Secure Onion Encryption and the Case of Counter Galois Onion *

Jean Paul Degabriele[1], Alessandro Melloni[2], and Martijn Stam[2]

[1] Technology Innovation Institute, Masdar City, Abu Dhabi, U.A.E.
jeanpaul.degabriele@tii.ae
[2] Simula UiB, Bergen, Norway.
alessandro.melloni.29@gmail.com,martijn@simula.no

Abstract. The recently introduced Counter Galois Onion (CGO) is a new symmetric onion encryption scheme designed to replace the current one used by Tor, with integration in Tor's Rust implementation Arti ongoing. Intuitively, CGO uses an updatable tweakable split-domain cipher as its building block, which provides it with the necessary non-malleability properties while attaining better performance than the alternative approach of realising it from a wide blockcipher (with full SPRP security). However, onion encryption as used in Tor with various functionality features and security trade-offs, is not that well-studied by the cryptographic community. As a result, the requirements of this important primitive which protects the privacy of millions of users on a daily basis, is not well understood and whether CGO fulfills all its security goals unclear.
In this work, we initiate the study of real-world symmetric onion encryption by presenting a new security model capturing Tor's leaky pipes functionality, associated data, and partial forward security, neither of which were covered previously. We then use this new security model to solidify the security claims of CGO in the forward direction by proving that if the underlying primitive is a suitably secure tweakable split-domain cipher, then CGO is a secure onion encryption scheme.

Keywords: Tor · Onion Encryption · Tagging Attacks · Forward Security · RPRP

1 Introduction

Onion encryption is crucial for many privacy solutions and it appears in many guises. Its most practical incarnation is in Tor, which aims to provide online anonymity to the masses. Its onion encryption is akin to that of a secure channel: a Tor proxy first settles on a route, then runs a sequence of public-key protocols to exchange keys with each of the nodes on the route, thus establishing a circuit that can be used to send and receive the data using fast symmetric cryptography. This approach contrasts with other applications, where onion encryption functions analogously to public-key encryption and each ciphertext is self-contained and routed individually [5]. Although the rationale behind both kinds of onion routing is similar, they require fundamentally different treatments to formalize (cf. secure channels vs. secure email). Furthermore, the symmetric encryption and decryption along a circuit by its very nature involves multiple parties (the proxy and typically 3 to 5 routers), setting Tor's record layer apart from traditional secure channels [**?**,15]. Finally, Tor's overall design deliberately provides a trade-off between security and efficiency: most obviously, the choice for low-latency makes Tor more susceptible to traffic analysis, but surprisingly efficiency considerations also directly impact security considerations of the symmetric onion encryption layer, where Tor's designers have showed a willingness to accept less-than-perfect security in cases where the efficiency, or functionality, gains outweigh the reduced security. Consequently, the complicated circuit functionality of Tor's onion encryption and its associated security, are not as well understood as, for instance, AEAD or stateful encryption. Some recent works [9,17] formalized elements of Tor's onion encryption and routing, but only did so partially, leaving a comprehensive formal treatment open.

As we mentioned, the Tor designers are acutely aware of security–efficiency tradeoffs. Unsurprisingly, these risk assessments change over time, as both our understanding of the potency of attacks and the general advantages and disadvantages of certain design choices become clearer. In the case of Tor, its symmetric component has remained unchanged since Tor was first introduced in 2004 [10] and it can be considered somewhat outdated to modern standards. As explained by Mathewson in Tor's Proposal 202 [13], the current scheme has a number of weaknesses, namely its malleability and susceptibility to tagging attacks, the rather short authentication tag, the use of a vulnerable MAC construction to compute authentication tags, and the adherence to a MAC-then-encrypt approach for the onion encryption—which

* ©Under submission, an extended abstract of this paper will appear; this is the full version.

in the case of authenticated encryption is known to be generically insecure. Therefore, Tor is currently upgrading its symmetric onion encryption, responsible for protecting the data traffic over the Tor network. Specifically, it will be replaced with a recently introduced onion encryption scheme called Counter Galois Onion, or CGO [8], which is currently being integrated in Arti (a Rust implementation of Tor). CGO aims to address all existing shortcomings and additionally provide forward security, while also offering good performance. Although CGO contains a clear design rationale, its security claims would benefit from a more detailed, formal analysis.

Our Contribution. In this work we provide a security treatment of CGO that places its security claims on firm ground. Our contribution is twofold.

New Security Model OAEF_ℓ. Firstly, in Section 4 we propose an all-in-one security model OAEF_ℓ (where ℓ parameterizes the circuit length) for the forward direction of onion encryption. Our model respects the full functionality expected by Tor's onion encryption and captures all the required security properties. Our model is the first to include leaky pipes, which introduces new syntactical and functional requirements on the onion encryption scheme. Tor's *leaky pipes* feature enables the proxy to send messages to any router within the circuit (and vice versa) and not just the last (exit) router in the circuit. Accordingly, any suitable scheme must allow a router to determine unambiguously whether it is the intended recipient of a ciphertext, yet without leaking the intended recipient to any earlier routers on the circuit. Specifically, the cryptographic processing at an onion router in the forward direction must now be able to determine, solely from the information contained in an incoming ciphertext, whether it is the intended recipient of that ciphertext or whether it should forward the processed ciphertext to the next router in the circuit. (Similarly, in the backward direction, the onion proxy needs to determine from which onion router an incoming ciphertext originated.) Leaky pipes significantly complicate security modelling, as they inevitably also allow accidental spillage, where a ciphertext is overeagerly accepted by an earlier router than the intended destination, implying a non-zero correctness error (plus, obviously, an additional integrity-related attack vector for an adversary).

Another complication is the desire to provide forward security to the end-points of the secure channel, without imposing forward security on any of the intermediate routers. Thus, a suitable security definition for Tor should avoid being too strong. Our OAEF_ℓ security notion for onion encryption carefully encompasses the security properties outlined above, namely, confidentiality, end-to-end integrity, non-malleability, resistance to tagging attacks, and end-to-end forward security.

Security of CGO. Secondly, in Section 5 we prove the OAEF_ℓ security of CGO's, in other words, we show that CGO is a secure onion encryption in the forward direction, provided that the underlying updatable split-domain tweakable cipher is secure. The proof is a concrete game-hopping proof where we use Degabriele et al.'s more involved multi-instance CRND_ℓ security notion for the underlying primitive, as it allows us to swap out all calls to the updatable split-domain tweakable cipher with freshly generated randomness in an single hop. However, in order to do so, we first need a number of hops to reduce superfluous calls and flag problematic calls, as those would invalidate a reduction to the CRND_ℓ security of the underlying primitive. Subsequently, we face another minor problem, due to a mismatch between when randomness is used: in the 'ideal' OAEF_ℓ game, it should be just in time (when an oracle query is made), whereas in the sequence of game hops after replacing cipher calls with freshly generated randomness, it is primarily during the proxy's encryption. Thus a furhter sequence of game hops reconciles the two. Although in hindsight, none of the steps is particularly challenging on its own, finding the right sequence of steps was not straightforward for various reasons.

Related Works. Two prior works [9, 17] analysed the security of Mathewson's proposal 261, containing a scheme based on a strong pseudorandom permutation [14]. These two analyses were concurrent and independent, using rather different security models that both have their limitations (see below). Consequently, our treatment of Tor's onion encryption is the first to consider leaky pipes and forward security.

On the one hand, Rogaway and Zhang [17] focus solely on the onion encryption component, providing an all-in-one security definition that is derived directly from the desired functionality using the *silencing approach* [18]. A critical role in this approach is the correctness of the scheme, which is used to establish an ideal, best-achievable version of the primitive, as well as all sequences of adversarial queries that would lead to trivial wins (based on the correctness notion) and hence have to be silenced (i.e. suppressed). The

advantage of Rogaway and Zhang's silencing approach is that the overall definition is relatively compact, but a potential disadvantage is that that compactness makes the resulting security intuitively less clear.

Our security model OAEF_ℓ is loosely based on Rogaway and Zhang's all-in-one security notion, but rather than using the silencing approach and relying on an exact correctness definition, we explicitly spell out the flow of our distinguishing game. The explicit nature of our game is critical, as it allows us to deviate from their definition in two important ways. Firstly, our model supports leaky pipes, which complicates matters significantly as correctness is no longer perfect, which (at the very least) hampers the application of the silencing approach. Secondly, we model partial forward security through corruptions, which is not included by Rogaway and Zhang. Their main security notion and subsequent security analysis does not model node corruptions, however, they do describe an extension of their security model that includes static node corruptions *prior* to any oracle queries. Here the corruptions affect the ideal functionality of the scheme, making the silencing approach suitable. Evidently, for forward security, one would have to allow corruptions *after* the oracle queries (as we do), which would not affect functionality at all, and hence suitability of the silencing approach is unclear, especially as we only desire partial forward security (namely of end-to-end confidentiality and not of intermediate routing anonymity).

On the other hand, Degabriele and Stam [9] consider the security of onion routing at large, formulating confidentiality, authenticity, and anonymity as separate security notions. The routing mechanism, which affects security, is considered explicitly as part of the scheme. Arguably, their security model is closer to the intuitive security goal, but it is also more complex. For instance, onion routing derives its security in practice from hiding among the masses, in the sense that anonimity can only be achieved if enough users send messages more or less simultaneously. Degabriele and Stam's security model explicitly incorporates said aspect, by restricting an adversary to batches of queries to avoid trivial wins based on routing information leaking through traffic analysis. Furthermore, their model contains neither leaky pipes nor corruptions (and hence no forward security).

2 Preliminaries

Notation. Our notational conventions are fairly standard. In our code-based experiments, we deploy some simple shorthands for assignments, especially involving lists or sets. Namely, we use the shorthand $\mathcal{X} \xleftarrow{\cup} x$ for the operation $\mathcal{X} \leftarrow \mathcal{X} \cup \{x\}$ and $\mathtt{X} \xleftarrow{\frown} x$ to append the element x to the list $\mathtt{X}$. The shorthand $(B, C) \twoheadleftarrow A$ indicates that A is parsed as the tuple (B, C). By convention all sets, lists, and lazy functions are initialized empty.

For stateful algorithms, we use bracketed subscripts to implicitly allow state updates. As an example, in Fig. 3 we write $C_{i+1} \leftarrow \mathsf{OR\text{-}Dec}^{AD}_{\langle \varrho_i \rangle}(C_i)$, which means that the algorithm $\mathsf{OR\text{-}Dec}$ takes ϱ_i as state variable and the values AD and C_i as regular inputs. The algorithm's output is assigned to C_{i+1}, but underneath the hood, the algorithm may also change the state variable C_i as part of its processing.

2.1 Updatable Tweakable Split-Domain Ciphers

The conceptual building block behind CGO is an updatable tweakable split-domain cipher. The domain of these ciphers is split into two sets, so $\mathcal{D} = \mathcal{D}_L \times \mathcal{D}_R$, referred to as the left set ($\mathcal{D}_L$), respectively the right set ($\mathcal{D}_R$); in the context of CGO, and henceforth for us, $\mathcal{D}_L = \{0,1\}^n$ and $\mathcal{D}_R = \{0,1\}^m$ with $0 < n < m$. An updatable tweakable split-domain cipher is characterized by its enciphering algorithm

$$\mathsf{EE} : \{0,1\}^k \times \mathcal{H} \times \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n \times \{0,1\}^m \ ,$$

and update algorithm

$$\mathsf{EU} : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^k \times \{0,1\}^n \ .$$

Here, for any $K \in \{0,1\}^k$ and any $H \in \mathcal{H}$, the function EE^H_K identifies a permutation over $\{0,1\}^n \times \{0,1\}^m$, whose inverse we denote by ED^H_K. The update algorithm EU takes a key $K \in \{0,1\}^k$ and a left element $X_L \in \{0,1\}^n$, and returns new values for each, without any structure imposed.

Tweakable split-domain ciphers (without the update functionality) were introduced by Degabriele and Karadžić [7] under the nomer *rugged pseudorandom permutations* (RPRP). Technically, RPRP is a new security notion for split-domain ciphers that is stronger than standard PRP (pseudorandom permutations [11]), yet weaker than sPRP (strong pseudorandom permutations [12]), as the adversary has full access to the enciphering algorithm but only restricted access to the deciphering algorithm. Degabriele et al. [8] build on the RPRP notion and considera more involved CRND_ℓ notion, where an

Experiment $\mathsf{Exp}^{\mathrm{CRND}_\ell\text{-}b}_{\mathsf{EE},\mathsf{EU}}(\mathbb{A})$

for $h = 1$ **to** ℓ
  $K_{h[0]} \xleftarrow{\$} \{0,1\}^k$
  $fresh_h \leftarrow \mathsf{true}$
  $i_h \leftarrow 0$
$\hat{b} \leftarrow \mathbb{A}^{\mathrm{En,De,Gu,Up,Co}}$
return $\hat{b}$

$\mathrm{En}(h, j, H, X_L, X_R)$

if $(j > i_h) \vee (j = i_h \wedge h \in \mathcal{Z})$
  return $\lightning$
if $j = i_h$
  $fresh_h \leftarrow \mathsf{false}$
if $b = 1$
  $(Y_L, Y_R) \leftarrow \mathsf{EE}^H_{K_{h[j]}}(X_L, X_R)$
else
  $(Y_L, Y_R) \xleftarrow{\$} \mathcal{D}_L \times \mathcal{D}_R$
$\mathcal{F}^j_h \xleftarrow{\cup} Y_L,\ \mathcal{U}^j_h \xleftarrow{\cup} (H, Y_L, Y_R)$
return (Y_L, Y_R)

$\mathrm{Gu}(h, j, H, Y_L, Y_R, X'_L)$

if $(j > i_h) \vee (j = i_h \wedge h \in \mathcal{Z}) \vee (H, Y_L, Y_R) \in \mathcal{U}^j_h$
  return $\lightning$
if $j = i_h$
  $fresh_h \leftarrow \mathsf{false}$
if $b = 1$
  $(X_L, X_R) \leftarrow \mathsf{ED}^H_{K_{h[j]}}(Y_L, Y_R)$
  return $X_L = X'_L$
else
  return false

$\mathrm{De}(h, j, H, Y_L, Y_R)$

if $(j > i_h) \vee (j = i_h \wedge h \in \mathcal{Z}) \vee (Y_L \in \mathcal{F}^j_h)$
  return $\lightning$
if $j = i_h$
  $fresh_h \leftarrow \mathsf{false}$
if $b = 1$
  $(X_L, X_R) \leftarrow \mathsf{ED}^H_{K_{h[j]}}(Y_L, Y_R)$
else
  $(X_L, X_R) \xleftarrow{\$} \mathcal{D}_L \times \mathcal{D}_R$
$\mathcal{F}^j_h \xleftarrow{\cup} Y_L,\ \mathcal{U}^j_h \xleftarrow{\cup} (H, Y_L, Y_R)$
return (X_L, X_R)

$\mathrm{Up}(h, T)$

if $h \in \mathcal{Z}$
  return $\lightning$
if $b = 1$
  $(K_{h[i_h+1]}, T) \leftarrow \mathsf{EU}_{K_{h[i_h]}}(T)$
else
  $K_{h[i_h+1]} \times T \xleftarrow{\$} \{0,1\}^k \times \mathcal{D}_L$
$i_h \leftarrow i_h + 1, fresh_h \leftarrow \mathsf{true}$
return T

$\mathrm{Co}(h)$

if $(h \in \mathcal{Z}) \vee (fresh_h = \mathsf{false})$
  return $\lightning$
$\mathcal{Z} \xleftarrow{\cup} h$
return $K_{h[i_h]}$

Fig. 1. The $\mathsf{Exp}^{\mathrm{CRND}_\ell\text{-}b}_{\mathsf{EE},\mathsf{EU}}(\mathbb{A})$ experiment used to define CRND_ℓ security instances for ℓ independent instances of an updatable tweakable split-domain cipher with continuous key updates. (Figure reproduced from [8, Fig. 5].)

adversary can ask for keys to be updated and then query the enciphering and deciphering algorithm on those updated keys.

