Upgrading Security Properties for Updatable Public-Key Encryption through Modular Transformations

by

Pranshu Kumar

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Pranshu Kumar 2026

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Updatable public-key encryption (UPKE) serves as a key tool for ensuring compromise resilience in dynamic group messaging systems such as MLS and TreeKEM. Prior work has introduced a variety of security notions for UPKE, including chosen-randomness and chosen-update security, chosen ciphertext attack (CCA) security, adaptive corruption resilience, public verifiability, and adversarially generated forking update structures. However, these notions make the security definition complex, and there is very little work on instantiating schemes that satisfy it.

In this work, we develop generic transformations that enable a sequence of upgrades from weaker to stronger UPKE notions, each introducing additional security properties. These security properties include straightline and forking update structures, selective and adaptive corruptions, public verifiability, chosen randomness and chosen-update security, and CPA and CCA notions. We define intermediate security notions that serve as useful stepping stones while adding the aforementioned properties. Some transformations rely only on standard tools such as random oracles, while others additionally exploit natural structural properties of schemes such as update uniformity and ciphertext updatability, which we identify and formalize in this work. This modular structure allows one to isolate and focus on specific combinations of properties of interest, facilitating easier security analysis.

Acknowledgements

First and foremost, I would like to express my sincere thanks to my supervisors, Douglas Stebila and Mohammad Hajiabadi, for their incredible support and guidance. This project would not have been possible without your insights and the countless discussions we have had. I hope to continue learning from you both in the next few years.

Thank you to my committee members, Ian Goldberg and Sam Jaques, for taking the time to read the thesis and providing helpful feedback. I would also like to thank Roman Langrehr for the many hours of discussions we have had and for always being patient with me.

I want to thank the members of the CrySP lab for creating such a friendly, collaborative, and fun environment. The past two years have been wonderful, and I am glad to have made so many great friends and memories here. I would like to specifically thank Thomas, Lucas, John, Shane, Karaneh, Vecna, and Ross for all the time we have spent together, both in and out of the lab.

Thank you, Megha, for always being there and believing in me.

Finally, I would like to thank my parents and my sister for all that they have done for me and for their support.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

List of Figures vii

1 Introduction 1

1.1 Our Contributions 3

1.2 Organization 5

2 Preliminaries 6

2.1 Updatable PKEs 7

2.2 Zero Knowledge Proofs 9

3 Straightline Security 11

3.1 S-IND-CR-CPA: Straightline Security, Chosen Randomness 11

3.2 OW-CR-CPA: One-Wayness, Chosen Randomness 13

3.3 S-IND-CU-CPA: Straightline Security, Chosen Updates 14

3.4 Lifting Chosen Randomness To Chosen Update Security 15

4 Adding Forking And Public Verifiability **23**

4.1 HF-IND-CU-CPA-PV: Half-Forking, Chosen Updates 25

4.2 Lifting Straightline to Half-Forking Security 25

4.3 F-IND-CU-CPA-PV Security 35

4.4 Lifting Half-Forking to Forking Security 35

5 Chosen Ciphertext Security **45**

5.1 CCA-Secure UKEM 45

5.2 Lifting CPA UPKE To CCA UKEM 48

6 Corruptions **54**

6.1 Selective Corruptions 56

6.2 An Intermediate Step: F-IND-CU-CCA-PV2 Security ("Reordered Forking") 58

6.3 Lifting CCA UKEM To Reordered Forking Security 58

6.4 Lifting Reordered Forking Secure UKEM To Security Against Selective Corruptions 72

7 Applicability **80**

8 Conclusions and Future Work **83**

References **84**

List of Figures

1.1 Summary of our transformations. 4

3.1 The S-IND-CR-CPA game for a UPKE. 12
3.2 The OW-CR-CPA game for a UPKE. 13
3.3 The S-IND-CU-CPA game for a UPKE. 14
3.4 Theorem 1 transformation. 15
3.5 Theorem 1 game 0. 18
3.6 Theorem 1 game 1. 19
3.7 Theorem 1 game 2. 20
3.8 Theorem 1 game 3. 21
3.9 Theorem 1 reduction. 22

4.1 The HF-IND-CU-CPA-PV game for a UPKE. 26
4.2 Theorem 2 transformation. 27
4.3 Theorem 2 game 0. 29
4.4 Theorem 2 game 1. 30
4.5 Theorem 2 game 2. 31
4.6 Theorem 2 game 3 and game 4. 32
4.7 Theorem 2 reduction. 34
4.8 The F-IND-CU-CPA-PV game for a UPKE. 36
4.9 Theorem 3 game 0. 39

4.10 Theorem 3 game 1. 40
4.11 Theorem 3 game 2. 41
4.12 Theorem 3 game 5. 42
4.13 Theorem 3 reduction. 43

5.1 The F-IND-CU-CCA-PV game for a UKEM. 47
5.2 Theorem 4 transformation. 48
5.3 Theorem 4 game 0. 50
5.4 Theorem 4 game 1. 51
5.5 Theorem 4 game 2. 52
5.6 Theorem 4 reduction. 53

6.1 The F-IND-CU-CCA-PV game with adaptive corruptions for an UKEM. . 55
6.2 The Sel-F-IND-CU-CCA-PV game for an UKEM. 57
6.3 The F-IND-CU-CCA-PV2 game for a UKEM. 59
6.4 Theorem 5 game 0. 62
6.5 Theorem 5 game 1. 63
6.6 Theorem 5 game 2. 63
6.7 Theorem 5 game 3. 64
6.8 Theorem 5 game 4. 65
6.9 Theorem 5 game 6 and game 7. 66
6.10 Theorem 5 game 8. 68
6.11 Theorem 5 game 9. 68
6.12 Theorem 5 game 10. 69
6.13 Theorem 5 game 11. 71
6.14 Theorem 5 game 12. 72
6.15 Theorem 6 game 0. 74
6.16 Theorem 6 game 1. 75

6.17 Theorem 6 game 2 and game 3. 76
6.18 Theorem 6 reduction. 77
6.19 Theorem 6 reduction contd. 78

7.1 The combined transformation. 81

Chapter 1

Introduction

Modern secure messaging applications have cryptographic needs beyond simple confidentiality, integrity, and authentication. Secure messaging sessions are often long-lived, increasing the likelihood that secret keys will be compromised. To protect against such leaks, these applications often incorporate properties such as forward secrecy (FS) and post-compromise security (PCS). Forward secrecy promises that even if the long-term secret key leaks, past messages remain confidential. Similarly, post-compromise security ensures that there is a mechanism to return to a secure state after a compromise, so that future messages remain secure. These properties are most commonly achieved by frequently updating the secret key material used to encrypt messages, and it is imperative that both parties update to the same key, even if one of them is offline. The Messaging Layer Security (MLS) [BBR+23] protocol is a standard for secure group messaging that incorporates many security properties, including FS and PCS for groups. An important building block for achieving these properties in secure group messaging is continuous group key agreement (CGKA) [ACDT20, AAN+22, ACJM20] which helps the group agree on and update secret key values. In MLS, the TreeKEM component uses updatable public-key encryption schemes (UPKEs) to implement the CGKA process.

