'')$;
 2. Return $r' \leftarrow$ Enc$_{K_R}(\hat{m}, \mu_1, \mu_2, \mu_3, \ell''$, prg$(\rho))$. // output fake $r$ with fake plaintext,
    the transcript, and the level copied from $\mu_3$
```

Fig. 7: Programs P2, Dec, RFake.

ensuring that the requisite decryption behavior is observed (that fake r decrypts correctly transcripts with larger level, but fails to decrypt transcripts with smaller level); for this, it is important that RFake copies the level from the transcript to r.

Outline of security proof in oracle-access model. Since the proof even in this simpler (oracle-access) model is somewhat lengthy, we only outline the main steps here, with intuition for each. The proof proceeds in four main hybrid steps. We start with a real execution corresponding to plaintext m_0^*, where the adversary receives real randomness s^*, r^*.

– **Step I: indistinguishability of sender explanations.** Instead of giving the adversary real s^*, we give it $s' = \mathsf{Enc}_{K_S}(m_0^*, \mu_1{}^*, \mu_2{}^*, \mu_3{}^*, \ell = 0)$ (note that this s' contains level 0, unlike fake randomness produced by SFake which contains level at least 1).

Intuitively, the reason why we can switch from s^* to s' indistinguishably is because all programs treat s^* and s' indistinguishably. That is:

- either the programs output the same value, possibly via different branches of execution (e.g., $\mathsf{P1}$ on input (s^*, m_0^*) outputs ${\mu_1}^*$ via its main step and on input (s', m_0^*) outputs ${\mu_1}^*$ via its trapdoor step);
- or the programs execute the same code, possibly outputting different results (e.g., $\mathsf{P1}$, on input (s^*, m_1^*) or (s', m_1^*), evaluates a PRF on its input and outputs the result).

The above, and the fact that s' is pseudorandom, allow us to change s^* to s' (similarly to the [SW14] proof for sender-deniable encryption).

– **Step II: indistinguishability of receiver explanations.** Instead of giving the adversary real r^*, we give it fake r', i.e., $r' = \mathsf{Enc}_{K_R}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell = 0, \rho_R)$. Unlike in Step I, here there is a transcript with respect to which the decryption program treats r^* and r' distinguishably.
Recall that r^* honestly decrypts *all* related transcripts, while r' only honestly decrypts "forward", i.e., for related transcripts with level $\ell \geq 1$. Thus, the two programs may treat level-0 transcripts differently. Consider a transcript $({\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$, where $\overline{{\mu_3}^*} = \mathsf{Enc}_K(m_1^*, {\mu_1}^*, {\mu_2}^*, \ell = 0)$ is like ${\mu_3}^*$ except that it encrypts the wrong plaintext m_1^*. This transcript decrypts correctly to m_1^* with r^*, but decrypting it with r' returns $\perp$ due to the level comparison logic. This single transcript makes r^* and r' distinguishable. As a result, the proof of Step I does not work here. Therefore, we first move to a hybrid where this "differing" transcript doesn't exist, as follows. First, since s^* (the preimage of PRF output ${\mu_1}^*$) is not part of the distribution anymore, we can move ${\mu_1}^*$ outside the PRF image. Then we argue that $\mathsf{P3}$ never outputs $\overline{{\mu_3}^*}$:

- *The main step cannot output* $\overline{{\mu_3}^*}$, since it is executed only if the validity check passes *via a correct PRF preimage*, which now does not exist.
- *The mixed step cannot output* $\overline{{\mu_3}^*}$, since $\mathsf{P3}$ can only output a ciphertext with level 0 (like $\overline{{\mu_3}^*}$) via the mixed step if its input randomness has level 0, and such input randomness is hard to find since SFake never outputs randomness with level 0.
- *The trapdoor step cannot output* $\overline{{\mu_3}^*}$, since $\mathsf{P3}$ can only output $\overline{{\mu_3}^*}$ via the trapdoor step if it receives as input fake randomness that has $\overline{{\mu_3}^*}$ inside to begin with. Since there are no other means of computing $\overline{{\mu_3}^*}$, such randomness is also hard to find.