Essentially, in the CRND_ℓ game (reproduced in Fig. 1), there are ℓ independent instances of the updatable split-domain tweakable cipher. Think of these instances of each representing a Tor circuit. The adversary can trigger an arbitrary number of key updates for any cipher (think circuit) of its choosing, each time receiving the updated T. To keep track of the latest version of a cipher instance's key, the index i_h is used and when the adversary wants to make an encryption or decryption query (using any of the oracles $\text{En}, \text{De}, \text{Gu}$), it has to specify both an instance handle h and an index j to the oracle. Additionally, an adversary can use the oracle Co to corrupt any cipher instance h, thus obtaining the *latest* key version, namely $K_{h[i_h]}$. To avoid trivial wins, the adversary is restricted to only corrupting cipher instances whose latest key is still fresh (has not been used yet by En, De, or Gu) and, once corrupted, that latest key can no longer be used (specifically, that cipher instance can no longer be updated). The CRND_ℓ advantage is defined as a standard distinguishing advantage based on the CRND_ℓ game.

Definition 1 (CRND_ℓ **Advantage (Reproduced from [8, Def. 1]))**. *For any tweakable split-domain cipher* EE *over* $\mathcal{D}_L \times \mathcal{D}_R$ *and any associated update function* EU*, the corresponding* CRND_ℓ *advantage of an adversary* $\mathbb{A}$ *is given by:*

$$\mathsf{Adv}^{\text{CRND}_\ell}_{\mathsf{EE},\mathsf{EU}}(\mathbb{A}) = \Pr\left[\mathsf{Exp}^{\text{CRND}_\ell\text{-}0}_{\mathsf{EE},\mathsf{EU}}(\mathbb{A}) = 1\right] - \Pr\left[\mathsf{Exp}^{\text{CRND}_\ell\text{-}1}_{\mathsf{EE},\mathsf{EU}}(\mathbb{A}) = 1\right] ,$$

where $\mathsf{Exp}^{\text{CRND}_\ell\text{-}b}_{\mathsf{EE},\mathsf{EU}}(\mathbb{A})$ *is defined in Fig. 1.*

3 Onion Encryption and CGO

Notation. For ease of comparison, we use the same notation as introduced by Degabriele et al. [8]. We briefly give an overview of its main tenets, restricted to the forward direction (as we only analyse that direction in Sections 4.2 and 5).

For a circuit of length ℓ, the OR nodes are labelled $1, \ldots, \ell$, where the OP will send its ciphertext (cell really) to OR 1 and OR ℓ is the last node in the circuit. If we think of the OP as a node with index 0, then for any node $i > 0$ within a circuit we refer to node $i - 1$ as its predecessor and node $i + 1$ as its successor (clearly, the last node ℓ has no successor). Ciphertexts between node $i - 1$ and i may be indexed by i, so in the forward direction router i will take as input C_i whereas in the backward direction router i will output C_i.

The messages M that a proxy can send are in the message space $\mathcal{M}$, whereas the ciphertexts exchange by the routers (or sent by the proxy to the first router) are in the ciphertext space $\mathcal{C}$, where $\mathcal{M}$ and $\mathcal{C}$ are disjoint and thus messages can easily be distinguished from ciphertexts (typically through their length). Associated data lives in the space $\mathcal{AD}$.

Syntax for the Forward Direction. A forward-direction onion encryption scheme $\mathsf{OE\text{-}Fw}$ consists of three algorithms $(\mathsf{Init\text{-}Fw}, \mathsf{OP\text{-}Enc}, \mathsf{OR\text{-}Dec})$. The descriptions of these algorithms below are copied verbatim from Degabriele et al. [8, Section 3.1].

- "The circuit initialisation algorithm $\mathsf{Init\text{-}Fw}$ takes as input the circuit size ℓ and returns the initial encryption state $\boldsymbol{\sigma}$ intended for the proxy (which consists of ℓ components σ_i, one for each router on the circuit), as well as the initial decryption states $\varrho_1, \ldots, \varrho_\ell$ for each router in the circuit."
- "The deterministic algorithm $\mathsf{OP\text{-}Enc}$ is used by the proxy to encrypt messages with associated data to a specific node within a circuit. Given the current encryption state $\boldsymbol{\sigma}$ for a circuit, the index $d \in [\ell]$ of the destination node within that circuit, associated data $AD \in \mathcal{AD}$, and a message $M \in \mathcal{M}$, the call $C_1 \leftarrow \mathsf{OP\text{-}Enc}^{AD}_{\langle\boldsymbol{\sigma}\rangle}(d, M)$ returns the ciphertext $C_1 \in \mathcal{C}$ intended for the first available router; as a side-effect the algorithm $\mathsf{OP\text{-}Enc}$ may update its state $\boldsymbol{\sigma}$."
- "The deterministic algorithm $\mathsf{OR\text{-}Dec}$ is used by the routers in the circuit to process incoming ciphertexts. It takes as input the node's decryption state ϱ_i, the associated data AD, and a ciphertext $C_i \in \mathcal{C}$ to update its state and output $C_{i+1} \in \mathcal{M} \cup \mathcal{C}$. If $C_{i+1} \in \mathcal{M}$ the output is deemed intended for the local node, otherwise it is supposed to be forwarded to the next node in the circuit."

Some minor remarks concerning syntax choices. Firstly, the assumption implicit in the circuit initialisation is that somehow the circuit setup phase including key exchange takes place in one go. In reality,

$\underline{\mathsf{Init\text{-}Fw}(\ell)}$

for $i = 1$ **to** ℓ
  $K \leftarrow\!\!\$\ \{0,1\}^k$
  $N \leftarrow\!\!\$\ \{0,1\}^n$
  $T' \leftarrow 0^n$
  $\sigma_i \leftarrow (K, N, T')$
  $\varrho_i \leftarrow (K, N, T')$
$\boldsymbol{\sigma} \leftarrow (\sigma_1, \ldots, \sigma_\ell)$
return $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell)$

$\underline{\mathsf{OP\text{-}Enc}^{AD}_{\langle \boldsymbol{\sigma} \rangle}(d, M)}$

if $(d > \ell)$ **return** $\perp$
for $i = d$ **to** 1
  $(K, N, T') \twoheadleftarrow \sigma_i$
  $H \leftarrow T' \parallel AD$
  if $i = d$ // initialize top layer
    $(T, \overline{C}) \leftarrow (N, M)$
  $T' \leftarrow T$
  $(T, \overline{C}) \leftarrow \mathsf{ED}^H_K(T, \overline{C})$
  if $i = d$
    // update destination's key
    $(K, N) \leftarrow \mathsf{EU}_K(N)$
  $\sigma_i \leftarrow (K, N, T')$
return $T \parallel \overline{C}$

$\underline{\mathsf{OR\text{-}Dec}^{AD}_{\langle \varrho \rangle}(T \parallel \overline{C})}$

$(K, N, T') \twoheadleftarrow \varrho$
$H \leftarrow T' \parallel AD$
$(T, \overline{C}) \leftarrow \mathsf{EE}^H_K(T, \overline{C})$
if $T = N$
  // ciphertext recognised
  $(K, N) \leftarrow \mathsf{EU}_K(N)$
  $X \leftarrow \overline{C}$
else
  // forwarding ciphertext
  $X \leftarrow T \parallel \overline{C}$
$\varrho \leftarrow (K, N, T)$
return X

Fig. 2. CGO in the forward direction. (Figure reproduced from [8, Fig. 2].)

Tor uses a telescoping mechanism to slowly build circuits, involving quite some communication and usage of the circuit-so-far in the process. Nonetheless, pretending that all parties' states are initialized simultaneously (without further communication necessary) is a convenient simplification, that is quite common in theory [9,17], without imposing any limitations in practice.

Secondly, ciphertext rejections are not modelled explicitly, as they are neither needed nor desired for CGO. Intermediate routers do not reject ciphertexts, but rather forward a ciphertext to the next router (if they do not recognize the ciphertext as intended for them). For the final router in the circuit, forwarding is admittedly not an option, but in that case rejection can take place at the routing level (so outside of the cryptographic processing modelled here).

3.1 CGO's Specification

Although CGO was explicitly designed to be used in Tor, its high-level description adheres to the abstract syntax of the previous section. Admittedly, the syntax itself is already tailored to support several Tor-centric features, but it is only in its instantiation that CGO becomes truly Tor specific. For our analysis of CGO we can remain at a high level. As we will only model and prove security of the forward direction (where the proxy sends messages to various routers in the circuit), we will only describe that component.

A high-level pseudocode description of CGO in the forward direction is provided in Fig. 2. As promised, this high-level description is stated in terms of an updatable tweakable split-domain cipher EE with update function EU. The message space $\mathcal{M} = \{0,1\}^m$ and each ciphertext consists of two strings T and $\overline{C}$ of sizes n and m respectively (thus $\mathcal{C} = \{0,1\}^{n+m}$; for completeness, $\mathcal{H} = \{0,1\}^n \times \mathcal{AD}$).

4 Modelling Tor's Onion Encryption (Forward Direction)

Mathewson [13] already laid out a number of design desiderata for onion encryption. CGO's designers, when discussing CGO's design rationale, already highlighted various security requirements, specifically that it should provide a forward secure channel between senders and receivers while also being resistant against tagging attacks. However, for the latter property, forward security is not required, or even desired if it would increase overhead. Here, we provide a formalization of the forward direction by stipulating a non-trivial correctness notion, followed by a detailed mapping of the security desiderata onto a game-based security notion. We leave a similar treatment of the backward direction to future work.

4.1 Correctness and Symmetry

Perfect correctness states that when everyone is behaving honestly, the intended functionality is always achieved. In the context of Tor's forward direction, this would imply that all messages arrive as sent at the intended destination. In practice, proposals for Tor's onion encryption do not achieve perfect correctness

Experiment $\mathsf{FwCorrect}_{\mathsf{OE\text{-}Fw}}\left(\ell, (d_j, AD_j, M_j)_{j\in[N]}\right)$

$(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
if $\boldsymbol{\sigma} \neq (\varrho_1, \ldots, \varrho_\ell)$ **then return** bad_{init}
for $j = 1$ **to** N
  $d \leftarrow d_j, AD \leftarrow AD_j$
  $C_1 \leftarrow \mathsf{OP\text{-}Enc}^{AD}_{\langle\boldsymbol{\sigma}\rangle}(d, M_j)$
  $i \leftarrow 1$
  while $C_i \in \mathcal{C} \wedge i < d$
    $C_{i+1} \leftarrow \mathsf{OR\text{-}Dec}^{AD}_{\langle\varrho_i\rangle}(C_i)$
    $i \leftarrow i + 1$
  if $C_i \in \mathcal{M}$
    return $bad_{drop-(j,i)}$
  else
    $X \leftarrow \mathsf{OR\text{-}Dec}^{AD}_{\langle\varrho_d\rangle}(C_d)$
    if $X \neq M_j$ **then return** bad_{plain}
    if $\boldsymbol{\sigma} \neq (\varrho_1, \ldots, \varrho_\ell)$ **then return** bad_{asym}
return true

Fig. 3. Symmetric correctness for forward-direction onion encryption.

in the forward direction, primarily due to the need to support leaky pipes. We can distinguish between three possible violations of correctness that might occur: (1) a cell is overeagerly "accepted" by an intermediate router, effectively resulting in the cell being dropped from the circuit; (2) the destination router does not recognize the cell as being intended for it; or (3) the destination router recognizes the cell but fails to retrieve the correct message. In our correctness experiment $\mathsf{FwCorrect}_{\mathsf{OE\text{-}Fw}}$ (Fig. 3) we model events (2) and (3) by terminating with output bad_{plain} whenever the received "message" X by the intended destination does not match the sent message M_j. Event (1) is caught by returning $bad_{drop-(j,i)}$, indicating which cell (j) was dropped where (i). For a scheme to be correct we will insist events (2) and (3) do not occur, but we will allow event (1) with a small probability.

Tor has a further property that the state of the proxy is essentially the combination of all the routers' states, so $\boldsymbol{\sigma} = (\varrho_1, \ldots, \varrho_\ell)$. Throughout the protocol, this property is held as an invariant as long as all the routers are fully in-sync, meaning that all the ciphertexts created by the proxy have been fully processed, in the correct order, by the routers (up to their destinations, assuming no drops and no other ciphertexts have been processed by any of the routers). So far, our syntax does not prescribe this high-level symmetric structure of the state. However, our symmetric correctness game does include a check that that symmetry between the proxy and the routers is created and maintained, returning bad_{init}, respectively bad_{asym}, if not.

Consequently, in an honest scenario, the proxy can always reconstruct the states of the routers at any given point and thus predict exactly how its ciphertexts will be passed along. This predictability is useful for formalizing security games and has been used in prior work related to onion encryption [5,9,17]. The symmetry between the proxy on the one hand and the collection of routers on the other, effectively rules out public key solutions and has further security ramifications, especially when considering corruptions.

Definition 2 (Symmetric Correctness). *A forward-direction onion encryption scheme* $\mathsf{OE\text{-}Fw}$ *is said to be* δ*-correct, if for any* $N \geq 1$*, any* $\ell \geq 1$*, and any* $(d_j, AD_j, M_j)_{j\in[N]} \in ([\ell] \times \mathcal{AD} \times \mathcal{M})^N$*, and any* $j' \in [N]$*,* $i \in [\ell]$ *it holds that:*

$$\Pr\left[\mathsf{FwCorrect}_{\mathsf{OE\text{-}Fw}}\left(\ell, (d_j, AD_j, M_j)_{j\in[N]}\right) \in \{bad_{plain}, bad_{init}, bad_{asym}\}\right] = 0$$
$$\Pr\left[\mathsf{FwCorrect}_{\mathsf{OE\text{-}Fw}}\left(\ell, (d_j, AD_j, M_j)_{j\in[N]}\right) = bad_{drop-(j',i)}\right] \leq \delta\,,$$

where $\mathsf{FwCorrect}$ *is the game defined in Fig. 3.*

Once event (1) happens, the router responsible for dropping the cell can no longer be guaranteed to process any subsequent cells correctly. Although the circuit up to that point could still perform as before, we do not insist so.

Remark 1. A subtlety with our correctness definition is that, even for small $\delta > 0$, ciphertext-dependent messages may still not be delivered with overwhelming probability. As a consequence of the quantification, once the initial state $\boldsymbol{\sigma}$ (and with it the ϱ_i) is fixed, there will most likely exist some bad (d_j, AD_j, M_j) sequences (that trigger some $bad_{drop-\ldots}$), yet for any given sequence, the probability of generating an initial state for which that particular sequence is bad, can still be low (and upper bounded by δ). Although one could strengthen the correctness definition, for instance by allowing an adversary to select the (d_j, AD_j, M_j) adaptively, we chose to stick with the simpler option for correctness and treat the adaptive case in conjunction with our upcoming security notion.

The correctness and symmetry of CGO. By inspection, one can verify that for CGO's forward direction (Fig. 2) the flags bad_{plain}, bad_{init}, and bad_{asym} never get set. However, the dropping of ciphertexts resulting in the setting of some $bad_{drop-\ldots}$ can occur. In fact, the proxy can already predict when a drop will occur, namely only when the processing $(T, C) \leftarrow \mathsf{ED}_K^H(T, C)$ for one layer i results in a T that happens to equal the N of the preceding layer $i-1$, i.e., when after $(K, N, T') \twoheadleftarrow \sigma_{i-1}$, $N = T$, then the $i-1$'th router is guaranteed to drop the ciphertext (if it has not been dropped already).