UPKEs [JMM19, EJKM22, HLP22, AW25, DKW21, HPS23] allow one user to issue updates to other users so they can roll their secret key material forward. More specifically, UPKEs augment standard public key encryption (PKE) with two algorithms, UpdPk and UpdSk, that facilitate key updates. These two algorithms enable any sender to initiate a key update by updating the public key and generating an update token, which the receiver uses to update their private key accordingly, achieving a form of forward secrecy. The security definition for UPKEs builds on the IND-CPA game of standard PKEs and aims to capture forward secrecy by giving the adversary access to two additional oracles: one

for adversarial updates to the key and the other for a non-adversarial update followed by a leak of the secret key material. We call the latter an honest-update-and-reveal query. Chosen randomness security for UPKEs captures the notion of security when the adversary is allowed to tell a party to update their secret key with a well-formed update token, then reveal the secret key after an honest update; the adversary should not be able to learn anything about messages encrypted under earlier keys. Chosen-update security strengthens chosen-randomness security by capturing the notion that the adversary may submit malformed update tokens, somewhat analogous to chosen-ciphertext attacks on PKEs, where the adversary can submit malformed ciphertexts.

MLS uses UPKEs in TreeKEM, and requires that they are secure against chosen-update and chosen-ciphertext attacks. However, Alwen, Fuchsbauer, and Mularczyk [AFM24] showed that there are scenarios in which an adversary can still attack UPKEs in TreeKEM and proposed stronger security notions with additional properties for UPKEs. In TreeKEM, if synchronization fails, the adversary can cause different users to update the same key material in different ways. From the adversary's perspective, the key may no longer be a single chain of updates, but a tree of updates; *forking* security definitions for UPKEs allow the adversary to construct such trees. Since TreeKEM is designed to be asynchronous, it might be the case that some user updates the public key for a receiver while that receiver is offline. In this case, other users would like to know whether the update token and the new public key are a valid update, since an adversary could have instead sampled a fresh public key and submitted a bogus token; this property is called *public verifiability*. At the same time, with forking enabling a tree of keys, it does not make sense to restrict the adversary to a single honest-update-and-reveal query; so the security definition can be extended to allow arbitrary honest updates, as well as *corruption* of multiple nodes in the tree, provided that such reveals do not make the game trivial to win.

The UPKEs used in the current MLS standard do not include forking, public verifiability, or corruption in their security definitions; likely because it is hard to construct efficient schemes that satisfy such complex security requirements. To our knowledge, only two schemes in the literature satisfy these security properties. When [AFM24] proposed these properties, they instantiated them with a scheme based on the discrete-log assumption in groups. Later, [AFMR25] modified the lattice-based UPKE of [HPS23] to add these properties, but it remains unclear how generalizable these modifications are. There is a need for more constructions of strong UPKEs to understand the security and efficiency characteristics of schemes instantiated under this definition, which will help choose a suitable candidate to strengthen TreeKEM and prevent potential attacks.

1.1 Our Contributions

In this work, we provide generic transformations that lift UPKEs to these stronger security definitions. Starting with a straightline, chosen-randomness, CPA-secure UPKE, we introduce security properties like chosen-updates, forking, public verifiability (PV), CCA-security and corruptions one-by-one to come up with many intermediate security definitions, and provide transforms for achieving these definitions. To facilitate these transforms, we identify two useful properties for UPKEs: ciphertext updatability and update uniformity, which, alongside the random oracle model, help us prove that our transforms satisfy these security definitions. Our first transform uses Asano and Watanabe's [AW25] FO-like derandomization and re-encryption in the update function to lift a straightline IND-CR-CPA-secure UPKE to a straightline IND-CU-CPA-secure one. Then, using ciphertext updatability, along with a straight-line simulation extractable NIZK-PoK, we can lift it to an intermediate "half-forking" IND-CU-CPA UPKE with PV. The key idea of the reduction for this transform is using ciphertext updatability to update the challenge ciphertext encrypted under a given key to a key that the adversary expects. After that, we lift the scheme to full forking security, while maintaining IND-CU-CPA and PV by using update uniformity. Update uniformity implies that it is hard for an adversary to know if a keypair resulted from a direct update on a parent node A or another node B elsewhere in a given update tree. This property, along with the ZK property of the NIZK-PoK and one-wayness of the UPKE allows our reduction to forge an honest update carried out by the challenger. So if an adversary asks for an honest-update-and-reveal query on a node A, we can ask the challenger for the same query on a different node B and pass the result to the adversary as if it was performed on node A.

Next we use a modified version of the Fujisaki–Okamoto (FO) transform [HHK17] commonly used in modern post-quantum KEMs like ML-KEM (Kyber) to add *chosen ciphertext* security while maintaining forking, chosen-update security with PV. Before adding selective corruption, we introduce another intermediate definition called *reordered* forking IND-CU-CCA security with PV, which allows the adversary to ask for the honest-update-and-reveal before asking for the challenge ciphertext; this intermediate step makes our final transform much easier. We lift our UPKE to this definition using ciphertext updatability to once again update the challenge ciphertext in the reduction. Finally, we add selective corruptions to our security requirements. In this setting, the adversary must first commit to the tree structure of the updates it intends to perform, along with the indices of the nodes in the challenge set. The reduction can make use of this selective information (the challenge set), along with update uniformity, to forge honest updates between nodes inside the challenge set to those outside it. This forgery is the same as the ones we used