Once the differing transcript $({\mu_1}^*, {\mu_2}^*, \overline{{\mu_3}^*})$ is eliminated, we can switch r^* to r' similarly to Step I.

– **Step III: indistinguishability of plaintexts.** The next step is to switch ${\mu_3}^*$ from encrypting m_0^* to encrypting m_1^*. This is done by "detaching" ${\mu_3}^*$ from its key K in programs $\mathsf{P3}$ and Dec. Concretely:

- $\mathsf{P3}$ can only output ${\mu_3}^*$ via the trapdoor thread (which does not use the key K). The reason is very similar to the case-by-case analysis of $\mathsf{P3}$ in Step II: the main step requires a PRF preimage, which does not exist, and the mixed step requires level-0 sender randomness, which is hard to find.

- Dec can only "decrypt" ${\mu_3}^*$ via the trapdoor thread (which, again, does not use K). To guarantee this, we first move ${\mu_2}^*$ outside of the PRF image (this is possible since r^* is no longer part of the distribution). Then ${\mu_3}^*$ is never decrypted via the main step because the preimage for ${\mu_2}^*$ does not exist. Further, ${\mu_3}^*$ cannot be decrypted via the mixed step either, because the "correctness forward" decryption rule outputs $\perp$ unless the input receiver randomness has level smaller than the level in ${\mu_3}^*$, and this is not possible since ${\mu_3}^*$ has the smallest possible level, 0.

In other words, neither $\mathsf{P3}$ nor Dec need to use K to encrypt or decrypt ${\mu_3}^*$. Therefore we can "detach" K and ${\mu_3}^*$ and change the plaintext to m_1^*.
Note that the transcript now contains m_1^*, and both sender and receiver randomness s', r' are consistent with m_0^*. However, the proof is not finished yet, since parties cannot produce such s' themselves (since s' has level 0).

– **Step IV: indistinguishability of levels.** The last step is to change the level inside s' from 0 to 1, i.e., let $s' = \mathsf{Enc}_{K_S}(m_0^*, {\mu_1}^*, {\mu_2}^*, {\mu_3}^*, \ell = 1)$. To understand the challenge of this step, it is instructive to take a "level-centric" perspective: let's put aside that the scheme is about transmitting plaintexts, and instead think of fake s as an encryption of level (0 or 1), think of ${\mu_3}^*$ as an encryption of level 0, and think of the programs of deniable encryption as implementing homomorphic operations on encrypted levels. For example, program SFake outputs fake randomness which is an encryption of incremented level, and thus implements a homomorphic $\mathsf{Increment}$ operation on levels. Program Dec compares levels inside μ_3 and r and, based on that, decrypts or outputs $\perp$, and thus it implements a homomorphic isLess function on levels, which reveals (in the clear) if one level is smaller than the other.
From this perspective, step IV essentially requires switching s' from an encryption of 0 to an encryption of 1, while the adversary has access to homomorphic functions $\mathsf{Increment}$ and isLess.[16] In the oracle-access model, it can be easily shown that polynomially bounded adversaries cannot distinguish between $\mathsf{Enc}(0)$ and $\mathsf{Enc}(1)$, even given oracle access to isLess and $\mathsf{Increment}$, as long as the largest allowed level T is superpolynomial: this is because the adversary can only generate polynomial-length sequences of encryptions — $\mathsf{Enc}(1), \mathsf{Enc}(2), \ldots$ or $\mathsf{Enc}(2), \mathsf{Enc}(3), \ldots$ (depending on whether the challenge ciphertext was $\mathsf{Enc}(0)$ or $\mathsf{Enc}(1)$) — and the oracles' behavior will be identical on both sequences.

This concludes the proof outline in the oracle-access model. We underline that in the actual construction we need special types of PRFs, encryption schemes, and a special *level system* primitive in order to prove security with iO. The proofs of steps I-III in the final construction roughly follow the same outline (sometimes with several hybrids per each logical step), but the proof of the step IV (indistinguishability of levels) requires substantial additional work when the adversary possesses the code of the programs.

[16] Recall that the adversary also has ${\mu_3}^*$ which is an encryption of level 0. For simplicity, we ignore this fact in this high-level overview.

3 Defining Bideniable and Off-the-Record Deniable Encryption

We present the definition of interactive deniable encryption, or, more formally, interactive bideniable message transmission, in the CRS model.