The easiest way to bound the probability of the event $bad_{drop-(j,i)}$ is by treating the tweakable blockcipher E inside the UIV+ construction used to instantiate EE (and, by extension, ED) as ideal. In that case, $bad_{drop-(j,i)} = 2^{-n}$, so CGO satisfies symmetric correctness with $\delta = 2^{-n}$ in the ideal cipher model.

Remark 2. The way we model the states $\boldsymbol{\sigma}$ and $\varrho_1, \ldots, \varrho_\ell$, together with their initialisation by Init-Fw would allow routers to be aware of the length of and their own position in the circuit (so the cryptographic processing could in principle depend on it). However, we do not mandate such awareness, nor is it beneficial (let alone necessary) to meet our security notion. For instance, for Tor, Init-Fw will actually run ℓ independent copies of an initialization algorithm to obtain the ϱ_i and subsequently $\boldsymbol{\sigma}$ is set to the union of all the ϱ_is. Although we model the relation between $\boldsymbol{\sigma}$ and the ϱ_i (under symmetry), we care not whether ϱ_i itself depends on the values of i or ℓ).

4.2 Security

Once a circuit has been established, we expect that messages from the proxy to any of the routers remain confidential; moreover, routers should only accept messages as sent by the router (and in the correct order). These are the classical security properties of both stateful authenticated encryption [3,15] and secure channels [6,15], where a crucial difference between their respective, modern security notions is that the former is typically stated in terms of ciphertexts whereas the latter is more plaintext oriented. For secure onion encryption we also expect anonymity, in the sense that ciphertexts progressing through the network cannot be linked (as belonging to the same circuit), bringing us closer to a ciphertext-oriented notion: the processing of ciphertexts by a single intermediate honest router should look random.

Furthermore, we want to limit the damage when routers get compromised. In particular, when compromising a router, messages received in the past should remain hidden; in other words, we expect forward security for confidentiality. As compromising a router will likely reveal the routing information held by that router (at a higher level of the protocol), we need not care about forward security of the anonymity, nor do we worry about forward security of integrity.

Bearing the above considerations in mind, we carefully craft a game-based indistinguishability notion that models the particular security we have in mind, nothing more and nothing less. The formal definition is given below, based on the $\mathsf{Exp}^{\mathsf{OAEF}_\ell\text{-}b}_{\mathsf{OE\text{-}Fw}}(\mathbb{A})$ experiments provided in Fig. 4, where OAEF_ℓ stands for "Onion Authenticated Encryption, Forward" and ℓ is a parameter indicating the circuit length. An explanation of the rationale behind the game follows.

Definition 3. *The* OAEF_ℓ *advantage of an adversary* $\mathbb{A}$ *relative to a simulator* Sim *against a forward onion encryption scheme* OE-Fw *with a circuit of length* ℓ *is defined as*

$$\mathsf{Adv}^{\mathsf{OAEF}_\ell}_{\mathsf{OE\text{-}Fw},\mathsf{Sim}}(\mathbb{A}) = \Pr\Big[\mathsf{Exp}^{\mathsf{OAEF}_\ell\text{-}0}_{\mathsf{OE\text{-}Fw},\mathsf{Sim}}(\mathbb{A}) = 1\Big] - \Pr\Big[\mathsf{Exp}^{\mathsf{OAEF}_\ell\text{-}1}_{\mathsf{OE\text{-}Fw}}(\mathbb{A}) = 1\Big] \ ,$$

where $\mathsf{Exp}^{\mathsf{OAEF}_\ell\text{-}b}_{\mathsf{OE\text{-}Fw}}(\mathbb{A})$ *is given in Fig. 4.*

The distinguishing experiments depend on a bit b which the adversary tries to guess using its three oracles: in a first stage, $\mathbb{A}$ has access to an encryption oracle ENC to model the proxy and a decryption

oracle DEC to model the routers, and in a second stage its corruption oracle COR models the compromise of routers (we do not allow the proxy to be compromised). If $b = 1$, the experiment corresponds to the 'real' world and, in principle, the oracles return the proper outputs, resp. states, of the various parties (code highlighted in green). If $b = 0$, the experiment corresponds to an 'ideal' world where outputs are random strings and compromised states are appropriately simulated (code highlighted in purplish blue).

The first stage. An adversary with only access to the ENC-oracle corresponds to a relatively passive 'IND$–CPA' adversary [16]. In the real world, such an adversary can ask for a sequence of messages to be sent from the proxy to various destinations and subsequently observe the ciphertexts being communicated between the proxy and the first router (or guard node in Tor parlance). In contrast, in the ideal world the adversary instead gets to see ciphertexts that were uniformly and independently drawn from the ciphertext space. Thus, in the ideal world the ciphertexts provide no information about either the contents of the fixed-length message or its intended destination. This property is inherited by the real world if an adversary cannot tell the two worlds apart: the circuit therefore provides both receiver and message confidentiality.

An adversary with additional access to the DEC-oracle is closer to a typical, active 'IND$-CCA' adversary: it can query each router individually and adaptively, learning the output of its processing. In the ideal world, all the ciphertexts look random so it is not clear a priori when the destination has been reached. In contrast, in the real world, an adversary could progress the ciphertext output by the proxy until it eventually reaches its destination, resulting in the original message. To avoid trivial wins, we need to adjust our game accordingly.

To this end, we extend the notion of in-sync from stateful encryption [3] to our setting with multiple routers (code highlighted in red). For each router h, a queue $\mathtt{Q}_h$ keeps track of destination–associated-data–ciphertext triples (d, AD, C_h) yet to be processed by router h. Assuming honest behaviour throughout, a router dequeues (d, AD, C_h), processes C_h under AD yielding C_{h+1} and, if relevant, enqueues (d, AD, C_{h+1}) to $\mathtt{Q}_{h+1}$. Even in the ideal world, where the ciphertexts C_h and C_{h+1} are randomly and independently generated (the purplish blue code), the d-part in $\mathtt{Q}_h$'s triplet allows the game to check whether an in-sync ciphertext has reached its intended destination d. Once an active adversary deviates from simply forwarding ciphertexts and instead modifies or injects them, a router h gets out-of-sync, which the game captures by setting the flag $sync_h$ to false (by inspection, it can never return to true).

For in-sync destinations, the output is always suppressed, both in the real and ideal world, to avoid trivial wins.[3] For in-sync intermediate routers, the output is added to the queue for the next router before being returned to the adversary: these ciphertexts should look random to an adversary in the ideal world to model that they do not leak information. Finally, for out-of-sync routers, the output will no longer be suppressed, nor will the queue for its successor router still be added to. Effectively, the successor of an out-of-sync router is automatically deemed out-of-sync once it would receive the output from its out-of-sync predecessor.

How integrity is captured. Our mechanism simultaneously captures integrity, as in the ideal world a router will never accept an adversarially generated ciphertext as intended for itself (it will simply output a randomly sampled ciphertext from the ciphertext space), whereas in the real world an integrity break would correspond to an output in the message space instead (leading to an easy distinction). Similarly, the accidental dropping of honestly processed ciphertexts (due to a leaky-pipe) is treated as an integrity violation. As we do not model any explicit rejection, the return of a ciphertext instead (except for an in-sync destination) is similar to the release of unverified plaintext [1,2], explaining the intuition behind Proposal 295 [19].

How tagging attacks are captured. The ability to inject and modify traffic suffices to model tagging attacks, for instance against the current implementation of Tor based on the highly malleable counter mode. For a typical length-3 circuit, an adversary controlling the guard node and exit node can use a tagging attack to ascertain that those two nodes are routing the same circuit. The guard sends the middle router a modified cell, which in our game will force the middle router to go out-of-sync. Consequently, the middle router will directly return the processed modified cell without updating the queue of the exit node. In the real world, the malleability of counter mode allows the adversary to untag the processed modified cell to obtain the cell as the proxy intended it to be for the exit node; however, crucially this 'correct' cell is not in the exit node's queue and hence processing it will force the exit node out-of-sync

[3] Alternatively, one could store quadruples (M, d, AD, C_h) in $\mathtt{Q}$ and return M.

too and it returns the message (as sent by the proxy). In contrast, in the ideal world, the out-of-sync exit node will return a random element in the cell space, which is clearly distinct from a message.

The second stage. During the second stage, an adversary can use its COR oracle to corrupt routers, receiving a router's internal state ϱ_h in the real world. In the ideal world, routers do not have an internal state and instead the game uses Sim to simulate one instead (the purple blue box). To facilitate this simulation, the game maintains a list $\mathtt{C}_h$ of past, incoming ciphertexts. As corruptions serve to model forward security, we only allow them after all ENC and DEC queries have been made during Stage 1 (and see Remark 5). Still, we need to restrict which routers can be corrupted for two reasons.

Firstly, after corrupting a router h and receiving ϱ_h, an adversary can run $\mathsf{OR\text{-}Dec}^{\cdot}_{\langle \varrho_h \rangle}(\cdot)$ on its own, essentially enabling it to run the DEC oracle without any safeguards. As an adversary can, just as easily as the game, keep track of the queues $\mathtt{Q}_h$ and, after corrupting an in-sync router h, it can subsequently process said queue $\mathtt{Q}_h$ locally. Thus, if the queue contains any ciphertexts with h itself as destination, the corruption would result in a trivial win (and hence needs to be disallowed). Furthermore, if the queue contains a destination beyond h but with all further intermediate routers and that destination already corrupted, we have to assume the adversary can also decrypt that ciphertext all the way to its destination, again yielding a trivial win.

Essentially, based on $\mathtt{Q}_h$, or an in-sync router's future, a corruption might have to be disallowed. The COR-oracle explicitly checks the queue $\mathtt{Q}_h$ to determine whether h can be corrupted or not (the first orange box). If it can, that implies that all elements remaining in $\mathtt{Q}_h$ have to be processed by a further honest router before (or as) their destination. Once h is corrupted, that first honest router following h is the next line of defence and its queue needs to be updated with the $\mathtt{Q}_h$ (the second orange box).

Secondly, we purely desire forward confidentiality yet not forward anonymity. If an in-sync router has most recently processed a ciphertext not destined for itself (and hence forwarded the result), we do not care if subsequent corruption of that router reveals that incoming and outgoing ciphertexts are somehow related (thereby revealing the routing). As in the ideal world incoming and outgoing ciphertexts are, by construction, independenly sampled at random (and hence unrelated), we will have to pose limits on corruptions instead.

Essentially, based on an router's past as determined by its DEC-oracle queries and answers, corruptions might have to be disallowed. Concretely, each DEC-oracle call (re)sets the flag $corr_h$ depending on the flow. If the DEC-oracle results in an element from the message space $\mathcal{M}$, then we set $corr_h$ to true, otherwise to false. A slight complication occurs for in-sync destination, where we suppress the message in both the real and ideal worlds, but where crucially we treat the output as an element from $\mathcal{M}$. For completeness and by convention, all flags $corr_h$ are set to true at initialisation.

How forward security is captured. In the ideal world, the state received by an adversary after corruption is, by design, independent of the messages that were received by that router. More precisely, the full view of the adversary after corruptions is. Hence, corruptions cannot reveal information about past messages. indistinguishability between the ideal and real worlds ensures that property carries over to the real world.

Remark 3. In principle, when simulating a corrupted state, the simulator should have access to all information that is allowed to leak, nothing more and nothing less. To assure forward security of confidentiality (but not anonymity), this means all the ciphertexts that were input or output by a router, but not any of its received messages. Thus, in the ideal world, the router's state will be simulated based on a sequence of randomly chosen ciphertexts (independent of the message or destination) and adversarially chosen associated data (one small caveat is that to preserve security, the protocol's use of associated data at a higher level should not leak information about either destination or the underlying message, cf. [4]).

Remark 4. In principle, our game allows an adversary to corrupt out-of-sync routers. However, we argue that an adversary gains no benefit from doing so, in the sense that an adversary successfully corrupting an out-of-sync router could have gained the same (or better) distinguishing advantage without said corruption. Principally, the game's checks whether to allow a corruption or not can all be performed locally by the adversary, thus whether or not a corruption succeeds conveys no information. Furthermore, in the ideal world an out-of-sync router can never be corrupted, which follows by inspection (in the DEC-oracle, the $corr_h$ flag will always be set to false prior to determining syncness and, as the router is out-of-sync, cannot be reset to true after). Conversely, in the real world an out-of-sync router can only be corrupted if a prior, out-of-sync DEC-query results in a message (i.e., an element of $\mathcal{M}$) being output. Such an integrity break is easy to spot by an adversary and, as it cannot occur in the ideal world, is a

sure-fire distinguishing event. In other words, without loss of generality we may assume that an adversary does *not* corrupt any out-of-sync routers.

Remark 5. Our game does not allow *fully* adaptive corruptions that are interleaved with the ENC and DEC oracles. Allowing such adaptive corruptions would yield a stronger security notion and an argument can be made why such a stronger security notion would be desirable: ideally, communication between the proxy and an honest router should remain authenticated and confidential even if all intermediate routers are dishonest, which can be modelled by allowing corruption of those intermediate routers prior to any relevant ENC calls to the proxy; furthermore, in order for a real-life Tor adversary to inject and modify traffic between routers, it will either need to break the link encryption (TLS), or more realistically, control the router entirely, which again would be most adequately captured by fully adaptive corruptions.

Experiment $\mathsf{Exp}^{\mathsf{OAEF}_\ell\text{-}b}_{\mathsf{OE\text{-}Fw}}(\mathbb{A})$

$\forall i\ sync_i \leftarrow \mathsf{true}, corr_i \leftarrow \mathsf{true}$
if $b = 1$
  $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
$b' \leftarrow \mathbb{A}^{\mathsf{ENC},\mathsf{DEC},\mathsf{COR}}()$
return b'

oracle $\mathsf{ENC}(d, AD, M)$

// Disallow proxy queries after corruptions
if $\mathcal{Z} \neq \emptyset$ **then return** $\lightning$
if $b = 0$
  $C_1 \leftarrow\!\!\$\ \mathcal{C}$
else
  $C_1 \leftarrow \mathsf{OP\text{-}Enc}^{AD}_{\langle\boldsymbol{\sigma}\rangle}(d, M)$
// Admin and output decision by 'ghost' macro
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{COR}(h)$

if $h \in \mathcal{Z}$ **then return** $\lightning$
// Can only corrupt certain routers
$corr \leftarrow corr_h$
// Determine next honest router
$i^* \leftarrow h + 1$
while $i^* \leq \ell \wedge i^* \in \mathcal{Z}$
  $i^* \leftarrow i^* + 1$
if $\exists (d, AD, C) \in \mathtt{Q}_h$ s.t. $d < i^*$ **then** $corr \leftarrow \mathsf{false}$
if $corr = \mathsf{false}$ **then return** $\lightning$
// Successfully corrupting a router
$\mathcal{Z} \xleftarrow{\cup} h$
// Simulate router's state in the random world
if $b = 0$ **then** $\varrho_h \leftarrow \mathsf{Sim}(h, \mathtt{C}_h)$
// Update next honest router's queue
if $i^* \leq \ell$
  while $\mathtt{Q}_h \neq \emptyset$
    $(d, AD, C_i) \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    $\mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
return ϱ_h

oracle $\mathsf{DEC}(h, AD, C)$

// Disallow router queries after corruptions
if $\mathcal{Z} \neq \emptyset$ **then return** $\lightning$
if $b = 0$
  $C_{h+1} \leftarrow\!\!\$\ \mathcal{C}$
  // Update view of the corrupted-state simulator
  $\mathtt{C}_h \xleftarrow{\frown} (C, AD, C_{h+1})$
else
  $C_{h+1} \leftarrow \mathsf{OR\text{-}Dec}^{AD}_{\langle\varrho_h\rangle}(C)$
if $C_{h+1} \in \mathcal{M}$
  $corr_h \leftarrow \mathsf{true}$
else
  $corr_h \leftarrow \mathsf{false}$
// Determine syncness
if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      // Query not the top of the queue
      $sync_h \leftarrow \mathsf{false}$
    if $h < d \wedge C_{h+1} \in \mathcal{M}$
      // Correctness drop causes sync loss
      $sync_h \leftarrow \mathsf{false}$
  else
    // Querying router with empty queue
    $sync_h \leftarrow \mathsf{false}$
if $sync_h = \mathsf{true}$
  if $h = d$
    // Always suppress in-sync destination
    $corr_h \leftarrow \mathsf{true}$
    return $\lightning$
  else
    // Queue in-sync non-destination
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  return C_{h+1}

Fig. 4. The experiment $\mathsf{Exp}^{\mathsf{OAEF}_\ell\text{-}b}_{\mathsf{OE\text{-}Fw}}(\mathbb{A})$ to model authenticated encryption with limited forward-security for circuits of length ℓ of an onion encryption scheme OE-Fw in the forward direction. Colour coding explained in the main body.