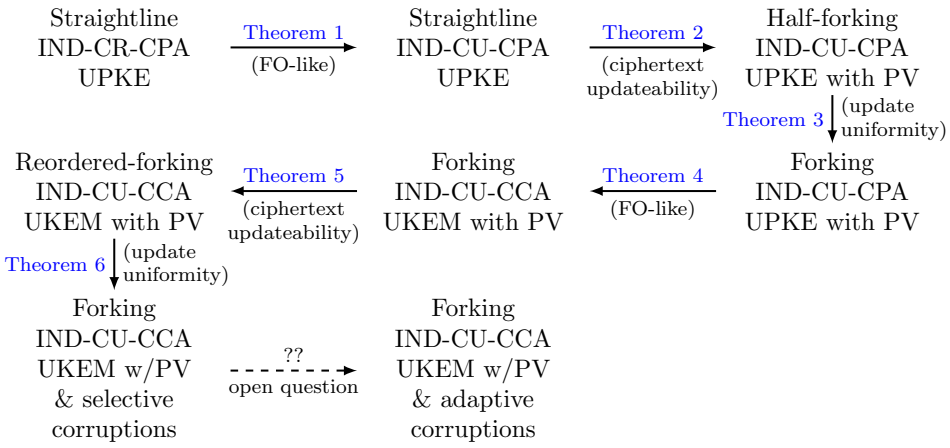


Figure 1.1: Summary of our transformations.

to lift half-forking to full-forking security, just repeated many times. Taken together, our transforms allow any straightline IND-CR-CPA UPKE with ciphertext updates and update uniformity to be lifted to a forking IND-CU-CCA secure UPKE with public verifiability and selective corruptions. A summary can be found in Figure 1.1.

Here is a summary of all the terms discussed so far:

- **Straightline security**: The security setting where the adversary is only allowed to update the latest node, resulting in a graph of updates that looks like a straight line chain.
- **Forking security**: The security setting where the adversary is allowed to update any node that exists, resulting in a graph of updates that looks like a tree.
- **Chosen randomness security**: The security setting where the adversary provides the randomness used to update a node honestly. This property is related to the security of the update mechanism.
- **Chosen update security**: The security setting where the adversary may provide malformed update tokens to update a node. This property is related to the security of the update mechanism.
- **Public verifiability**: The security property which allows other members of the group (non secret key holders) to verify that an update was generated properly.

- **Corruptions**: The security property that allows the adversary to ask for the secret keys of arbitrary nodes, as long as they do not allow it to win trivially. It is a generalization of the honest-update-and-reveal query in the forking setting.
- **Selective corruptions**: The security setting where the adversary has to commit some information about the nodes it is going to corrupt at the start of the game.
- **Adaptive corruptions**: The security setting where the adversary can decide which nodes to corrupt on the fly without having committed to anything in the past.
- **Chosen plaintext security**: The security setting where the adversary is given access to a challenge oracle but no decryption oracle for ciphertexts. This property is related to the security of the ciphertexts.
- **Chosen ciphertext security**: The security setting where the adversary is given access to a decryption oracle alongside the challenge oracle. The adversary cannot ask for decryption of the challenge ciphertext under the corresponding challenge key to prevent trivial wins. This property is related to the security of the ciphertexts.

1.2 Organization

The remainder of this work is organized as follows. In Chapter 2, we describe our notation, terminology, and review concepts that are useful to understand our work. Chapter 3 covers straightline security definitions and the transform that lifts a chosen-randomness-secure UPKE to a chosen-update-secure one in the straightline setting. In Chapter 4, we add forking security and public verifiability to our definitions and present transformations achieving them. Chapter 5 upgrades chosen plaintext attack (CPA) security to chosen ciphertext attack (CCA) security, and Chapter 6 covers transformations to add security against corruptions. In Chapter 7 we discuss the utility of our transforms and how generally applicable they are before finally closing with some limitations and discussion of future work in Chapter 8.

Chapter 2

Preliminaries

Notation. We use λ for the security parameter. For a distribution $\mathcal{D}$, we write $x \in \mathcal{D}$ (e.g., $(\mathsf{pk}_0, \mathsf{sk}_0) \in \mathsf{KGen}(1^\lambda)$) to mean x is in the support of $\mathcal{D}$, and use $x \leftarrow\!\!\$\ \mathcal{D}$ to mean x is sampled from $\mathcal{D}$. For a set S, $x \leftarrow\!\!\$\ S$ means x is uniformly sampled from S.

Namespaces. Throughout the paper, we talk about many schemes with much overlapping syntax. We use namespaces with schemes and their functions, such as $\mathsf{CR}.\Pi$ or $\mathsf{CU}.\mathsf{UpdPk}$, to disambiguate the scheme or function in question.

Stacks and Queues. In our reductions, we will need to make use of some basic data structures. We define the syntax for those here.

The first structure is called a *stack*. It is a last-in-first-out (LIFO) data structure, meaning that the items added to the stack are removed in the reverse order of their insertion. We write $A \leftarrow \mathit{Stack}()$ to initialize a stack in the variable A. We add an item x to the stack using $A.\mathit{push}(x)$. Finally, $A.\mathit{pop}()$ removes and returns the most recent item added to the stack. We also use the functions $A.\mathit{is_empty}()$ and $A.\mathit{is_not_empty}()$ to check whether the stack is empty or contains items, respectively.

Similar to a stack, the *queue* is a data structure. It is a first-in-first-out (FIFO) data structure, meaning items are retrieved from the queue in the order they were added. We write $A \leftarrow \mathit{Queue}()$ to initialize a queue in the variable A. We can add an item x to the queue using $A.\mathit{add}(x)$. Finally, $A.\mathit{get}()$ removes and returns the oldest item in the queue. We also use the functions $A.\mathit{is_empty}()$ and $A.\mathit{is_not_empty}()$ to check whether the queue is empty or contains items, respectively.

2.1 Updatable PKEs

We review the syntax of updatable public-key encryption schemes (UPKEs).