Syntax. An interactive deniable encryption scheme π consists of seven algorithms $\pi = (\mathsf{Setup}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake})$, where Setup is used to generate the public programs (i.e., the CRS), programs $\mathsf{P1}$, $\mathsf{P3}$ and SFake are used by the sender, and programs $\mathsf{P2}$, Dec and RFake are used by the receiver. Let $\mathsf{tr} = \pi(s, r, m)$ denote the transcript of a protocol execution on input plaintext m, sender randomness s, and receiver randomness r, i.e., the sequence of three messages sent in the protocol execution. That is, $\pi(s, r, m) = \mathsf{tr} = (\mu_1, \mu_2, \mu_3)$, where $\mu_1 = \mathsf{P1}(s, m)$, $\mu_2 = \mathsf{P2}(r, \mu_1)$, and $\mu_3 = \mathsf{P3}(s, m, \mu_1, \mu_2)$.

The faking algorithms have the following syntax: $\mathsf{SFake}(s, m, m', \mathsf{tr}; \rho)$ expects to take as input a transcript tr along with the true random coins s and true plaintext m which were used to compute tr, and a desired fake plaintext m'. SFake is randomized and ρ denotes its randomness. RFake has the same syntax except that it expects receiver randomness r instead of sender randomness s.

Bideniable and off-the-record-deniable encryption in the CRS model. Next, we define standard and off-the-record deniability for interactive deniable encryption in the CRS model. For simplicity, we focus on bit encryption. The definitions are naturally extensible to multi-bit plaintexts.

Formally, the deniable encryption algorithms should take the CRS as input. We omit this for notational simplicity as it is unnecessary in our construction (where the CRS contains the programs, and the programs do not take the CRS as input).

Definition 1 Bideniable bit encryption in the CRS model. $\pi = (\mathsf{Setup}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{Dec}, \mathsf{SFake}, \mathsf{RFake})$ *is a 3-message bideniable interactive encryption scheme for message space* $\mathcal{M} = \{0, 1\}$*, if it satisfies the following correctness and bideniability properties.*

- **Correctness:** *There exists a negligible function* $\nu(\lambda)$ *such that for at least a* $(1 - \nu)$ *fraction of randomness* $r_{\mathsf{Setup}} \in \{0, 1\}^{|r_{\mathsf{Setup}}|}$*, for any* $m \in \mathcal{M}$*,*

$$\mathsf{Pr}\left[m' \neq m \quad : \quad \begin{array}{l} \mathsf{CRS} \leftarrow \mathsf{Setup}(r_{\mathsf{Setup}}) \\ s \leftarrow \{0,1\}^{|s|} \\ r \leftarrow \{0,1\}^{|r|} \\ \mathsf{tr} \leftarrow \pi(s, r, m) \\ m' \leftarrow \mathsf{Dec}(r, \mathsf{tr}) \end{array} \right] \leq \nu(\lambda) \ .$$

- **Bideniability:** *No PPT adversary* Adv *has more than negligible advantage in the following game, for any* $m_0, m_1 \in \mathcal{M}$*:*
  1. *The challenger chooses random* r_{Setup} *and generates* $\mathsf{CRS} \leftarrow \mathsf{Setup}(r_{\mathsf{Setup}})$*. It also chooses a bit* b *at random.*

2. If $b = 0$, then the challenger behaves as follows:
   (a) It chooses random s^*, r^* and computes $\mathsf{tr}^* = \pi(s^*, r^*, m_0)$.
   (b) It gives the adversary $(\mathsf{CRS}, m_0, m_1, s^*, r^*, \mathsf{tr}^*)$.

3. If $b = 1$, then the challenger behaves as follows:
   (a) It chooses random s^*, r^* and computes $\mathsf{tr}^* \leftarrow \pi(s^*, r^*, m_1)$.
   (b) It sets $s' \leftarrow \mathsf{SFake}(s^*, m_1, m_0, \mathsf{tr}^*; \rho_S)$ for random ρ_S.
   (c) It sets $r' \leftarrow \mathsf{RFake}(r^*, m_1, m_0, \mathsf{tr}^*; \rho_R)$ for random ρ_R.