5 The Security of CGO

Although CGO was introduced with a clear security goal in mind, neither of the existing security models for onion encryption [9,17] captured the desired functionality and security. Thus, lacking a formal security model, the security argument by Degabriele et al. [8] is incomplete: although it's shown formally that their instantiation of the updatable tweakable split-domain cipher is secure and informally that that security would entail CGO's forward security and tagging-resistance, the latter depends on the designer's intuition that the updatable tweakable split-domain cipher is used appropriately. For instance, that CGO refreshes keys, nonces, and tags at the right times and with sufficient care to avoid clashes.

With the introduction OAEF_ℓ security (Fig. 4) in the previous section, capturing what security entails for an onion-encryption scheme in the forward direction, precisely tailored to Tor's design desiderata as expressed in Proposal [13] and CGO's introduction [8], we can formally analyse CGO's security (in the forward-direction). We do so below (Theorem 1) and we conclude that CGO is indeed OAEF_ℓ secure if the underlying updatable tweakable split-domain cipher is CRND_ℓ secure (Fig. 1).

Theorem 1 (OAEF_ℓ security of CGO). *Let* EE *be an updatable split-domain tweakable cipher and let* CGO *be the construction given in Fig. 2. For any ℓ there exists a simple fully black box reduction $\mathbb{B}$ such that for all OAEF_ℓ adversaries $\mathbb{A}$ against* CGO

$$\mathsf{Adv}^{\mathsf{OAEF}_\ell}_{\mathsf{CGO}}(\mathbb{A}) \leq 2 \cdot \mathsf{Adv}^{\mathrm{CRND}_\ell}_{\mathsf{EE}}(\mathbb{B}) + (\ell \cdot q_{\mathsf{ENC}} + q_{\mathsf{DEC}})^2 2^{-n} \ ,$$

where q_{ENC} is the total number of $\mathbb{A}$'s ENC *queries,* DEC *is the total number of $\mathbb{A}$'s out-of-sync* DEC*-queries, and $\mathbb{B}$'s overhead is essentially linear in the number of queries by $\mathbb{A}$.*

Proof. Our proof proceeds through a sequence of game hops, divided into three stages. The initial game G_0 corresponds to the 'real', $b = 1$ case of the original game OAEF_ℓ instantiated with CGO. The first stage (G_0–G_4) is dedicated to reducing the number of keyed evaluations of the split domain cipher (EE, ED, and EU) and puts in place administrative overhead, including some bad events, to facilitate the second stage, namely substitution of the remaining invocations of EE, ED, and EU by randomly sampled bitstrings in G_5 instead. This crucial hop from real primitives to uniform, independent random bitstrings will be justified by a CRND_ℓ-reduction; a related CRND_ℓ-reduction ensures we can analyse all previously introduced bad events in G_5 (as opposed to their games of origin). Finally, in the third stage (G_5–G_8) we "undo" some of the administrative overhead introduced in the first stage and slowly move to a just-in-time sampling of randomness, thus eventually bringing us to the 'ideal', $b = 0$ case of the original game OAEF_ℓ.

We will detail the games and the hops below, where we defer the exact analysis of the bad events to two lemmas following this main proof. For the games, we only provide condensed code-snippets that concentrate on the changes from the previous games (for the reader's convenience, corresponding full game code can be found in Section A). The bad events, all introduced in the first stage, are

1. bad_{tf}, capturing out-of-sync integrity breaks; if triggered, an adversary can instantly conclude it is in the real OAEF_ℓ game (thus a distinguishing event).
2. bad_{ld}, capturing in-sync integrity breaks (leaky pipe); here, we preventively catch its potential of occuring already at the ENC oracle, rather than waiting for the integrity break actually occurring at the DEC oracle. Although an integrity break at the DEC-oracle would instantly be a distinguishing event, catching the event early in the ENC-oracle (as we do) is somewhat generous to the adversary, as in some contrived cases, the bad event could get triggered without an adversary being able to use it to distinguish.
3. bad_{tc} capturing repeat tags, which would invalidate the CRND_ℓ-reduction; this event is an artefact of our choice to use split-domain ciphers and their security notion based on rugged permutations, where an adversary is not allowed to repeat the same left-domain value (tag) across queries (to the same key).

First, for game G_0 (see Fig. 12) we take the $b = 1$ case of the OAEF_ℓ game (from Fig. 4) and instantiate it with CGO (from Fig. 2) with some minor notational changes and delaying inlining the $\mathsf{OR\text{-}Dec}^{AD}_{\langle \varrho_h \rangle}(C)$ invocation in the DEC oracle to a later game. Furthermore, as a small cosmetic change indicated by the greyed out code, we omit the explicit checks that the adversary indeed makes all its ENC and DEC oracle calls prior to any COR call, does not repeat COR calls, and does not call ENC with an out-of-bound destination ($d > \ell$). As there is a trivial reduction to deal with adversaries violating those checks, we will simply assume proper adversarial behaviour and conclude that

$$\Pr\Big[\mathsf{Exp}^{\mathsf{OAEF}_\ell\text{-}1}_{\mathsf{CGO}}(\mathbb{A}) = 1\Big] = \Pr\big[G^{\mathbb{A}}_0 \Rightarrow 1\big] \ .$$

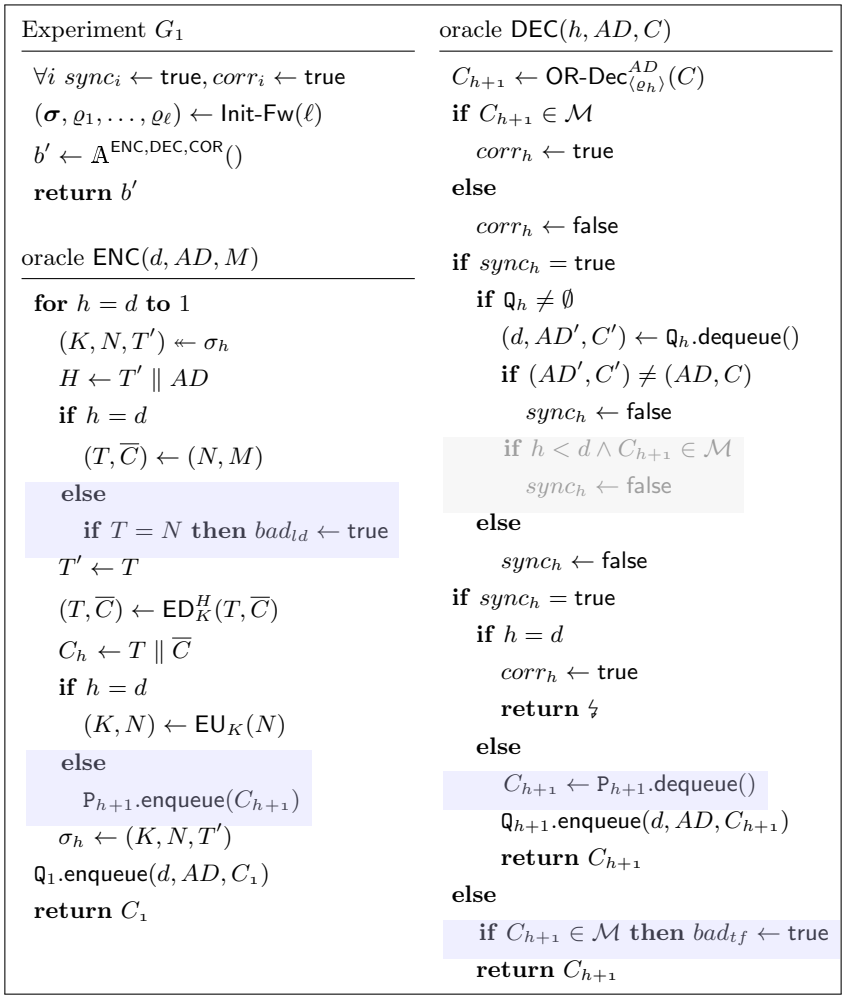

Fig. 5. G_1: Catching integrity breaks, and in-sync decryption errors early on (during encryption), allowing the introductiong of all in-sync ciphertext lists at the proxy. Grey code not to be executed. (Algorithm $\mathsf{Init\text{-}Fw}(\ell)$ and oracle $\mathsf{COR}(h)$ omitted as unchanged.)

In game G_1 (Figs. 5 and 13), we introduce our first two bad events. Firstly, bad_{tf} catches out-of-sync integrity breaks; the code is introduced in the DEC oracle, highlighted at the bottom. Secondly, we catch in-sync decryption errors early on using bad_{ld}. Rather than reacting to an intermediate in-sync ciphertext that triggers a sync loss at a router (the greyed out code in G_1's DEC code), we pre-empt by checking the trajectory of a ciphertext during creation at the proxy (the highlighted code in G_1's ENC code). Specifically, a proxy will "wrap" the onion ciphertext $C_d \to C_{d-1} \to \cdots \to C_1$ (assume $d > 1$ for the purpose of this explanation). When the first router processes C_1, if that router is still in-sync, then it is guaranteed to result in C_2 *unless* it recognizes the ciphertext, i.e. iff the T-part of C_2 equals the N-component of the state ϱ_1. During the proxy's preparation that equality can be checked in the $h = 1$ case of the for-loop: the state σ_1 from which N is parsed will match the state ϱ_1 that router 1 will use to process C_1 (again, assuming router 1 will still be in sync at that time); on the other hand, the value T will originate from the previous run through the loop (as $1 \neq d$) and thus equals the T part of C_2.

If bad_{ld} does not happen, we are guaranteed that in-sync ciphertexts do not inadvertently drop, making the greyed out code in the DEC oracle redundant. Furthermore, we can already store the correct ciphertext to be output by the DEC-oracle for these non-dropped in-sync ciphertext as they are prepared by the proxy. The lists P serve that purpose. We conclude that G_0 and G_1 are equivalent-until-bad, thus

$$\Pr\left[G_0^{\mathbb{A}} \Rightarrow 1\right] - \Pr\left[G_1^{\mathbb{A}} \Rightarrow 1\right] \leq \Pr\left[G_0^{\mathbb{A}} \,:\, bad\right];$$

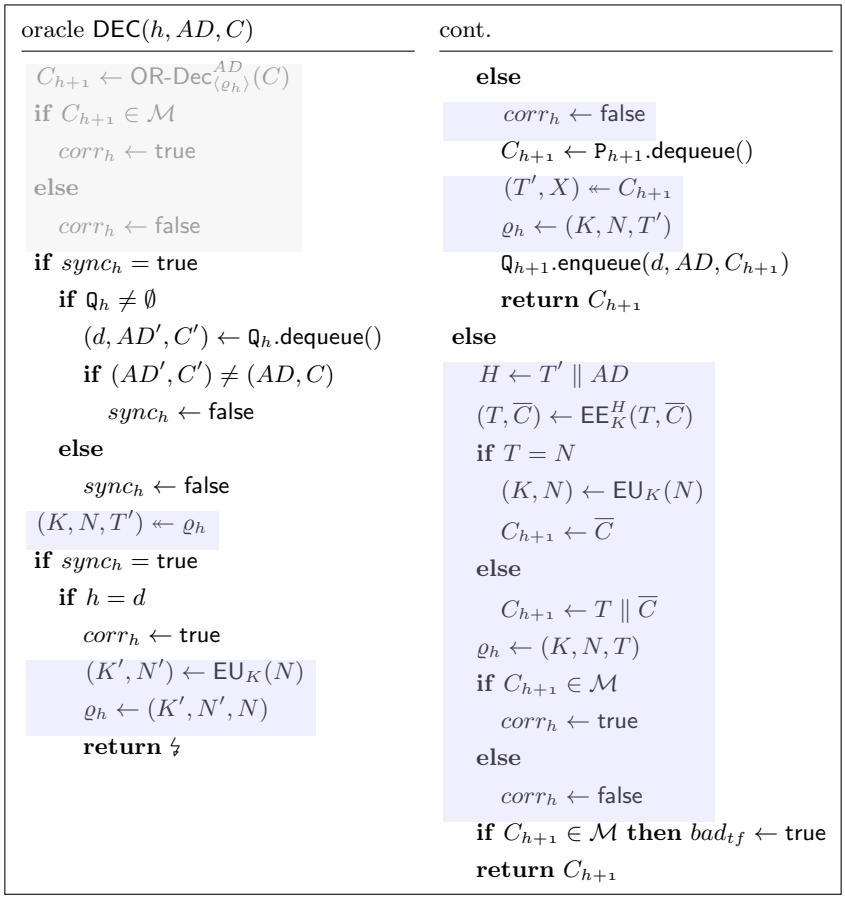

Fig. 6. G_2: Deferring and inlining the $\mathsf{OR\text{-}Dec}^{AD}_{\langle \varrho_h \rangle}(C)$ invocation only where it is needed.

where we use the convention that the generic *bad* captures all the possible bad flags present in the game at hand. Thus, for G_1 we have that $bad = bad_{ld} \vee bad_{tf}$, but for future games the number of included bad events will grow. We further use the convention that *as soon as a bad flag gets set, the game aborts.*

Observe that with the changes to G_1's DEC-oracle, especially the removal of the redundant code to catch inadvertent in-sync leaks, syncness can be determined purely based on DEC's input, thus the $\mathsf{OR\text{-}Dec}^{AD}_{\langle \varrho_h \rangle}(C)$ invocation plus setting of the $corr_h$ flag can be moved down. We do this in game G_2 (Figs. 6 and 14), where we instantiate said invocation using CGO's code, but with a twist:

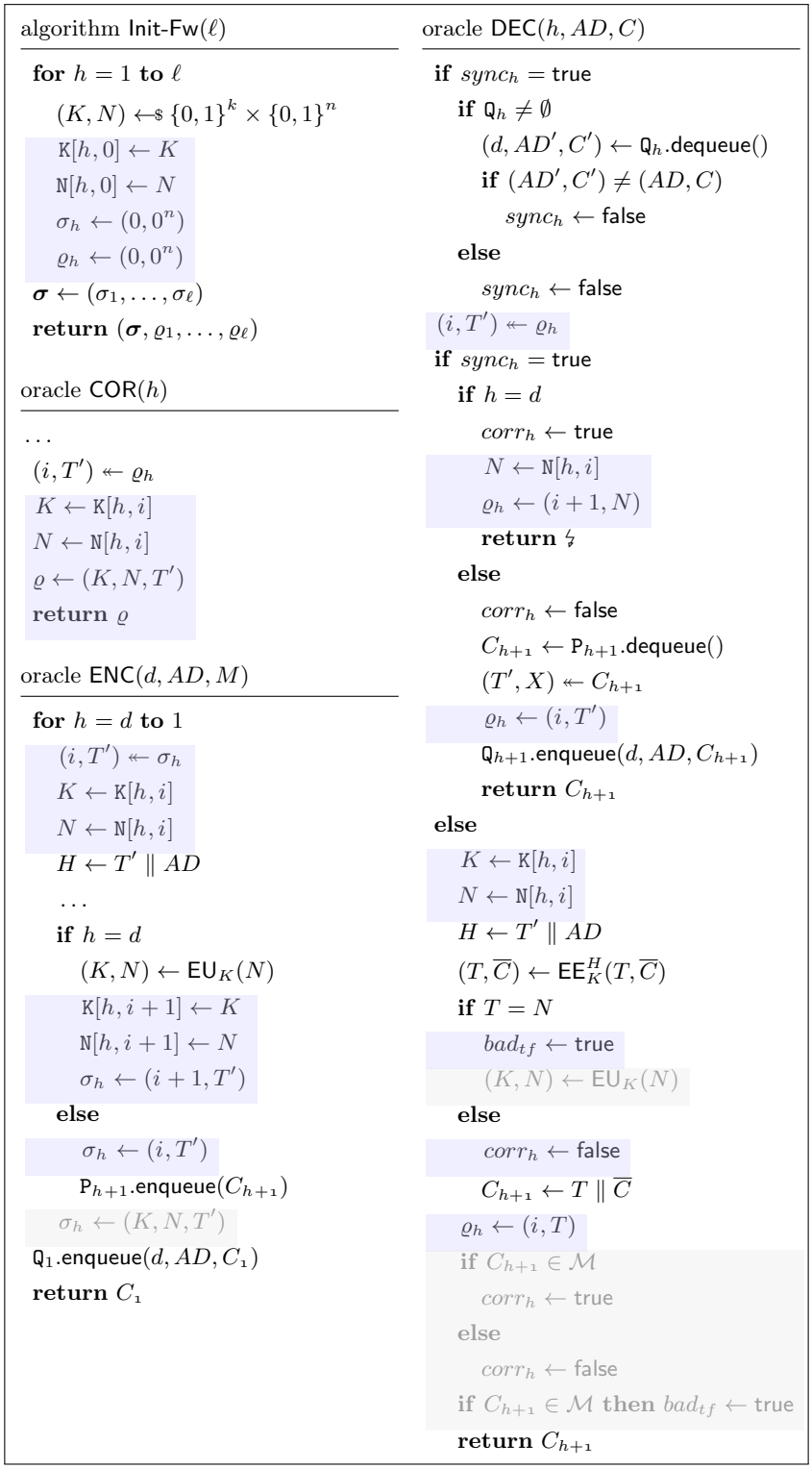

Fig. 7. G_3: Introducing the key-store and re-arranging code accordingly.

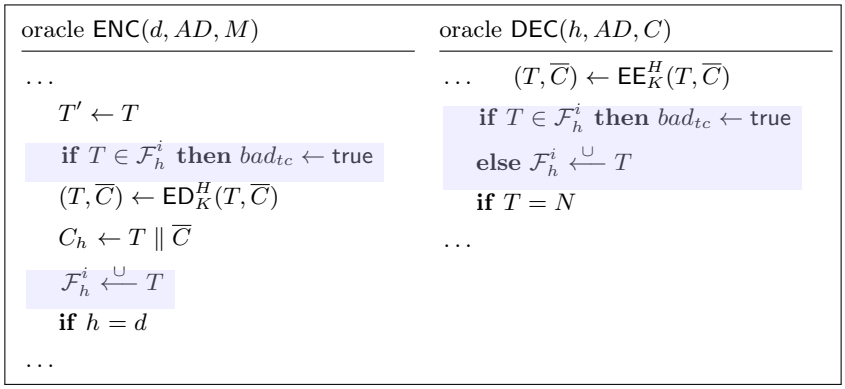

Fig. 8. G_4: Catching repeat use of left-values.

for in-sync ciphertexts we no longer need $\mathsf{OR\text{-}Dec}^{AD}_{\langle \varrho_h \rangle}(C)$'s output (as we either suppress it or retrieve it from $\mathtt{P}_{h+1}$) so only need to worry about updating ϱ_h appropriately. For out-of-sync ciphertexts we use the exact CGO code from Fig. 2 (just relabelling X to C_{h+1}). We conclude that G_1 and G_2 are equivalent, or

$$G_1 \equiv G_2 \,.$$

For out-of-sync ciphertexts, the DEC-oracle's code has three if-statements all based on whether the ciphertext is recognized. We declutter these in G_3 (Figs. 7 and 15) by merging the bodies of those three if-statements into a single conditional. As we set the bad_{tf} flag if the ciphertext is recognized, we the rest of that branch becomes irrelevant (due to abort triggered automatically by convention when setting a bad flag), allowing us to forgo the usual key update.

Yet, the main change introduced in game G_3 is the way state is represented: rather than keeping (K, N) explicitly as part of the states σ and ϱ, relevant values for K and N are kept in stores $\mathtt{K}$ and $\mathtt{N}$, respectively; the states σ and ϱ only contain a pointer to which elements in $\mathtt{K}$ and *noncestore* to address. For example, given a router handle h, the state σ_h contains a pointer i identifying which key to look up in $\mathtt{K}$, namely $\mathtt{K}[h, i]$. Crucially, both $\mathtt{K}$ and $\mathtt{N}$ are, after initialization, only updated during ENC oracle calls and never by the DEC oracle. Overall, games G_2 and G_3 are equivalent, thus

$$G_2 \equiv G_3 \,.$$

Game G_4 (Figs. 8 and 16) introduces an additional bad event bad_{tc} to catch repeat use of left-values of the split-domain cipher (as these would invalidate our subsequent reduction to G_5). By inspection, besides surfacing the bad_{tc} flag and some related administration in G_4, the two games are equivalent, so

$$\Pr\big[G_3^{\mathbb{A}} \Rightarrow 1\big] - \Pr\big[G_4^{\mathbb{A}} \Rightarrow 1\big] \leq \Pr\big[G_4^{\mathbb{A}} \,:\, bad\big] \,.$$

In game G_5 (Fig. 17), we replace all invocations of EE, ED, and EU with freshly sampled randomness. We argue that, conditioned on $\neg bad_{tc}$, the difference between G_4 and G_5 can be captured by a reduction to CRND. Indeed, let $\mathbb{A}$ be an OAEF_ℓ-adversary, then create a simple, fully black-box reduction $\mathbb{B}$ against CRND as follows: $\mathbb{B}$ creates the experiment G_4 while avoiding handling keys directly. Specifically, in the first stage of the OAEF_ℓ game (prior to any COR queries by $\mathbb{A}$), all calls to EE, ED, and EU are replaced with calls to its own oracles from the CRND-game, whereas in the second stage (to answer COR queries), $\mathbb{B}$ uses its own corruption oracle in the CRND-game. By careful inspection, irrespective of $\mathbb{A}$'s behaviour, the reduction $\mathbb{B}$ will never receive $\lightning$ from one of its oracles (rather, a bad flag will get set in the simulated G_4, allowing $\mathbb{B}$ to abort the simulation and output 0 to its own game). For the resulting reduction $\mathbb{B}$, we have that

$$\begin{aligned}
\Pr\big[G_4^{\mathbb{A}} \Rightarrow 1\big] - \Pr\big[G_5^{\mathbb{A}} \Rightarrow 1\big] &\leq \mathsf{Adv}^{\mathrm{CRND}_\ell}(\mathbb{B}) \,; \\
\Pr[G_4 \,:\, bad] - \Pr[G_5 \,:\, bad] &\leq \mathsf{Adv}^{\mathrm{CRND}_\ell}(\mathbb{B}) \,.
\end{aligned}$$

In game G_6 (Figs. 9 and 18) we start simplifying by removing code. Firstly, we are no longer interested in the bad events and their associated administration, so we can remove those lines. Secondly, as neither

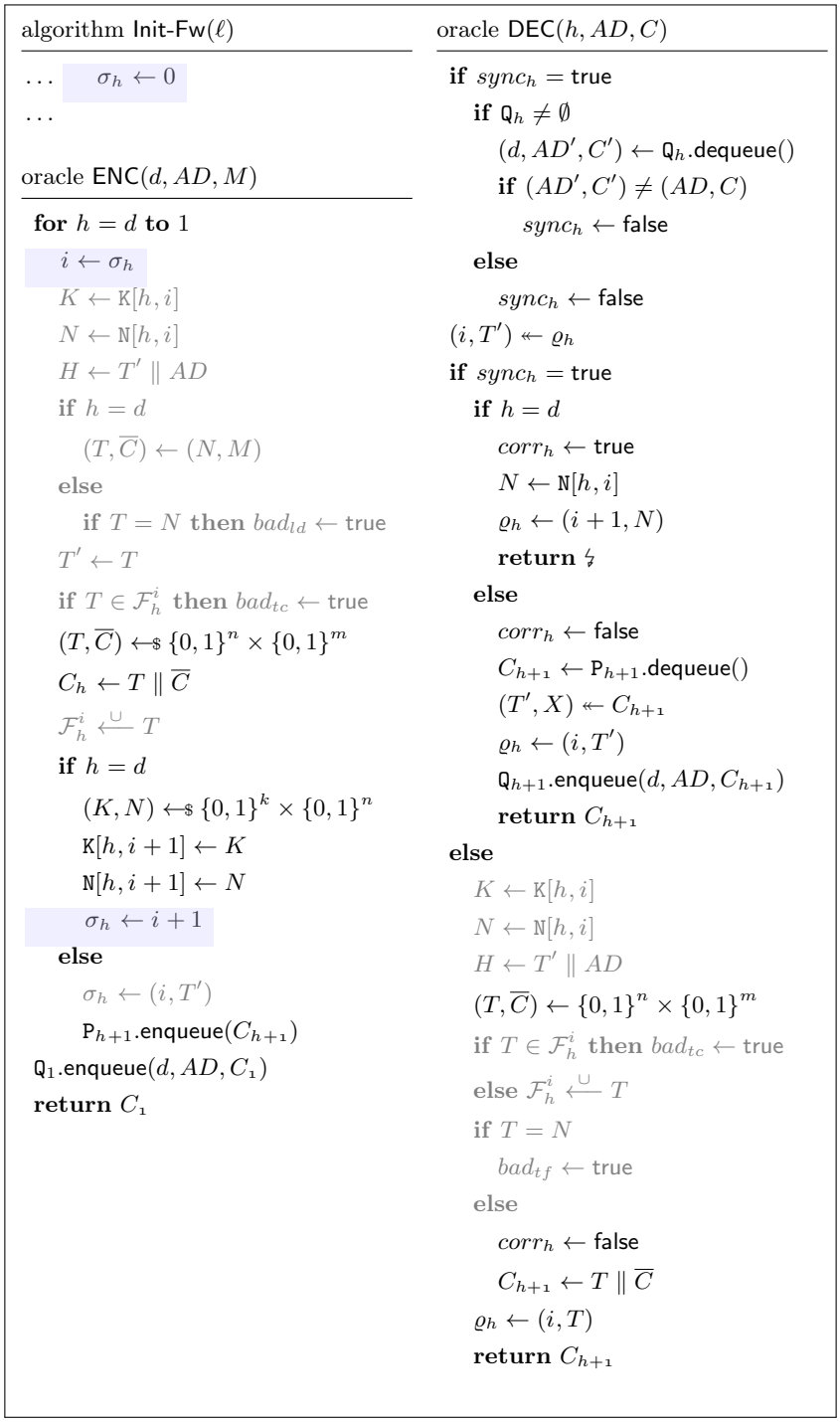

Fig. 9. G_6: Simplifications, where the greyed out code is no longer executed.

ENC nor DEC is still calling the split-domain cipher, they no longer need to read keys or calculate headers. Finally, as the encryption states σ_h never get exposed to the adversary, by inspection their T' component has become superfluous and can be pruned (turning σ_h into a simple counter). As games G_5 and G_6 are equivalent-until-bad, we have that

$$\Pr\left[G_5^{\mathbb{A}} \Rightarrow 1\right] - \Pr\left[G_6^{\mathbb{A}} \Rightarrow 1\right] \leq \Pr\left[G_5^{\mathbb{A}} \,:\, bad\right].$$

In game G_7 (Figs. 10 and 19), even more redundant game administration is cut away. As keys are no longer used or exposed in the game, except as part of a COR call, we no longer need to keep track

algorithm $\mathsf{Init\text{-}Fw}(\ell)$

for $h = 1$ **to** ℓ
  $N \leftarrow\!\!\$\ \{0,1\}^n$
  $\mathtt{K}[h,0] \leftarrow K$
  $\mathtt{N}[h,0] \leftarrow N$
  $\sigma_h \leftarrow 0$
  $\varrho_h \leftarrow 0$
$\boldsymbol{\sigma} \leftarrow (\sigma_1, \ldots, \sigma_\ell)$
return $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell)$

oracle $\mathsf{COR}(h)$

$\ldots$ $\quad \mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
$i \leftarrow \varrho_h$
if $i = 0$ **then** $T' \leftarrow 0^n$
else $T' \leftarrow \mathtt{N}[h, i-1]$
$K \leftarrow\!\!\$\ \{0,1\}^k$
$N \leftarrow \mathtt{N}[h,i]$
$\varrho \leftarrow (K, N, T')$
return ϱ

oracle $\mathsf{ENC}(d, AD, M)$

$i \leftarrow \sigma_d$
$(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
$C_d \leftarrow T \parallel \overline{C}$
$\mathtt{N}[d, i+1] \leftarrow\!\!\$\ \{0,1\}^n$
$\sigma_d \leftarrow i + 1$
for $h = d - 1$ **to** 1
  $i \leftarrow \sigma_h$
  $(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
  $C_h \leftarrow T \parallel \overline{C}$
  if $h = d$
    $(K, N) \leftarrow\!\!\$\ \{0,1\}^k \times \{0,1\}^n$
    $\mathtt{K}[h, i+1] \leftarrow K$
    $\mathtt{N}[h, i+1] \leftarrow N$
    $\sigma_h \leftarrow i + 1$
  else
    $\mathtt{P}_{h+1}.\mathsf{enqueue}(C_{h+1})$
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{DEC}(h, AD, C)$

$(T, \overline{C}) \leftarrow \{0,1\}^n \times \{0,1\}^m$
$corr_h \leftarrow \mathsf{false}$
$C_{h+1} \leftarrow T \parallel \overline{C}$
if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      $sync_h \leftarrow \mathsf{false}$
  else
    $sync_h \leftarrow \mathsf{false}$
$(i, T') \twoheadleftarrow \varrho_h$
if $sync_h = \mathsf{true}$
  if $h = d$
    $corr_h \leftarrow \mathsf{true}$
    $i \leftarrow \varrho_h$
    $N \leftarrow \mathtt{N}[h, i]$
    $\varrho_h \leftarrow i + 1$
    return $\lightning$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow \mathtt{P}_{h+1}.\mathsf{dequeue}()$
    $(T', X) \twoheadleftarrow C_{h+1}$
    $\varrho_h \leftarrow (i, T')$
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  $(T, \overline{C}) \leftarrow \{0,1\}^n \times \{0,1\}^m$
  $corr_h \leftarrow \mathsf{false}$
  $C_{h+1} \leftarrow T \parallel \overline{C}$
  $\varrho_h \leftarrow (i, T)$
  return C_{h+1}

Fig. 10. G_7: Further simplifications and reorderings.

of keys, so we dispense of the keystore $\mathtt{K}$ and instead sample a key afresh when it gets corrupted. In the ENC oracle, we take the initial, $h = d$ case outside of the for-loop, allowing us to simplify the body of said loop. In the DEC oracle, we move the sampling of $(T, \overline{C})$ (back) to the top and observe that the tag T is no longer needed as part of the state ϱ_h. Instead, we can determine the correct value of ϱ_h's T in case it gets corrupted, by exploiting that if a router is corruptable, its tag can be read from $\mathtt{N}$ (or still equals 0^n). By detailed inspection, the new game is still equivalent to the previous one, so