Definition 1 (UPKE [DKW21, ABL25]). *An* updatable public key encryption (UPKE) scheme *with message space $\mathcal{M}$, private key space $\mathcal{SK}$, public key space $\mathcal{PK}$, ciphertext space $\mathcal{C}$, update token space $\mathcal{UP}$, encryption randomness space $\mathcal{R}_1$, and update randomness space $\mathcal{R}_2$ consists of five probabilistic polynomial time (*PPT*) algorithms:*

- $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$*: takes as input 1^λ, and outputs an initial public-private key pair $(\mathsf{pk}_0, \mathsf{sk}_0) \in \mathcal{PK} \times \mathcal{SK}$.*
- $c \leftarrow \mathsf{Enc}(\mathsf{pk}_i, m; r)$*: takes as input a public key $\mathsf{pk}_i \in \mathcal{PK}$ and a message $m \in \mathcal{M}$, randomness $r \in \mathcal{R}_1$ and outputs a ciphertext $c \in \mathcal{C}$.*
- $m/\perp \leftarrow \mathsf{Dec}(\mathsf{sk}_i, c)$*: takes as input a secret key $\mathsf{sk}_i \in \mathcal{SK}$ and a ciphertext $c \in \mathcal{C}$, and outputs a message $m \in \mathcal{M}$ or $\perp$ on failure.*
- $(\mathsf{pk}_{i+1}, \mathsf{up}_i) \leftarrow \mathsf{UpdPk}(\mathsf{pk}_i; r)$*: takes as input a public key $\mathsf{pk}_i \in \mathcal{PK}$, and randomness $r \in \mathcal{R}_2$ and outputs a new public key $\mathsf{pk}_{i+1} \in \mathcal{PK}$ and an update token $\mathsf{up}_i \in \mathcal{UP}$.*
- $\mathsf{sk}_{i+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_i, \mathsf{pk}_{i+1})$*: takes as input a secret key $\mathsf{sk}_i \in \mathcal{SK}$, a corresponding public key $\mathsf{pk}_i \in \mathcal{PK}$, an update token $\mathsf{up}_i \in \mathcal{UP}$ and the resultant public key $\mathsf{pk}_{i+1} \in \mathcal{PK}$, and outputs a new secret key $\mathsf{sk}_{i+1} \in \mathcal{SK}$.*

For correctness we require that for all $(\mathsf{pk}_0, \mathsf{sk}_0) \in \mathsf{KGen}(1^\lambda)$, $m \in \mathcal{M}$, and $i \in \{0, 1, 2, \dots\}$, and $(\mathsf{pk}_{j+1}, \mathsf{up}_j) \leftarrow\!\!\$\ \mathsf{UpdPk}(\mathsf{pk}_j)$, $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_j, \mathsf{pk}_j, \mathsf{up}_j, \mathsf{pk}_{j+1})$ for all $0 \le j \le i$, we have

$$\Pr[\mathsf{Dec}(\mathsf{sk}_i, \mathsf{Enc}(\mathsf{pk}_i, m)) = m] \ge 1 - \mathsf{negl}(\lambda) \ ,$$

where the probability is taken over the randomness of Enc.

Now we introduce some additional terminology for UPKEs.

Definition 2. *Let $\mathcal{SK}$ and $\mathcal{PK}$ be the secret key and public key spaces, respectively,for a UPKE. We define the set $\mathtt{Corr} \subseteq \mathcal{PK} \times \mathcal{SK}$ to be the set of all $(\mathsf{pk}_0, \mathsf{sk}_0)$ such that, for all $m \in \mathcal{M}$ and $i \in \{0, 1, 2, \dots\}$, $(\mathsf{pk}_{j+1}, \mathsf{up}_j) \leftarrow\!\!\$\ \mathsf{UpdPk}(\mathsf{pk}_j)$ and $\mathsf{sk}_{j+1} \leftarrow\!\!\$\ \mathsf{UpdSk}(\mathsf{sk}_j, \mathsf{pk}_j, \mathsf{up}_j, \mathsf{pk}_{j+1})$ for all $0 \le j \le i$, we have*

$$\Pr\big[\mathsf{Dec}(\mathsf{sk}_{i+1}, \mathsf{Enc}(\mathsf{pk}_{i+1}, m)) = m\big] \ge 1 - \mathsf{negl}(\lambda),$$

where the probability is taken over the randomness of Enc*. From now on, we will use the term keypair to refer to elements of* Corr*.*

Note that the set of all outputs of the key generation algorithm, $\mathsf{Supp}(\mathsf{KGen})$ is a subset of $\mathtt{Corr}$.

Definition 3 (Shiftable UPKE)**.** *A UPKE is called* shiftable *if* UpdPk *emits an additional private output* d *called the shift value and there exist efficient deterministic algorithms* $\mathsf{ShiftPK}, \mathsf{ShiftSK}$ *such that for any* $(\mathsf{pk}_1, \mathsf{sk}_1) \in \mathtt{Corr}$ *with*

$$(\mathsf{pk}_2, \mathsf{up}_1, d) \leftarrow\!\!\$\ \mathsf{UpdPk}(\mathsf{pk}_1) \text{ and } \mathsf{sk}_2 \leftarrow \mathsf{UpdSk}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{up}_1, \mathsf{pk}_2) \ ,$$

we have

$$\mathsf{pk}_2 = \mathsf{ShiftPK}(\mathsf{pk}_1, d) \text{ and } \mathsf{sk}_2 = \mathsf{ShiftSK}(\mathsf{sk}_1, d) \ .$$

Since it is a private output, the shift value d is not part of the syntax, but it is always visible to the entity running the function. We note that the above definition of shiftable is trivially satisfied by any UPKE by setting the shift value to the random coins used in UpdPk and including the public key in the secret key. However, later in Chapter 4, when we want to prove that an adversarially provided key is the result of an update, the above formulation is useful for moving all the extra steps outside the proof itself, making it more efficient. Consider as an example a UPKE scheme where $(\mathsf{pk}_1, \mathsf{sk}_1)$ and $(\mathsf{pk}_2, \mathsf{sk}_2)$ have an algebraic relation and the UpdPk algorithm first passes its random coins through a function H to derive this relation, applies it to pk_1 to get pk_2, and finally calls another function G to generate the update token. One way to generate such a proof would be to prove each step of this computation, including the functions H and G, which would be expensive. On the other hand, there might exist efficient proofs of the algebraic relation connecting the keypairs, which might be sufficient for the security requirements. In such cases, the latter proof would be preferable for concrete efficiency, and formulating a UPKE as shiftable allows us to talk about such scenarios generically.

For the rest of the document, we will only work with shiftable UPKEs.