   (d) It gives the adversary $(\mathsf{CRS}, m_0, m_1, s', r', \mathsf{tr}^*)$.

4. Adv outputs b' and wins if $b = b'$.

Next, we define off-the-record deniability. We define it for an arbitrary message space instead of bit encryption, since having $|\mathcal{M}| > 2$ allows for an extra case when plaintexts claimed by the sender, by the receiver, and the real plaintext are three different strings (case $b = 2$ in the definition below).

Definition 2 Off-the-record deniable encryption in the CRS model. *We say that a scheme is off-the-record deniable, if it satisfies correctness as above and also has the following property.*

Off-the-record deniability: No PPT adversary Adv wins with more than negligible advantage in the following game, for any $m_0, m_1, m_2 \in \mathcal{M}$:

1. The challenger chooses random r_{Setup} and generates $\mathsf{CRS} \leftarrow \mathsf{Setup}(r_{\mathsf{Setup}})$. It also chooses random $b \in \{0, 1, 2\}$.
2. If $b = 0$, then the challenger generates the following variables:
   (a) The challenger chooses random s^*, r^* and computes $\mathsf{tr}^* \leftarrow \pi(s^*, r^*, m_0)$;
   (b) It sets $r' \leftarrow \mathsf{RFake}(r^*, m_0, m_1, \mathsf{tr}^*; \rho_R)$ for randomly chosen ρ_R.
   (c) It gives the adversary $(\mathsf{CRS}, m_0, m_1, m_2, s^*, r', \mathsf{tr}^*)$.
3. If $b = 1$, then the challenger generates the following variables:
   (a) The challenger chooses random s^*, r^* and computes $\mathsf{tr}^* \leftarrow \pi(s^*, r^*, m_1)$;
   (b) It sets $s' \leftarrow \mathsf{SFake}(s^*, m_1, m_0, \mathsf{tr}^*; \rho_S)$ for randomly chosen ρ_S.
   (c) It gives the adversary $(\mathsf{CRS}, m_0, m_1, m_2, s', r^*, \mathsf{tr}^*)$.
4. If $b = 2$, then the challenger generates the following variables:
   (a) The challenger chooses random s^*, r^* and computes $\mathsf{tr}^* \leftarrow \pi(s^*, r^*, m_2)$;
   (b) It sets $s' \leftarrow \mathsf{SFake}(s^*, m_2, m_0, \mathsf{tr}^*; \rho_S)$ for randomly chosen ρ_S.
   (c) It sets $r' \leftarrow \mathsf{RFake}(r^*, m_2, m_1, \mathsf{tr}^*; \rho_R)$ for randomly chosen ρ_R.
   (d) It gives the adversary $(\mathsf{CRS}, m_0, m_1, m_2, s', r', \mathsf{tr}^*)$.
5. Adv outputs b' and wins if $b = b'$.

We say that an encryption scheme is bideniable (resp., off-the-record deniable) with (t, ε)-security, if the distinguishing advantage of any any size-t adversary in the bideniability (resp., off-the-record deniability) game is at most ε.

Single-execution security implies multi-execution security. In definitions 1 and 2, the CRS is global (i.e., non-programmable). These definitions do not involve simulation and the same set of programs is used throughout. Furthermore, even though definitions 1 and 2 consider a single protocol execution, a simple hybrid argument shows that security of a single execution implies security of arbitrarily polynomially many executions with the same set of programs.[17]

Definition 3 Public receiver deniability. *A deniable scheme has* public receiver-deniability *if* RFake *takes as input only the transcript* tr *and fake plaintext* m' *(not true random coins of the receiver* r^* *and true plaintext* m*).*

This concludes the informal scheme description, proof intuition, and full definitions. We have overviewed the key ideas underlying the full construction. Please see the full version [CPP18] for complete details and proofs.

Acknowledgements. RC is a member of the Check Point Institute for Information Security. Supported by the NSF MACS project. SP's research is supported by the MIT Media Lab's Digital Currency Initiative and its funders, and earlier, was supported by the following grants: NSF MACS (CNS-1413920), DARPA IBM (W911NF-15-C-0236), Simons Investigator award agreement dated June 5th, 2012, and the Center for Science of Information (CSoI), an NSF Science & Technology Center, under grant agreement CCF-0939370.