Experiment G_8

$\forall i\ sync_i \leftarrow \mathsf{true}, corr_i \leftarrow \mathsf{true}$
$(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
$b' \leftarrow \mathbb{A}^{\mathsf{ENC},\mathsf{DEC},\mathsf{COR}}()$
return b'

oracle $\mathsf{COR}(h)$

$\ldots \quad \mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
$i \leftarrow \varrho_h$
if $\mathtt{C}_h = \emptyset$ **then** $T' \leftarrow 0^n$
else $T' \leftarrow\!\!\$\ \{0,1\}^n$
$K \leftarrow\!\!\$\ \{0,1\}^k$
$N \leftarrow \{0,1\}^n$
$\varrho \leftarrow (K, N, T')$
return ϱ

oracle $\mathsf{ENC}(d, AD, M)$

$i \leftarrow \sigma_d$
$(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
$C_d \leftarrow T \parallel \overline{C}$
$\mathtt{N}[d, i+1] \leftarrow\!\!\$\ \{0,1\}^n$
$\sigma_d \leftarrow i + 1$
for $h = d - 1$ **to** 1
  $(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
  $C_h \leftarrow T \parallel \overline{C}$
  $\mathtt{P}_{h+1}.\mathsf{enqueue}(C_{h+1})$
$(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
$C_1 \leftarrow T \parallel \overline{C}$
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{DEC}(h, AD, C)$

$(T, \overline{C}) \leftarrow \{0,1\}^n \times \{0,1\}^m$
$corr_h \leftarrow \mathsf{false}$
$C_{h+1} \leftarrow T \parallel \overline{C}$
$\mathtt{C}_h \xleftarrow{\frown} (C, AD, C_{h+1})$
if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      $sync_h \leftarrow \mathsf{false}$
  else
    $sync_h \leftarrow \mathsf{false}$
if $sync_h = \mathsf{true}$
  if $h = d$
    $corr_h \leftarrow \mathsf{true}$
    $i \leftarrow \varrho_h$
    $\varrho_h \leftarrow i + 1$
    return $\lightning$
  else
    $C_{h+1} \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  return C_{h+1}

Fig. 11. G_8: The final simplifications, sampling just-in-time and removing the need for initialization and state altogether.

$$G_6 \equiv G_7\,.$$

As our last step, in game G_8 (Figs. 11 and 20) we remove the reliance on states σ and ϱ entirely: their only remaining use was determining upon corruption of a router whether that router had already been used or not. By keeping track of processed ciphertext at a router using the list $\mathtt{C}$ (see the DEC oracle), we can check whether $\mathtt{C}_h$ is empty or not instead when needing to simulate a router h upon successful corruption. The other change is to sample randomness only where and when it is needed. Thus ENC oracle only samples the ciphertext C_1 intended for the first router, with all remaining ciphertexts only sampled during their respective DEC-query. Similarly, nonces no longer need sampling during the first stage of the game and only get sampled during the COR query. Games G_7 and G_8 result in the exact same environment for an adversary, i.e., they are equivalent:

$$G_7 \equiv G_8\,.$$

Finally, G_8 can be seen to match the ideal scenario for the OAEF_ℓ-experiment, simply by identifying the appropriate lines of code in COR and designating them the simulator. Thus,

$$\Pr\left[\mathsf{Exp}^{\mathsf{OAEF}_\ell\text{-}0}_{\mathsf{CGO}}(\mathbb{A}) = 1\right] = \Pr\left[G^{\mathbb{A}}_8 \Rightarrow 1\right]\,.$$

Combining all steps together, we have achieved that

$$\Pr\Big[\mathsf{Exp}^{\mathsf{OAEF}_\ell\text{-}0}_{\mathsf{CGO}}(\mathbb{A})=1\Big]-\Pr\Big[\mathsf{Exp}^{\mathsf{OAEF}_\ell\text{-}1}_{\mathsf{CGO}}(\mathbb{A})=1\Big]\leq 2\mathsf{Adv}^{\mathrm{CRND}_\ell}(\mathbb{B})+2\Pr\big[G_5^{\mathbb{A}}\ :\ bad\big]\ ,$$

where all that remains is bounding the bad event in G_5, which is relatively straightforward. There are only two oracles that affect setting the bad events, namely ENC and DEC. In DEC, a bad flag gets set if a freshly sampled $T \leftarrow\!\!\$\ \{0,1\}^n$ is in the set $\mathcal{F}_h^i$ or equals N. That event happens with probability at most $(|\mathcal{F}_h^i|+1)2^{-n}$. To bound the cardinality $|\mathcal{F}_h^i|$, observe that each invocation of DEC can add at most one element to it, whereas each invocation of ENC could add up to $d \leq \ell$. For ENC, setting a bad flag is similar though slightly more involved, as the bad flags appear to be checked just above the sampling of T in the code. However, unless $h = d$, the T at hand will have been freshly generated in the previous step of the loop, and the same bound as for DEC setting bad applies. For the $h = d$ case, the first flag cannot be set, but the check against $\mathcal{F}_h^i$ is still relevant. Here, we can check instead whether a freshly generated N is in $\mathcal{F}_h^i$ (during the key update) and note that anytime $\mathcal{F}_h^i$ were to be expanded with N, a bad flag is set anyway. Overall, the probability is upper bounded by

$$(\ell \cdot q_{\mathsf{ENC}} + q_{\mathsf{DEC}})^2 2^{-n}\ ,$$

where q_{ENC} is the total number of ENC queries and DEC is the total number of out-of-sync DEC-queries.

References

1. Andreeva, E., Bogdanov, A., Luykx, A., Mennink, B., Mouha, N., Yasuda, K.: How to securely release unverified plaintext in authenticated encryption. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part I. LNCS, vol. 8873, pp. 105–125. Springer, Berlin, Heidelberg (Dec 2014). https://doi.org/10.1007/978-3-662-45611-8_6
2. Barwell, G., Page, D., Stam, M.: Rogue decryption failures: Reconciling AE robustness notions. In: Groth, J. (ed.) 15th IMA International Conference on Cryptography and Coding. LNCS, vol. 9496, pp. 94–111. Springer, Cham (Dec 2015). https://doi.org/10.1007/978-3-319-27239-9_6
3. Bellare, M., Kohno, T., Namprempre, C.: Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In: Atluri, V. (ed.) ACM CCS 2002. pp. 1–11. ACM Press (Nov 2002). https://doi.org/10.1145/586110.586112
4. Bellare, M., Ng, R., Tackmann, B.: Nonces are noticed: AEAD revisited. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part I. LNCS, vol. 11692, pp. 235–265. Springer, Cham (Aug 2019). https://doi.org/10.1007/978-3-030-26948-7_9
5. Camenisch, J., Lysyanskaya, A.: A formal treatment of onion routing. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 169–187. Springer, Berlin, Heidelberg (Aug 2005). https://doi.org/10.1007/11535218_11
6. Canetti, R., Krawczyk, H.: Analysis of key-exchange protocols and their use for building secure channels. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 453–474. Springer, Berlin, Heidelberg (May 2001). https://doi.org/10.1007/3-540-44987-6_28
7. Degabriele, J.P., Karadžić, V.: Overloading the nonce: Rugged PRPs, nonce-set AEAD, and order-resilient channels. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part IV. LNCS, vol. 13510, pp. 264–295. Springer, Cham (Aug 2022). https://doi.org/10.1007/978-3-031-15985-5_10
8. Degabriele, J.P., Melloni, A., Münch, J.P., Stam, M.: Counter Galois onion (CGO) for tor: Fast non-malleable onion encryption. Cryptology ePrint Archive, Report 2025/583 (2025), https://eprint.iacr.org/2025/583
9. Degabriele, J.P., Stam, M.: Untagging Tor: A formal treatment of onion encryption. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 259–293. Springer, Cham (Apr / May 2018). https://doi.org/10.1007/978-3-319-78372-7_9
10. Dingledine, R., Mathewson, N., Syverson, P.F.: Tor: The second-generation onion router. In: USENIX Security Symposium. pp. 303–320. USENIX (2004)
11. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions (extended abstract). In: 25th FOCS. pp. 464–479. IEEE Computer Society Press (Oct 1984). https://doi.org/10.1109/SFCS.1984.715949
12. Luby, M., Rackoff, C.: How to construct pseudo-random permutations from pseudo-random functions (abstract). In: Williams, H.C. (ed.) CRYPTO'85. LNCS, vol. 218, p. 447. Springer, Berlin, Heidelberg (Aug 1986). https://doi.org/10.1007/3-540-39799-X_34
13. Mathewson, N.: Proposal 202: Two improved relay encryption protocols for Tor cells. https://github.com/torproject/torspec/blob/main/proposals/202-improved-relay-crypto.txt (June 2012)
14. Mathewson, N.: Proposal 261: AEZ for relay cryptography. https://github.com/torproject/torspec/blob/main/proposals/261-aez-crypto.txt (October 2015)
15. Namprempre, C.: Secure channels based on authenticated encryption schemes: A simple characterization. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 515–532. Springer, Berlin, Heidelberg (Dec 2002). https://doi.org/10.1007/3-540-36178-2_32
16. Rogaway, P.: Nonce-based symmetric encryption. In: Roy, B.K., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 348–359. Springer, Berlin, Heidelberg (Feb 2004). https://doi.org/10.1007/978-3-540-25937-4_22
17. Rogaway, P., Zhang, Y.: Onion-AE: Foundations of nested encryption. PoPETs **2018**(2), 85–104 (Apr 2018). https://doi.org/10.1515/popets-2018-0014
18. Rogaway, P., Zhang, Y.: Simplifying game-based definitions - indistinguishability up to correctness and its application to stateful AE. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 3–32. Springer, Cham (Aug 2018). https://doi.org/10.1007/978-3-319-96881-0_1
19. Tomer Ashur, Orr Dunkelman, A.L.: Using ADL for relay cryptography (solving the crypto-tagging-attack). https://github.com/torproject/torspec/blob/dc3683f929a747876cfb66fa1b91edff68dfc27d/proposals/295-relay-crypto-with-adl.txt (July 2019)

A Games used in CGO's Security Proof

Experiment G_0

$\forall i\ sync_i \leftarrow \mathsf{true}, corr_i \leftarrow \mathsf{true}$
$(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
$b' \leftarrow \mathbb{A}^{\mathsf{ENC},\mathsf{DEC},\mathsf{COR}}()$
return b'

algorithm $\mathsf{Init\text{-}Fw}(\ell)$

for $h = 1$ **to** ℓ
  $(K, N) \leftarrow\!\!\$\ \{0,1\}^k \times \{0,1\}^n$
  $\sigma_h \leftarrow (K, N, 0^n)$
  $\varrho_h \leftarrow (K, N, 0^n)$
$\boldsymbol{\sigma} \leftarrow (\sigma_1, \ldots, \sigma_\ell)$
return $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell)$

oracle $\mathsf{COR}(h)$

if $h \in \mathcal{Z}$ **then return** $\lightning$
$corr \leftarrow corr_h$
$i^* \leftarrow h + 1$
while $i^* \leq \ell \wedge i^* \in \mathcal{Z}$
  $i^* \leftarrow i^* + 1$
if $\exists (d, AD, C) \in \mathtt{Q}_h$ s.t. $d < i^*$
  $corr \leftarrow \mathsf{false}$
if $corr = \mathsf{false}$ **then return** $\lightning$
$\mathcal{Z} \xleftarrow{\cup} h$
if $i^* \leq \ell$
  while $\mathtt{Q}_h \neq \emptyset$
    $(d, AD, C_i) \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    $\mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
return ϱ_h

oracle $\mathsf{ENC}(d, AD, M)$

if $\mathcal{Z} \neq \emptyset$ **then return** $\lightning$
if $(d > \ell)$ **then return** $\perp$
for $h = d$ **to** 1
  $(K, N, T') \twoheadleftarrow \sigma_h$
  $H \leftarrow T' \parallel AD$
  if $h = d$
    $(T, \overline{C}) \leftarrow (N, M)$
  $T' \leftarrow T$
  $(T, \overline{C}) \leftarrow \mathsf{ED}_K^H(T, \overline{C})$
  $C_h \leftarrow T \parallel \overline{C}$
  if $h = d$
    $(K, N) \leftarrow \mathsf{EU}_K(N)$
  $\sigma_h \leftarrow (K, N, T')$
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{DEC}(h, AD, C)$

if $\mathcal{Z} \neq \emptyset$ **then return** $\lightning$
$C_{h+1} \leftarrow \mathsf{OR\text{-}Dec}^{AD}_{\langle \varrho_h \rangle}(C)$
if $C_{h+1} \in \mathcal{M}$
  $corr_h \leftarrow \mathsf{true}$
else
  $corr_h \leftarrow \mathsf{false}$
if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      $sync_h \leftarrow \mathsf{false}$
    if $h < d \wedge C_{h+1} \in \mathcal{M}$
      $sync_h \leftarrow \mathsf{false}$
  else
    $sync_h \leftarrow \mathsf{false}$
if $sync_h = \mathsf{true}$
  if $h = d$
    $corr_h \leftarrow \mathsf{true}$
    return $\lightning$
  else
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  return C_{h+1}

Fig. 12. G_0: Essentially, the game $\mathsf{Exp}^{\mathsf{OAEF}_{\ell-1}}_{\mathsf{CGO}}(\mathbb{A})$, corresponding to the real setting instantiated with CGO, whose state initialization and encryption routine have been inlined.

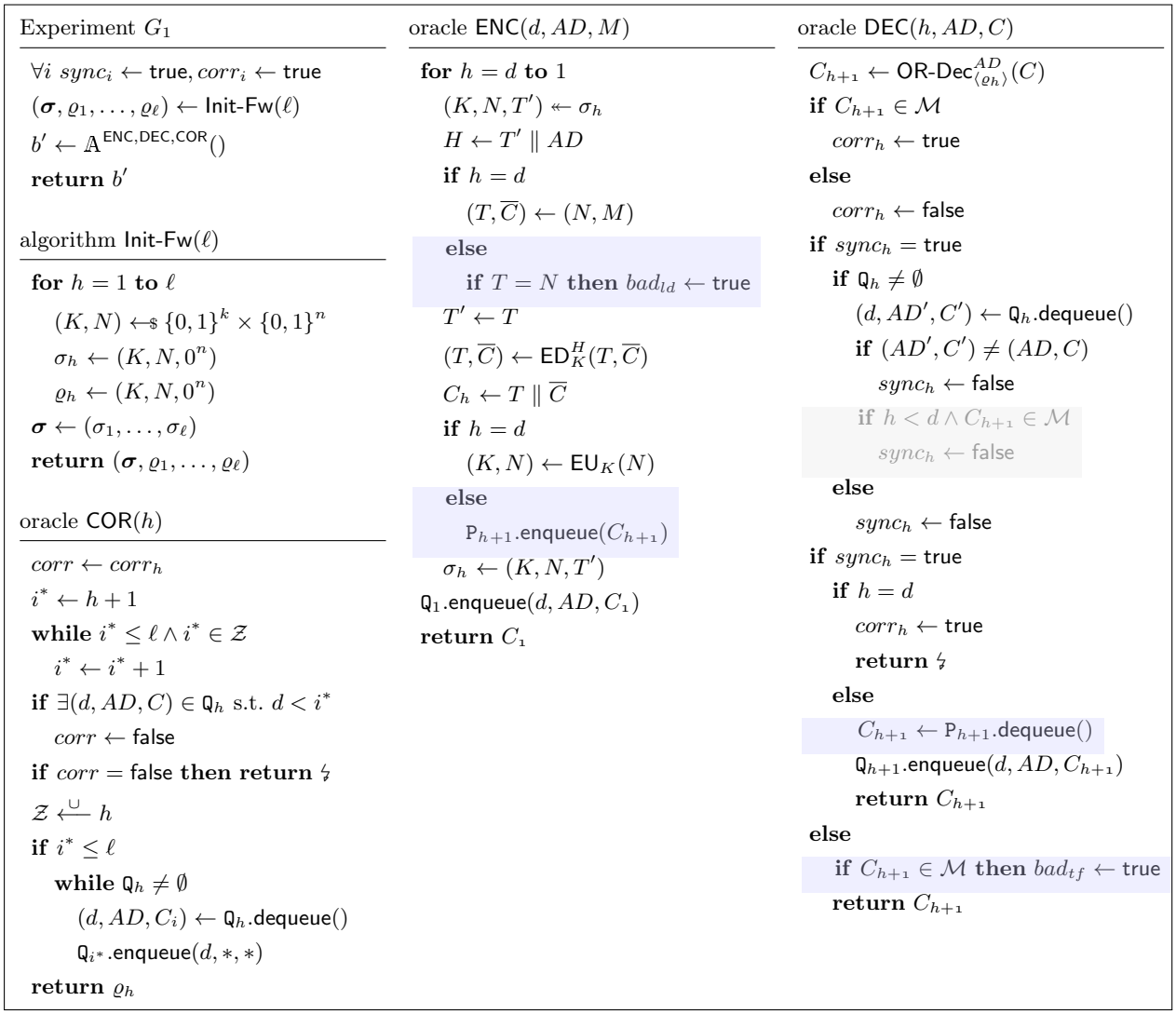

Fig. 13. G_1: Catching integrity breaks, and in-sync decryption errors early on (during encryption), allowing the introductiong of all in-sync ciphertext lists at the proxy. Grey code not to be executed.