Definition 4. *For a set* $\mathcal{S}$*,* $\mathtt{Updatable}(\mathcal{S})$ *denotes the set of all keypairs that can be obtained by running* UpdPk *and* UpdSk *on keypairs in* $\mathcal{S}$*, over all the randomness of* UpdPk*. We sometimes omit* $\mathcal{S}$ *when it is clear from the context. We define* $\mathtt{Updatable}^i = \mathtt{Updatable}(\mathtt{Updatable}^{i-1})$ *with* $\mathtt{Updatable}^1(\mathcal{S}) = \mathtt{Updatable}(\mathcal{S})$ *and* $\mathtt{Updatable}^* = \bigcup_{i \in \mathbb{Z}^+} \mathtt{Updatable}^i$*. For a single keypair* $k \in \mathtt{Corr}$*, we define the update closure of* k *as* $\mathtt{Updatable}^*(\{k\})$*.*

We adapt the definition of min-entropy of ciphertexts from PKEs [FO13, HHK17] to UPKEs.

Definition 5 (Min-entropy)**.** *For a public key* pk_i *and message* m *we define* $\gamma(\mathsf{pk}_i, m) := -\log_2 \max_{c \in \mathcal{C}} \Pr_{r \leftarrow \mathcal{R}}[c = \mathsf{Enc}(\mathsf{pk}_i, m; r)]$.

We say that UPKE is γ-spread if, for every keypair $(\mathsf{pk}_i, \mathsf{sk}_i) \in \mathtt{Corr}$ and every message $m \in \mathcal{M}$, $\gamma(\mathsf{pk}_i, m) \geq \gamma$. In particular, this implies that for every possible ciphertext $c \in \mathcal{C}$, $\Pr_{r \leftarrow \mathcal{R}}[c = \mathsf{Enc}(\mathsf{pk}_i, m; r)] \leq 2^{-\gamma}$.

2.2 Zero Knowledge Proofs

Formally, for a $\mathcal{NP}$ relation $\mathcal{R}$, we associate the language $\mathcal{L}_\mathcal{R} := \{x \mid \exists w : (x, w) \in \mathcal{R}\}$. We will define our NIZK-PoK using this language.

Definition 6. *A Non-Interactive Zero Knowledge system for a* $\mathcal{NP}$ *relation* $\mathcal{R}$ *is a set of three PPT algorithms* $(\mathsf{CRSGen}, \mathsf{Prove}, \mathsf{Verify})$ *with the following syntax:*

- $\mathsf{crs} \leftarrow\!\!\$\ \mathsf{CRSGen}(1^\lambda)$*: takes in the security parameter and outputs a common reference string* crs*.*
- $\pi \leftarrow\!\!\$\ \mathsf{Prove}(\mathsf{crs}, x, w)$*: takes in the* crs*, a statement* x *and a witness* w *to output a proof* π*.*
- $0/1 \leftarrow \mathsf{Verify}(\mathsf{crs}, x, \pi)$*: takes in the* crs*, statement* x *and proof* π *to output 0 or 1.*

Every non-interactive zero knowledge system should satisfy the following three properties:

Definition 7 (Completeness)**.** *A NIZK system* $(\mathsf{CRSGen}, \mathsf{Prove}, \mathsf{Verify})$ *for a* $\mathcal{NP}$ *relation* $\mathcal{R}$ *is complete if for all* $(x, w) \in \mathcal{R}$*, for* $\mathsf{crs} \leftarrow\!\!\$\ \mathsf{CRSGen}(1^\lambda), \pi \leftarrow\!\!\$\ \mathsf{Prove}(\mathsf{crs}, x, w)$*,*

$$\Pr[\mathsf{Verify}(\mathsf{crs}, x, \pi) = 1] \geq 1 - \mathsf{negl}(\lambda)$$

with the probability being taken over the random coins of CRSGen *and* Prove*.*

Definition 8 (Soundness)**.** *A NIZK system* $(\mathsf{CRSGen}, \mathsf{Prove}, \mathsf{Verify})$ *for a* $\mathcal{NP}$ *relation* $\mathcal{R}$ *is sound if for all PPT adversaries* $\mathcal{A}$*, for all* $(x, w) \in \mathcal{R}$*, for* $\mathsf{crs} \leftarrow\!\!\$\ \mathsf{CRSGen}(1^\lambda)$ *and* $(x, \pi) \leftarrow\!\!\$\ \mathcal{A}(\mathsf{crs})$*,*

$$\Pr[\mathsf{Verify}(\mathsf{crs}, x, \pi) = 1 \wedge x \notin \mathcal{L}_\mathcal{R}] \leq \mathsf{negl}(\lambda)$$

with the probability taken over the random coins of CRSGen *and* $\mathcal{A}$*.*

Definition 9 (Zero Knowledge)**.** *A NIZK system* $(\mathsf{CRSGen}, \mathsf{Prove}, \mathsf{Verify})$ *for a* $\mathcal{NP}$ *relation* $\mathcal{R}$ *is said to be* ϵ*-zero knowledge if there exists a PPT simulator* Sim *and an alternate generator* SimGen *such that no PPT adversary* $\mathcal{A}$*, with* $(x, w) \in \mathcal{R}$*,* $\mathsf{crs} \leftarrow\!\!\$\ \mathsf{CRSGen}(1^\lambda)$ *and* $(\mathsf{crs}', td) \leftarrow\!\!\$\ \mathsf{SimGen}(1^\lambda)$*, can distinguish between the following two distributions:*

$$\{\mathsf{crs}, x, \mathsf{Prove}(\mathsf{crs}, x, w)\}$$
$$\{\mathsf{crs}', x, \mathsf{Sim}(\mathsf{crs}', td, x)\}$$

with probability greater than ϵ*. That is, using the trapdoor* td*, we can simulate proofs for statements without knowing the corresponding witness.*

Additionally, we will require that our NIZK system also be a proof of knowledge that is straightline simulation extractable.

Definition 10 (SSE NIZK-PoK)**.** *A NIZK system* $(\mathsf{CRSGen}, \mathsf{Prove}, \mathsf{Verify})$ *for a* $\mathcal{NP}$ *relation* $\mathcal{R}$ *is said to be* ϵ *straightline simulation extractable proof of knowledge if there exists a PPT algorithm* Ext*, such that for any* $(x, w) \in \mathcal{R}$*, any* crs *output from either* CRSGen *or* SimGen*, and* $\pi \leftarrow\!\!\$\ \mathsf{Prove}(\mathsf{crs}, x, w; r)$ *where* r *are the random coins used by the prover, we have:*

$$w \leftarrow\!\!\$\ \mathsf{Ext}(\mathsf{crs}, \pi, x, r)$$

with failure probability ϵ*.*

In the random oracle model (ROM), there is no need for a crs, and hence there is no need for the CRSGen function either. Similarly, for simulations, the trapdoor td is the ability to program the oracle at certain points. Finally for extractability in the ROM, the random coins used by the prover is simply a transcript of all the calls it made to the RO, which is always available to the challenger. Since our proofs rely on the ROM, we assume the NIZK-POK is also instantiated in the ROM. For a NIZK instantiated in the ROM, none of the algorithms require a crs as input, and Ext no longer needs to be given the random coins of the prover as an input either.