References

AFL16. Daniel Apon, Xiong Fan, and Feng-Hao Liu. Deniable attribute based encryption for branching programs from LWE. In *TCC 2016-B, Proceedings Part II*, pages 299–329, 2016. 10

BGB04. Nikita Borisov, Ian Goldberg, and Eric A. Brewer. Off-the-record communication, or, why not to use PGP. In Vijay Atluri, Paul F. Syverson, and Sabrina De Capitani di Vimercati, editors, *Workshop on Privacy in the Electronic Society (WPES 2004), Proceedings*, pages 77–84. ACM, 2004. 3

BNNO11. Rikke Bendlin, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Lower and upper bounds for deniable public-key encryption. In *ASIACRYPT 2011, Proceedings*, pages 125–142, 2011. 2, 6, 7, 10, 11, 12, 13, 14, 17

BPR15. Nir Bitansky, Omer Paneth, and Alon Rosen. On the cryptographic hardness of finding a nash equilibrium. *Electronic Colloquium on Computational Complexity (ECCC)*, 22:1, 2015. 4, 9

BPW16. Nir Bitansky, Omer Paneth, and Daniel Wichs. Perfect structure on the edge of chaos - trapdoor permutations from indistinguishability obfuscation. In *TCC 2016-A, Proceedings, Part I*, pages 474–502, 2016. 4

CDMW09. Seung Geol Choi, Dana Dachman-Soled, Tal Malkin, and Hoeteck Wee. Improved non-committing encryption with applications to adaptively secure protocols. In *ASIACRYPT 2009, Proceedings*, pages 287–302, 2009. 10

CDNO96. Ran Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. *IACR Cryptology ePrint Archive*, 1996:2, 1996. 1, 2, 3, 10, 11

[17] We can change all executions from real to fake one by one, where the reduction from a single-execution security will generate other executions on its own, since knowing the CRS (but not its generation randomness) suffices to run all programs.

CFGN96. Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *STOC 1996, Proceedings*, pages 639–648, 1996. 10

CHJV14. Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Indistinguishability obfuscation of iterated circuits and RAM programs. *IACR Cryptology ePrint Archive*, 2014:769, 2014. 4, 9

CIO16. Angelo De Caro, Vincenzo Iovino, and Adam O'Neill. Deniable functional encryption. In *PKC 2016, Proceedings, Part I*, pages 196–222, 2016. 10

CPP18. Ran Canetti, Sunoo Park, and Oxana Poburinnaya. Fully bideniable interactive encryption. *IACR Cryptol. ePrint Arch.*, 2018:1244, 2018. 3, 9, 11, 12, 16, 20, 27

Dac12. Dana Dachman-Soled. On the impossibility of sender-deniable public key encryption. *IACR Cryptology ePrint Archive*, 2012:727, 2012. 10, 11

DH76. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976. 1

DKSW09. Yevgeniy Dodis, Jonathan Katz, Adam D. Smith, and Shabsi Walfish. Composability and on-line deniability of authentication. In *TCC 2009, Proceedings*, pages 146–162, 2009. 10

GKW17. Shafi Goldwasser, Saleet Klein, and Daniel Wichs. The edited truth. In *TCC 2017, Proceedings, Part I*, pages 305–340, 2017. 10

GM84. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984. 1

IKOS10. Yuval Ishai, Abishek Kumarasubramanian, Claudio Orlandi, and Amit Sahai. On invertible sampling and adaptive security. In *ASIACRYPT 2010, Proceedings*, pages 466–482, 2010.

KLW15. Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *STOC 2015, Proceedings*, pages 419–428, 2015. 4

Nie02. Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *CRYPTO 2002, Proceedings*, pages 111–126, 2002. 11

OPW11. Adam O'Neill, Chris Peikert, and Brent Waters. Bi-deniable public-key encryption. In *CRYPTO 2011, Proceedings*, pages 525–542, 2011. 3, 10

RSA78. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. 1

SW14. Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC 2014, Proceedings*, pages 475–484, 2014. 2, 6, 7, 9, 11, 12, 15, 23