Experiment G_2

$\forall i\ sync_i \leftarrow \mathsf{true}, corr_i \leftarrow \mathsf{true}$
$(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
$b' \leftarrow \mathbb{A}^{\mathsf{ENC},\mathsf{DEC},\mathsf{COR}}()$
return b'

algorithm $\mathsf{Init\text{-}Fw}(\ell)$

for $h = 1$ **to** ℓ
  $(K, N) \leftarrow\!\!\$\ \{0,1\}^k \times \{0,1\}^n$
  $\sigma_h \leftarrow (K, N, 0^n)$
  $\varrho_h \leftarrow (K, N, 0^n)$
$\boldsymbol{\sigma} \leftarrow (\sigma_1, \ldots, \sigma_\ell)$
return $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell)$

oracle $\mathsf{COR}(h)$

$corr \leftarrow corr_h$
$i^* \leftarrow h + 1$
while $i^* \leq \ell \wedge i^* \in \mathcal{Z}$
  $i^* \leftarrow i^* + 1$
if $\exists (d, AD, C) \in \mathtt{Q}_h$ s.t. $d < i^*$
  $corr \leftarrow \mathsf{false}$
if $corr = \mathsf{false}$ **then return** $\lightning$
$\mathcal{Z} \xleftarrow{\cup} h$
if $i^* \leq \ell$
  while $\mathtt{Q}_h \neq \emptyset$
    $(d, AD, C_i) \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    $\mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
return ϱ_h

oracle $\mathsf{ENC}(d, AD, M)$

for $h = d$ **to** 1
  $(K, N, T') \twoheadleftarrow \sigma_h$
  $H \leftarrow T' \parallel AD$
  if $h = d$
    $(T, \overline{C}) \leftarrow (N, M)$
  else
    if $T = N$ **then** $bad_{ld} \leftarrow \mathsf{true}$
  $T' \leftarrow T$
  $(T, \overline{C}) \leftarrow \mathsf{ED}_K^H(T, \overline{C})$
  $C_h \leftarrow T \parallel \overline{C}$
  if $h = d$
    $(K, N) \leftarrow \mathsf{EU}_K(N)$
  else
    $\mathtt{P}_{h+1}.\mathsf{enqueue}(C_{h+1})$
  $\sigma_h \leftarrow (K, N, T')$
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{DEC}(h, AD, C)$

$C_{h+1} \leftarrow \mathsf{OR\text{-}Dec}^{AD}_{\langle \varrho_h \rangle}(C)$
if $C_{h+1} \in \mathcal{M}$
  $corr_h \leftarrow \mathsf{true}$
else
  $corr_h \leftarrow \mathsf{false}$
if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      $sync_h \leftarrow \mathsf{false}$
  else
    $sync_h \leftarrow \mathsf{false}$
$(K, N, T') \twoheadleftarrow \varrho_h$
if $sync_h = \mathsf{true}$
  if $h = d$
    $corr_h \leftarrow \mathsf{true}$
    $(K', N') \leftarrow \mathsf{EU}_K(N)$
    $\varrho_h \leftarrow (K', N', N)$
    return $\lightning$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow \mathtt{P}_{h+1}.\mathsf{dequeue}()$
    $(T', X) \twoheadleftarrow C_{h+1}$
    $\varrho_h \leftarrow (K, N, T')$
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  $H \leftarrow T' \parallel AD$
  $(T, \overline{C}) \leftarrow \mathsf{EE}_K^H(T, \overline{C})$
  if $T = N$
    $(K, N) \leftarrow \mathsf{EU}_K(N)$
    $C_{h+1} \leftarrow \overline{C}$
  else
    $C_{h+1} \leftarrow T \parallel \overline{C}$
  $\varrho_h \leftarrow (K, N, T)$
  if $C_{h+1} \in \mathcal{M}$
    $corr_h \leftarrow \mathsf{true}$
  else
    $corr_h \leftarrow \mathsf{false}$
  if $C_{h+1} \in \mathcal{M}$ **then** $bad_{tf} \leftarrow \mathsf{true}$
  return C_{h+1}

Fig. 14. G_2: Deferring and inlining the $\mathsf{OR\text{-}Dec}^{AD}_{\langle \varrho_h \rangle}(C)$ invocation only where it is needed.

Experiment G_3

$\forall i\ sync_i \leftarrow \mathsf{true}, corr_i \leftarrow \mathsf{true}$
$(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
$b' \leftarrow \mathbb{A}^{\mathsf{ENC},\mathsf{DEC},\mathsf{COR}}()$
return b'

algorithm $\mathsf{Init\text{-}Fw}(\ell)$

for $h = 1$ **to** ℓ
  $(K, N) \leftarrow\!\!\$\ \{0,1\}^k \times \{0,1\}^n$
  $\mathtt{K}[h, 0] \leftarrow K$
  $\mathtt{N}[h, 0] \leftarrow N$
  $\sigma_h \leftarrow (0, 0^n)$
  $\varrho_h \leftarrow (0, 0^n)$
$\boldsymbol{\sigma} \leftarrow (\sigma_1, \ldots, \sigma_\ell)$
return $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell)$

oracle $\mathsf{COR}(h)$

$corr \leftarrow corr_h$
$i^* \leftarrow h + 1$
while $i^* \leq \ell \wedge i^* \in \mathcal{Z}$
  $i^* \leftarrow i^* + 1$
if $\exists (d, AD, C) \in \mathtt{Q}_h$ s.t. $d < i^*$
  $corr \leftarrow \mathsf{false}$
if $corr = \mathsf{false}$ **then return** $\lightning$
$\mathcal{Z} \xleftarrow{\cup} h$
if $i^* \leq \ell$
  while $\mathtt{Q}_h \neq \emptyset$
    $(d, AD, C_i) \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    $\mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
$(i, T') \twoheadleftarrow \varrho_h$
$K \leftarrow \mathtt{K}[h, i]$
$N \leftarrow \mathtt{N}[h, i]$
$\varrho \leftarrow (K, N, T')$
return ϱ

oracle $\mathsf{ENC}(d, AD, M)$

for $h = d$ **to** 1
  $(i, T') \twoheadleftarrow \sigma_h$
  $K \leftarrow \mathtt{K}[h, i]$
  $N \leftarrow \mathtt{N}[h, i]$
  $H \leftarrow T' \parallel AD$
  if $h = d$
    $(T, \overline{C}) \leftarrow (N, M)$
  else
    if $T = N$ **then** $bad_{ld} \leftarrow \mathsf{true}$
  $T' \leftarrow T$
  $(T, \overline{C}) \leftarrow \mathsf{ED}_K^H(T, \overline{C})$
  $C_h \leftarrow T \parallel \overline{C}$
  if $h = d$
    $(K, N) \leftarrow \mathsf{EU}_K(N)$
    $\mathtt{K}[h, i+1] \leftarrow K$
    $\mathtt{N}[h, i+1] \leftarrow N$
    $\sigma_h \leftarrow (i+1, T')$
  else
    $\sigma_h \leftarrow (i, T')$
    $\mathtt{P}_{h+1}.\mathsf{enqueue}(C_{h+1})$
  $\sigma_h \leftarrow (K, N, T')$
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{DEC}(h, AD, C)$

if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      $sync_h \leftarrow \mathsf{false}$
  else
    $sync_h \leftarrow \mathsf{false}$
$(i, T') \twoheadleftarrow \varrho_h$
if $sync_h = \mathsf{true}$
  if $h = d$
    $corr_h \leftarrow \mathsf{true}$
    $N \leftarrow \mathtt{N}[h, i]$
    $\varrho_h \leftarrow (i+1, N)$
    return $\lightning$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow \mathtt{P}_{h+1}.\mathsf{dequeue}()$
    $(T', X) \twoheadleftarrow C_{h+1}$
    $\varrho_h \leftarrow (i, T')$
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  $K \leftarrow \mathtt{K}[h, i]$
  $N \leftarrow \mathtt{N}[h, i]$
  $H \leftarrow T' \parallel AD$
  $(T, \overline{C}) \leftarrow \mathsf{EE}_K^H(T, \overline{C})$
  if $T = N$
    $bad_{tf} \leftarrow \mathsf{true}$
    $(K, N) \leftarrow \mathsf{EU}_K(N)$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow T \parallel \overline{C}$
  $\varrho_h \leftarrow (i, T)$
  if $C_{h+1} \in \mathcal{M}$
    $corr_h \leftarrow \mathsf{true}$
  else
    $corr_h \leftarrow \mathsf{false}$
  if $C_{h+1} \in \mathcal{M}$ **then** $bad_{tf} \leftarrow \mathsf{true}$
  return C_{h+1}

Fig. 15. G_3: Introducing the key-store and re-arranging code accordingly.

Experiment G_4

$\forall i\ sync_i \leftarrow \mathsf{true}, corr_i \leftarrow \mathsf{true}$
$(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
$b' \leftarrow \mathbb{A}^{\mathsf{ENC},\mathsf{DEC},\mathsf{COR}}()$
return b'

algorithm $\mathsf{Init\text{-}Fw}(\ell)$

for $h = 1$ **to** ℓ
  $(K, N) \leftarrow\!\!\$\ \{0,1\}^k \times \{0,1\}^n$
  $\mathtt{K}[h, 0] \leftarrow K$
  $\mathtt{N}[h, 0] \leftarrow N$
  $\sigma_h \leftarrow (0, 0^n)$
  $\varrho_h \leftarrow (0, 0^n)$
$\boldsymbol{\sigma} \leftarrow (\sigma_1, \ldots, \sigma_\ell)$
return $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell)$

oracle $\mathsf{COR}(h)$

$corr \leftarrow corr_h$
$i^* \leftarrow h + 1$
while $i^* \leq \ell \wedge i^* \in \mathcal{Z}$
  $i^* \leftarrow i^* + 1$
if $\exists (d, AD, C) \in \mathtt{Q}_h$ s.t. $d < i^*$
  $corr \leftarrow \mathsf{false}$
if $corr = \mathsf{false}$ **then return** $\lightning$
$\mathcal{Z} \xleftarrow{\cup} h$
if $i^* \leq \ell$
  while $\mathtt{Q}_h \neq \emptyset$
    $(d, AD, C_i) \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    $\mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
$(i, T') \twoheadleftarrow \varrho_h$
$K \leftarrow \mathtt{K}[h, i]$
$N \leftarrow \mathtt{N}[h, i]$
$\varrho \leftarrow (K, N, T')$
return ϱ

oracle $\mathsf{ENC}(d, AD, M)$

for $h = d$ **to** 1
  $(i, T') \twoheadleftarrow \sigma_h$
  $K \leftarrow \mathtt{K}[h, i]$
  $N \leftarrow \mathtt{N}[h, i]$
  $H \leftarrow T' \parallel AD$
  if $h = d$
    $(T, \overline{C}) \leftarrow (N, M)$
  else
    if $T = N$ **then** $bad_{ld} \leftarrow \mathsf{true}$
  $T' \leftarrow T$
  if $T \in \mathcal{F}_h^i$ **then** $bad_{tc} \leftarrow \mathsf{true}$
  $(T, \overline{C}) \leftarrow \mathsf{ED}_K^H(T, \overline{C})$
  $C_h \leftarrow T \parallel \overline{C}$
  $\mathcal{F}_h^i \xleftarrow{\cup} T$
  if $h = d$
    $(K, N) \leftarrow \mathsf{EU}_K(N)$
    $\mathtt{K}[h, i+1] \leftarrow K$
    $\mathtt{N}[h, i+1] \leftarrow N$
    $\sigma_h \leftarrow (i+1, T')$
  else
    $\sigma_h \leftarrow (i, T')$
    $\mathtt{P}_{h+1}.\mathsf{enqueue}(C_{h+1})$
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{DEC}(h, AD, C)$

if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      $sync_h \leftarrow \mathsf{false}$
  else
    $sync_h \leftarrow \mathsf{false}$
$(i, T') \twoheadleftarrow \varrho_h$
if $sync_h = \mathsf{true}$
  if $h = d$
    $corr_h \leftarrow \mathsf{true}$
    $N \leftarrow \mathtt{N}[h, i]$
    $\varrho_h \leftarrow (i+1, N)$
    return $\lightning$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow \mathtt{P}_{h+1}.\mathsf{dequeue}()$
    $(T', X) \twoheadleftarrow C_{h+1}$
    $\varrho_h \leftarrow (i, T')$
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  $K \leftarrow \mathtt{K}[h, i]$
  $N \leftarrow \mathtt{N}[h, i]$
  $H \leftarrow T' \parallel AD$
  $(T, \overline{C}) \leftarrow \mathsf{EE}_K^H(T, \overline{C})$
  if $T \in \mathcal{F}_h^i$ **then** $bad_{tc} \leftarrow \mathsf{true}$
  else $\mathcal{F}_h^i \xleftarrow{\cup} T$
  if $T = N$
    $bad_{tf} \leftarrow \mathsf{true}$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow T \parallel \overline{C}$
  $\varrho_h \leftarrow (i, T)$
  return C_{h+1}

Fig. 16. G_4: Catching repeat use of left-values.