Chapter 3

Straightline Security

In this chapter, we look at straightline security definitions, where the adversary is restricted to only updating the latest node in the security games. We define indistinguishability under chosen randomness (CR) and chosen plaintext attacks (CPA) in Section 3.1, followed by one-wayness under CR and CPA in Section 3.2 and indistinguishability under chosen updates and CPA in Section 3.3. Finally, we show the transformation for lifting a straightline chosen randomness CPA-secure UPKE to a straightline chosen updates CPA-secure UPKE and prove its security in Section 3.4.

3.1 S-IND-CR-CPA: Straightline Security, Chosen Randomness

We begin with a security notion from [JMM19], which models the security of UPKE via a game similar to IND-CPA for PKEs, but after sending the challenge ciphertext, we will run an update and reveal the secret key of the new node to the adversary. As long as the final update is done honestly by the challenger, we can let the adversary control the randomness of the previous updates to give it more power. This is called S-IND-CR-CPA security, described in Figure 3.1.

In this game, the challenger generates the first keypair $(\mathsf{pk}_0, \mathsf{sk}_0)$ and sends pk_0 to the adversary. The adversary can then ask to update the keys by providing randomness ρ to the update oracle UpdO. After a number of such calls, at some node i^*, the adversary makes its (single) challenge ciphertext query under the current public key. We allow the adversary to continue updating the key by calling UpdO, before the adversary asks for one

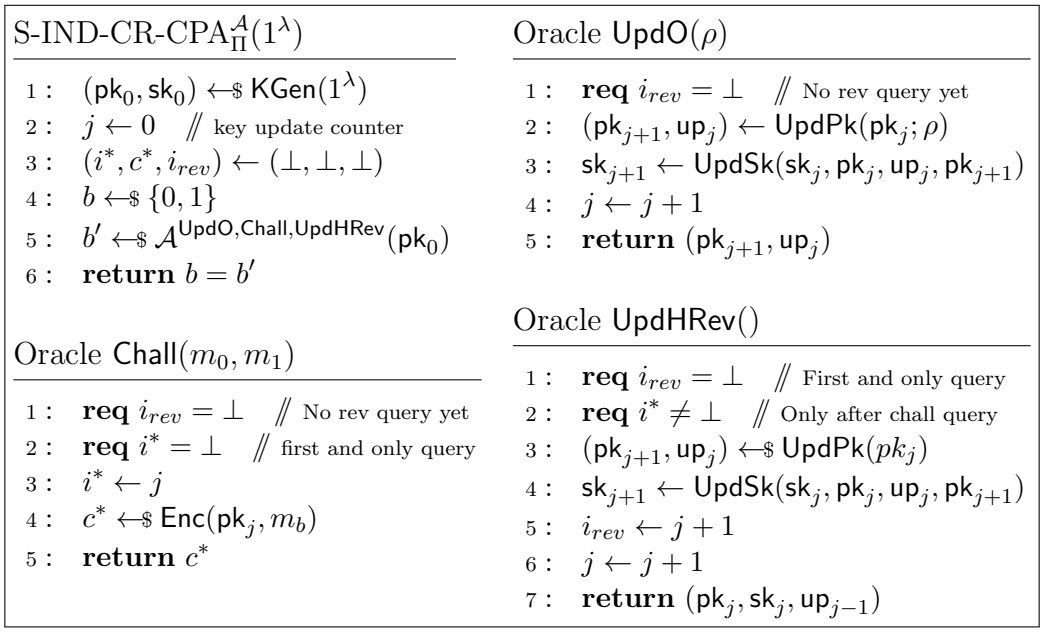


Figure 3.1: The S-IND-CR-CPA (straightline indistinguishability under chosen randomness and chosen plaintext attack) game for a UPKE scheme.

final honest update and reveal ($\mathsf{UpdHRev}$) on i_{rev}. In response to this query, the challenger performs an honest update and outputs the new public key, secret key, and the update token.

Definition 11 (S-IND-CR-CPA security of UPKE [JMM19])**.** *A UPKE scheme has straightline indistinguishability under chosen randomness, chosen plaintext attacks, if for any PPT* $\mathcal{A}$, $\mathsf{Adv}^{\text{S-IND-CR-CPA}}_{\mathcal{A},\Pi}(\lambda) = \left|\Pr[\text{S-IND-CR-CPA}^{\mathcal{A}}_{\Pi}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$*, where the* S-IND-CR-CPA *game is as described in Figure 3.1.*

There are many schemes in literature that satisfy this security definition. We mention a few examples based on various computational assumptions in Chapter 7.

As with standard PKEs, one can define an analogous security game based on One-Wayness rather than indistinguishability. We call this OW-CR-CPA security and define it in the following section.

$\text{OW-CR-CPA}^{\mathcal{A}}_{\Pi}(1^\lambda)$

1 : $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\$\ \mathsf{KGen}(1^\lambda)$
2 : $j \leftarrow 0$ // key update counter
3 : $(i^*, c^*, i_{rev}) \leftarrow (\perp, \perp, \perp)$
4 : $m^* \leftarrow\!\$\ \mathcal{M}$
5 : $m' \leftarrow\!\$\ \mathcal{A}^{\mathsf{UpdO},\mathsf{Chall},\mathsf{UpdHRev}}(\mathsf{pk}_0)$
6 : **return** $m^* = m'$

Oracle $\mathsf{Chall}()$

1 : **req** $i_{rev} = \perp$ // No rev query yet
2 : **req** $i^* = \perp$ // first and only query
3 : $i^* \leftarrow j$;
4 : $c^* \leftarrow\!\$\ \mathsf{Enc}(\mathsf{pk}_j, m^*)$
5 : **return** c^*

Oracle $\mathsf{UpdO}(\rho)$

1 : **req** $i_{rev} = \perp$ // No rev query yet
2 : $(\mathsf{pk}_{j+1}, \mathsf{up}_j) \leftarrow \mathsf{UpdPk}(\mathsf{pk}_j; \rho)$
3 : $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_j, \mathsf{pk}_j, \mathsf{up}_j, \mathsf{pk}_{j+1})$
4 : $j \leftarrow j + 1$
5 : **return** $(\mathsf{pk}_{j+1}, \mathsf{up}_j)$

Oracle $\mathsf{UpdHRev}()$

1 : **req** $i_{rev} = \perp$ // First and only query
2 : **req** $i^* \neq \perp$ // Only after chall query
3 : $(\mathsf{pk}_{j+1}, \mathsf{up}_j) \leftarrow\!\$\ \mathsf{UpdPk}(pk_j)$
4 : $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_j, \mathsf{pk}_j, \mathsf{up}_j, \mathsf{pk}_{j+1})$
5 : $i_{rev} \leftarrow j + 1$
6 : $j \leftarrow j + 1$
7 : **return** $(\mathsf{pk}_j, \mathsf{sk}_j, \mathsf{up}_{j-1})$

Figure 3.2: The OW-CR-CPA (One-Wayness under chosen randomness and chosen plaintext attack) game for a UPKE scheme.

3.2 OW-CR-CPA: One-Wayness, Chosen Randomness

The one-way security for a UPKE is defined similar to the S-IND-CR-CPA security, with the only difference being the challenge ciphertext and the expected response. Similar to one-way security for standard PKEs, in OW-CR-CPA security, the challenge ciphertext consists of a randomly encrypted message chosen by the challenger, and the adversary must correctly guess which message was encrypted to win the game. The rest of the game remains the same as S-IND-CR-CPA. This game is shown in Figure 3.2.

Definition 12 (OW-CR-CPA security of UPKE)**.** *A UPKE scheme is one-way under chosen randomness, chosen plaintext attacks, if for any PPT* $\mathcal{A}$*,* $\mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{A},\Pi}(\lambda) = \Pr[\text{OW-CR-CPA}^{\mathcal{A}}_{\Pi}(1^\lambda) = 1] = \mathsf{negl}(\lambda)$*, where the* OW-CR-CPA *game is as described in Figure 3.2.*

We also adapt the following lemma for standard PKEs to UPKEs from [HHK17]:

Lemma 1. *For any PPT* OW-CR-CPA *adversary* $\mathcal{B}$ *there exists a PPT* S-IND-CR-CPA *adversary* $\mathcal{A}$ *such that* $\mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B},\Pi}(\lambda) \leq \mathsf{Adv}^{\text{S-IND-CR-CPA}}_{\mathcal{A},\Pi}(\lambda) + 1/|\mathcal{M}|$.

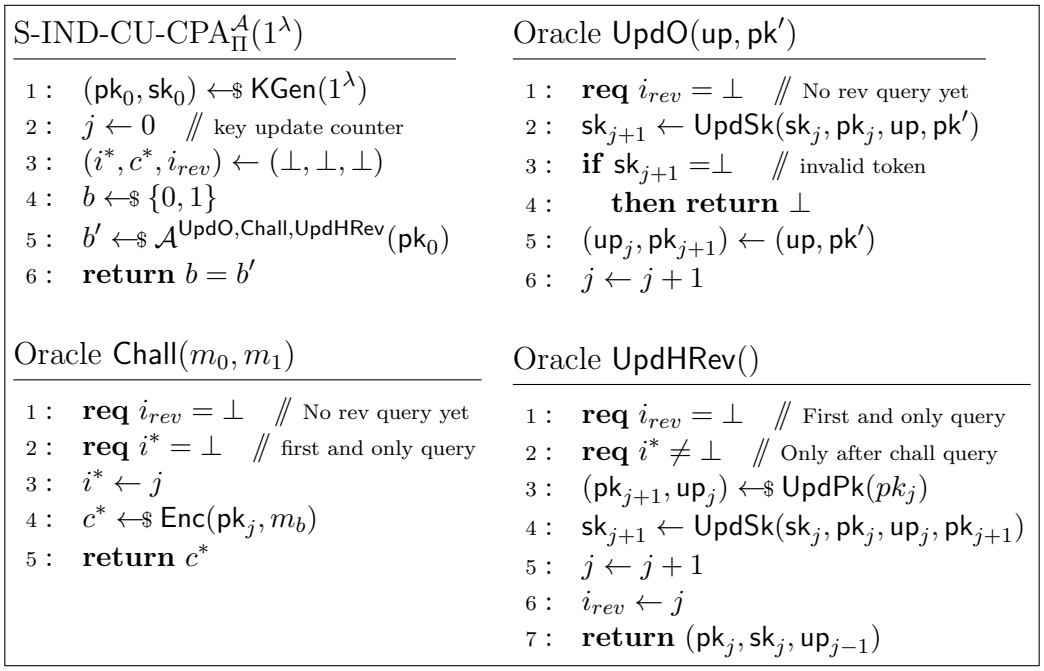


Figure 3.3: The S-IND-CU-CPA (straightline indistinguishability under chosen update and chosen plaintext attack) game for a UPKE scheme.

This is a straightforward adaptation as the update mechanism remains separate from the encryption mechanism, and they do not affect each other.

3.3 S-IND-CU-CPA: Straightline Security, Chosen Updates

Going a step further from S-IND-CR-CPA security, we can have the *chosen update* security notion (introduced in [DKW21]) by allowing the adversary to send arbitrary and potentially malformed update tokens. In particular, the adversary sends both up and pk' (the output of the UpdPk function) to the oracle. If the token and the public key are well-formed, a new node is created. Otherwise, the oracle will return $\perp$. Chosen update security is strictly stronger than chosen randomness security because the adversary can submit update tokens and public keys that were not necessarily output from the UpdPk algorithm, which is not possible in chosen randomness security. This game is shown in Figure 3.3.