Experiment G_5

$\forall i\ sync_i \leftarrow \mathsf{true}, corr_i \leftarrow \mathsf{true}$
$(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
$b' \leftarrow \mathbb{A}^{\mathsf{ENC},\mathsf{DEC},\mathsf{COR}}()$
return b'

algorithm $\mathsf{Init\text{-}Fw}(\ell)$

for $h = 1$ **to** ℓ
  $(K, N) \leftarrow\!\!\$\ \{0,1\}^k \times \{0,1\}^n$
  $\mathtt{K}[h, 0] \leftarrow K$
  $\mathtt{N}[h, 0] \leftarrow N$
  $\sigma_h \leftarrow (0, 0^n)$
  $\varrho_h \leftarrow (0, 0^n)$
$\boldsymbol{\sigma} \leftarrow (\sigma_1, \ldots, \sigma_\ell)$
return $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell)$

oracle $\mathsf{COR}(h)$

$corr \leftarrow corr_h$
$i^* \leftarrow h + 1$
while $i^* \leq \ell \wedge i^* \in \mathcal{Z}$
  $i^* \leftarrow i^* + 1$
if $\exists (d, AD, C) \in \mathtt{Q}_h$ s.t. $d < i^*$
  $corr \leftarrow \mathsf{false}$
if $corr = \mathsf{false}$ **then return** $\lightning$
$\mathcal{Z} \xleftarrow{\cup} h$
if $i^* \leq \ell$
  while $\mathtt{Q}_h \neq \emptyset$
    $(d, AD, C_i) \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    $\mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
$(i, T') \twoheadleftarrow \varrho_h$
$K \leftarrow \mathtt{K}[h, i]$
$N \leftarrow \mathtt{N}[h, i]$
$\varrho \leftarrow (K, N, T')$
return ϱ

oracle $\mathsf{ENC}(d, AD, M)$

for $h = d$ **to** 1
  $(i, T') \twoheadleftarrow \sigma_h$
  $K \leftarrow \mathtt{K}[h, i]$
  $N \leftarrow \mathtt{N}[h, i]$
  $H \leftarrow T' \parallel AD$
  if $h = d$
    $(T, \overline{C}) \leftarrow (N, M)$
  else
    if $T = N$ **then** $bad_{ld} \leftarrow \mathsf{true}$
  $T' \leftarrow T$
  if $T \in \mathcal{F}_h^i$ **then** $bad_{tc} \leftarrow \mathsf{true}$
  $(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
  $C_h \leftarrow T \parallel \overline{C}$
  $\mathcal{F}_h^i \xleftarrow{\cup} T$
  if $h = d$
    $(K, N) \leftarrow\!\!\$\ \{0,1\}^k \times \{0,1\}^n$
    $\mathtt{K}[h, i+1] \leftarrow K$
    $\mathtt{N}[h, i+1] \leftarrow N$
    $\sigma_h \leftarrow (i+1, T')$
  else
    $\sigma_h \leftarrow (i, T')$
    $\mathtt{P}_{h+1}.\mathsf{enqueue}(C_{h+1})$
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{DEC}(h, AD, C)$

if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      $sync_h \leftarrow \mathsf{false}$
  else
    $sync_h \leftarrow \mathsf{false}$
$(i, T') \twoheadleftarrow \varrho_h$
if $sync_h = \mathsf{true}$
  if $h = d$
    $corr_h \leftarrow \mathsf{true}$
    $N \leftarrow \mathtt{N}[h, i]$
    $\varrho_h \leftarrow (i+1, N)$
    return $\lightning$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow \mathtt{P}_{h+1}.\mathsf{dequeue}()$
    $(T', X) \twoheadleftarrow C_{h+1}$
    $\varrho_h \leftarrow (i, T')$
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  $K \leftarrow \mathtt{K}[h, i]$
  $N \leftarrow \mathtt{N}[h, i]$
  $H \leftarrow T' \parallel AD$
  $(T, \overline{C}) \leftarrow \{0,1\}^n \times \{0,1\}^m$
  if $T \in \mathcal{F}_h^i$ **then** $bad_{tc} \leftarrow \mathsf{true}$
  else $\mathcal{F}_h^i \xleftarrow{\cup} T$
  if $T = N$
    $bad_{tf} \leftarrow \mathsf{true}$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow T \parallel \overline{C}$
  $\varrho_h \leftarrow (i, T)$
  return C_{h+1}

Fig. 17. G_5: Substituting all remaining EE/ED/EU calls with freshly sampled random strings.

Experiment G_6

$\forall i\ sync_i \leftarrow \mathsf{true}, corr_i \leftarrow \mathsf{true}$
$(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
$b' \leftarrow \mathbb{A}^{\mathsf{ENC},\mathsf{DEC},\mathsf{COR}}()$
return b'

algorithm $\mathsf{Init\text{-}Fw}(\ell)$

for $h = 1$ **to** ℓ
  $(K, N) \leftarrow\!\!\$\ \{0,1\}^k \times \{0,1\}^n$
  $\mathtt{K}[h, 0] \leftarrow K$
  $\mathtt{N}[h, 0] \leftarrow N$
  $\sigma_h \leftarrow 0$
  $\varrho_h \leftarrow (0, 0^n)$
$\boldsymbol{\sigma} \leftarrow (\sigma_1, \ldots, \sigma_\ell)$
return $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell)$

oracle $\mathsf{COR}(h)$

$corr \leftarrow corr_h$
$i^* \leftarrow h + 1$
while $i^* \leq \ell \wedge i^* \in \mathcal{Z}$
  $i^* \leftarrow i^* + 1$
if $\exists (d, AD, C) \in \mathtt{Q}_h$ s.t. $d < i^*$
  $corr \leftarrow \mathsf{false}$
if $corr = \mathsf{false}$ **then return** $\lightning$
$\mathcal{Z} \xleftarrow{\cup} h$
if $i^* \leq \ell$
  while $\mathtt{Q}_h \neq \emptyset$
    $(d, AD, C_i) \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    $\mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
$(i, T') \twoheadleftarrow \varrho_h$
$K \leftarrow \mathtt{K}[h, i]$
$N \leftarrow \mathtt{N}[h, i]$
$\varrho \leftarrow (K, N, T')$
return ϱ

oracle $\mathsf{ENC}(d, AD, M)$

for $h = d$ **to** 1
  $i \leftarrow \sigma_h$
  $K \leftarrow \mathtt{K}[h, i]$
  $N \leftarrow \mathtt{N}[h, i]$
  $H \leftarrow T' \parallel AD$
  if $h = d$
    $(T, \overline{C}) \leftarrow (N, M)$
  else
    if $T = N$ **then** $bad_{ld} \leftarrow \mathsf{true}$
  $T' \leftarrow T$
  if $T \in \mathcal{F}_h^i$ **then** $bad_{tc} \leftarrow \mathsf{true}$
  $(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
  $C_h \leftarrow T \parallel \overline{C}$
  $\mathcal{F}_h^i \xleftarrow{\cup} T$
  if $h = d$
    $(K, N) \leftarrow\!\!\$\ \{0,1\}^k \times \{0,1\}^n$
    $\mathtt{K}[h, i+1] \leftarrow K$
    $\mathtt{N}[h, i+1] \leftarrow N$
    $\sigma_h \leftarrow i + 1$
  else
    $\sigma_h \leftarrow (i, T')$
    $\mathtt{P}_{h+1}.\mathsf{enqueue}(C_{h+1})$
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{DEC}(h, AD, C)$

if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      $sync_h \leftarrow \mathsf{false}$
  else
    $sync_h \leftarrow \mathsf{false}$
$(i, T') \twoheadleftarrow \varrho_h$
if $sync_h = \mathsf{true}$
  if $h = d$
    $corr_h \leftarrow \mathsf{true}$
    $N \leftarrow \mathtt{N}[h, i]$
    $\varrho_h \leftarrow (i + 1, N)$
    return $\lightning$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow \mathtt{P}_{h+1}.\mathsf{dequeue}()$
    $(T', X) \twoheadleftarrow C_{h+1}$
    $\varrho_h \leftarrow (i, T')$
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  $K \leftarrow \mathtt{K}[h, i]$
  $N \leftarrow \mathtt{N}[h, i]$
  $H \leftarrow T' \parallel AD$
  $(T, \overline{C}) \leftarrow \{0,1\}^n \times \{0,1\}^m$
  if $T \in \mathcal{F}_h^i$ **then** $bad_{tc} \leftarrow \mathsf{true}$
  else $\mathcal{F}_h^i \xleftarrow{\cup} T$
  if $T = N$
    $bad_{tf} \leftarrow \mathsf{true}$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow T \parallel \overline{C}$
  $\varrho_h \leftarrow (i, T)$
  return C_{h+1}

Fig. 18. G_6: Simplifications, where the greyed out code is no longer executed.

Experiment G_7

$\forall i\ sync_i \leftarrow \mathsf{true}, corr_i \leftarrow \mathsf{true}$
$(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
$b' \leftarrow \mathbb{A}^{\mathsf{ENC},\mathsf{DEC},\mathsf{COR}}()$
return b'

algorithm $\mathsf{Init\text{-}Fw}(\ell)$

for $h = 1$ **to** ℓ
  $N \leftarrow\!\!\$\ \{0,1\}^n$
  $\mathtt{K}[h, 0] \leftarrow K$
  $\mathtt{N}[h, 0] \leftarrow N$
  $\sigma_h \leftarrow 0$
  $\varrho_h \leftarrow 0$
$\boldsymbol{\sigma} \leftarrow (\sigma_1, \ldots, \sigma_\ell)$
return $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell)$

oracle $\mathsf{COR}(h)$

$corr \leftarrow corr_h$
$i^* \leftarrow h + 1$
while $i^* \leq \ell \wedge i^* \in \mathcal{Z}$
  $i^* \leftarrow i^* + 1$
if $\exists (d, AD, C) \in \mathtt{Q}_h$ s.t. $d < i^*$
  $corr \leftarrow \mathsf{false}$
if $corr = \mathsf{false}$ **then return** $\lightning$
$\mathcal{Z} \xleftarrow{\cup} h$
if $i^* \leq \ell$
  while $\mathtt{Q}_h \neq \emptyset$
    $(d, AD, C_i) \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    $\mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
$i \leftarrow \varrho_h$
if $i = 0$ **then** $T' \leftarrow 0^n$
else $T' \leftarrow \mathtt{N}[h, i-1]$
$K \leftarrow\!\!\$\ \{0,1\}^k$
$N \leftarrow \mathtt{N}[h, i]$
$\varrho \leftarrow (K, N, T')$
return ϱ

oracle $\mathsf{ENC}(d, AD, M)$

$i \leftarrow \sigma_d$
$(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
$C_d \leftarrow T \parallel \overline{C}$
$\mathtt{N}[d, i+1] \leftarrow\!\!\$\ \{0,1\}^n$
$\sigma_d \leftarrow i + 1$
for $h = d - 1$ **to** 1
  $i \leftarrow \sigma_h$
  $(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
  $C_h \leftarrow T \parallel \overline{C}$
  if $h = d$
    $(K, N) \leftarrow\!\!\$\ \{0,1\}^k \times \{0,1\}^n$
    $\mathtt{K}[h, i+1] \leftarrow K$
    $\mathtt{N}[h, i+1] \leftarrow N$
    $\sigma_h \leftarrow i + 1$
  else
    $\mathtt{P}_{h+1}.\mathsf{enqueue}(C_{h+1})$
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{DEC}(h, AD, C)$

$(T, \overline{C}) \leftarrow \{0,1\}^n \times \{0,1\}^m$
$corr_h \leftarrow \mathsf{false}$
$C_{h+1} \leftarrow T \parallel \overline{C}$
if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      $sync_h \leftarrow \mathsf{false}$
  else
    $sync_h \leftarrow \mathsf{false}$
$(i, T') \twoheadleftarrow \varrho_h$
if $sync_h = \mathsf{true}$
  if $h = d$
    $corr_h \leftarrow \mathsf{true}$
    $i \leftarrow \varrho_h$
    $N \leftarrow \mathtt{N}[h, i]$
    $\varrho_h \leftarrow i + 1$
    return $\lightning$
  else
    $corr_h \leftarrow \mathsf{false}$
    $C_{h+1} \leftarrow \mathtt{P}_{h+1}.\mathsf{dequeue}()$
    $(T', X) \twoheadleftarrow C_{h+1}$
    $\varrho_h \leftarrow (i, T')$
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  $(T, \overline{C}) \leftarrow \{0,1\}^n \times \{0,1\}^m$
  $corr_h \leftarrow \mathsf{false}$
  $C_{h+1} \leftarrow T \parallel \overline{C}$
  $\varrho_h \leftarrow (i, T)$
  return C_{h+1}

Fig. 19. G_7: Further simplifications and reorderings.

Experiment G_8

$\forall i\ sync_i \leftarrow \mathsf{true}, corr_i \leftarrow \mathsf{true}$
$(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell) \leftarrow \mathsf{Init\text{-}Fw}(\ell)$
$b' \leftarrow \mathbb{A}^{\mathsf{ENC},\mathsf{DEC},\mathsf{COR}}()$
return b'

algorithm $\mathsf{Init\text{-}Fw}(\ell)$

for $h = 1$ **to** ℓ
  $N \leftarrow\!\!\$\ \{0,1\}^n$
  $\mathtt{N}[h, 0] \leftarrow N$
  $\sigma_h \leftarrow 0$
  $\varrho_h \leftarrow 0$
$\boldsymbol{\sigma} \leftarrow (\sigma_1, \ldots, \sigma_\ell)$
return $(\boldsymbol{\sigma}, \varrho_1, \ldots, \varrho_\ell)$

oracle $\mathsf{COR}(h)$

$corr \leftarrow corr_h$
$i^* \leftarrow h + 1$
while $i^* \leq \ell \wedge i^* \in \mathcal{Z}$
  $i^* \leftarrow i^* + 1$
if $\exists (d, AD, C) \in \mathtt{Q}_h$ s.t. $d < i^*$
  $corr \leftarrow \mathsf{false}$
if $corr = \mathsf{false}$ **then return** $\lightning$
$\mathcal{Z} \xleftarrow{\cup} h$
if $i^* \leq \ell$
  while $\mathtt{Q}_h \neq \emptyset$
    $(d, AD, C_i) \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    $\mathtt{Q}_{i^*}.\mathsf{enqueue}(d, *, *)$
$i \leftarrow \varrho_h$
if $\mathtt{C}_h = \emptyset$ **then** $T' \leftarrow 0^n$
else $T' \leftarrow\!\!\$\ \{0,1\}^n$
$K \leftarrow\!\!\$\ \{0,1\}^k$
$N \leftarrow \{0,1\}^n$
$\varrho \leftarrow (K, N, T')$
return ϱ

oracle $\mathsf{ENC}(d, AD, M)$

$i \leftarrow \sigma_d$
$(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
$C_d \leftarrow T \parallel \overline{C}$
$\mathtt{N}[d, i+1] \leftarrow\!\!\$\ \{0,1\}^n$
$\sigma_d \leftarrow i + 1$
for $h = d - 1$ **to** 1
  $(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
  $C_h \leftarrow T \parallel \overline{C}$
  $\mathtt{P}_{h+1}.\mathsf{enqueue}(C_{h+1})$
$(T, \overline{C}) \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
$C_1 \leftarrow T \parallel \overline{C}$
$\mathtt{Q}_1.\mathsf{enqueue}(d, AD, C_1)$
return C_1

oracle $\mathsf{DEC}(h, AD, C)$

$(T, \overline{C}) \leftarrow \{0,1\}^n \times \{0,1\}^m$
$corr_h \leftarrow \mathsf{false}$
$C_{h+1} \leftarrow T \parallel \overline{C}$
$\mathtt{C}_h \xleftarrow{\frown} (C, AD, C_{h+1})$
if $sync_h = \mathsf{true}$
  if $\mathtt{Q}_h \neq \emptyset$
    $(d, AD', C') \leftarrow \mathtt{Q}_h.\mathsf{dequeue}()$
    if $(AD', C') \neq (AD, C)$
      $sync_h \leftarrow \mathsf{false}$
  else
    $sync_h \leftarrow \mathsf{false}$
if $sync_h = \mathsf{true}$
  if $h = d$
    $corr_h \leftarrow \mathsf{true}$
    $i \leftarrow \varrho_h$
    $\varrho_h \leftarrow i + 1$
    return $\lightning$
  else
    $C_{h+1} \leftarrow\!\!\$\ \{0,1\}^n \times \{0,1\}^m$
    $\mathtt{Q}_{h+1}.\mathsf{enqueue}(d, AD, C_{h+1})$
    return C_{h+1}
else
  return C_{h+1}

Fig. 20. G_8: The final simplifications, sampling just-in-time and removing the need for initialization and state altogether.