$\mathsf{CU.UpdPk}(\mathsf{pk}; r)$

1 : $s \leftarrow H(r)$
2 : $(\mathsf{CR.up}, \mathsf{pk}') \leftarrow \mathsf{CR.UpdPk}(\mathsf{pk}; s)$
3 : $h \leftarrow G(r, \mathsf{CR.up}, \mathsf{pk}')$
4 : $\mathsf{CU.up} \leftarrow \mathsf{Enc}(\mathsf{pk}, r; h)$
5 : **return** $\mathsf{pk}', \mathsf{CU.up}$

$\mathsf{CU.UpdSk}(\mathsf{sk}, \mathsf{pk}, \mathsf{up}', \mathsf{pk}')$

1 : $r \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{up}')$
2 : **req** $r \neq \perp$
3 : $s \leftarrow H(r)$
4 : $(\bar{\mathsf{pk}}, \mathsf{CR.up}) \leftarrow \mathsf{CR.UpdPk}(\mathsf{pk}; s)$
5 : **req** $\bar{\mathsf{pk}} = \mathsf{pk}'$
6 : $h \leftarrow G(r, \mathsf{CR.up}, \mathsf{pk}')$
7 : $\mathsf{CU.up} \leftarrow \mathsf{Enc}(\mathsf{pk}, r; h)$
8 : **req** $\mathsf{CU.up} = \mathsf{up}'$
9 : $\mathsf{sk}' \leftarrow \mathsf{CR.UpdSk}(\mathsf{sk}, \mathsf{pk}, \mathsf{CR.up}, \mathsf{pk}')$
10 : **return** sk'

Figure 3.4: The transform for lifting a chosen randomness (CR) CPA-secure UPKE to a chosen update (CU) CPA-secure one.

Definition 13 (S-IND-CU-CPA security of UPKE [DKW21])**.** *An UPKE scheme has indistinguishability under straightline chosen update, chosen plaintext attacks, if for any PPT* $\mathcal{A}$, $\mathsf{Adv}^{\text{S-IND-CU-CPA}}_{\mathcal{A},\Pi}(\lambda) = \left|\Pr[\text{S-IND-CU-CPA}^{\mathcal{A}}_{\Pi}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$*, where the* S-IND-CU-CPA *game is as described in Figure 3.3.*

3.4 Lifting Chosen Randomness To Chosen Update Security

We use the FO transform, via derandomization and re-encryption, to convert a S-IND-CR-CPA UPKE scheme $\mathsf{CR}.\Pi = (\mathsf{CR.KGen}, \mathsf{CR.Enc}, \mathsf{CR.Dec}, \mathsf{CR.UpdPk}, \mathsf{CR.UpdSk})$ into a S-IND-CU-CPA UPKE scheme $\mathsf{CU}.\Pi = (\mathsf{CU.KGen}, \mathsf{CU.Enc}, \mathsf{CU.Dec}, \mathsf{CU.UpdPk}, \mathsf{CU.UpdSk})$; the main idea is to apply the FO transform to ensure wellformedness of the *update token*.

We set $\mathsf{CU.KGen} = \mathsf{CR.KGen}, \mathsf{CU.Enc} = \mathsf{CR.Enc}, \mathsf{CU.Dec} = \mathsf{CR.Dec}$ and then define $\mathsf{CU.UpdPk}$ and $\mathsf{CU.UpdSk}$ in Figure 3.4. In the construction, H and G are two independent random oracles. Since the function $\mathsf{CR.UpdPk}$ is run inside the transform, the private output shift value is preserved by the transform. This will be useful for the next transform.

[AW25] proposed a FO-like transform for lifting an IND-CR-CPA UPKE to IND-CU-CCA security. Our transform is a stripped-down version of the construction

of [AW25], isolating the core mechanism responsible for achieving IND-CU-CPA security and ignoring the parts responsible for achieving CCA security. This isolation helps to modularize and simplify our transforms. Our proof proceeds in a similar manner to parts of the one in [AW25].

Theorem 1. *If* $\mathsf{CR}.\Pi$ *is a (forking or straightline)* IND-CR-CPA *secure UPKE that has* $1-\mathsf{negl}(\lambda)$ *correctness, and is* γ*-spread (Definition 5), and* $H : \{0,1\}^* \mapsto \mathcal{R}_2$ *&* $G : \{0,1\}^* \mapsto \mathcal{R}_1$ *are modelled as random oracles, then* $\mathsf{CU}.\Pi$*, the UPKE obtained by applying the FO transform on* $\mathsf{CR}.\Pi$ *as described in Figure 3.4, is a (forking or straightline)* IND-CU-CPA *secure UPKE.*

In particular, for any IND-CU-CPA *adversary* $\mathcal{A}$ *that issues at most* q_{up} *queries to the update oracle and* q_H *queries to* H*, there exists a* OW-CR-CPA *adversary* $\mathcal{B}'$ *and a* IND-CR-CPA *adversary* $\mathcal{B}$ *such that*

$$\mathsf{Adv}^{\text{IND-CU-CPA}}_{\mathcal{A},\mathsf{CU}.\Pi}(\lambda) \leq q_{\mathsf{up}} \cdot 2^{-\gamma} + \mathsf{negl}(\lambda) + q_H \cdot \mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B}',\mathsf{CR}.\Pi}(\lambda) + \mathsf{Adv}^{\text{IND-CR-CPA}}_{\mathcal{B},\mathsf{CR}.\Pi}(\lambda).$$

We first give a proof outline. The main idea is that we can determine whether an update query was generated properly and the randomness used for it by observing the random-oracle queries made by $\mathcal{A}$. Since the transformation requires an RO call at each step to derive the randomness, we should be able to see all parts of the update function being carried out in the RO queries.

For the games, we first start with the unmodified game $\mathbf{G}_0$. In game $\mathbf{G}_1$, we abort if we find a honestly generated, but faulty update token. In game $\mathbf{G}_2$, we then handle the rejection of update queries without knowing the secret key, by observing the RO queries made by the adversary. There is a negligible chance here that the adversary generated a valid update token without making the corresponding RO queries. Next in game $\mathbf{G}_3$, we generate the final update token by encrypting a random value r, and show that the adversary cannot notice this difference if $\mathsf{CR}.\Pi$ is OW-CR-CPA secure. Finally, we show that any adversary that beats $\mathbf{G}_3$ can be used to construct an adversary that beats the IND-CR-CPA game of the underlying UPKE.

This proof works for reducing straight-line IND-CU-CPA to straight-line IND-CR-CPA as well as reducing forking IND-CU-CPA to forking IND-CR-CPA, with forking security defined in Chapter 4. The reduction revolves around the ability to reject malformed update tokens and to extract randomness from honestly generated tokens using random oracles. This core ability does not depend on the games being straight-line or forking, so the proof proceeds identically for both. For the sake of brevity, we only show the details of the straight-line games below.

Proof. The proof proceeds as a sequence of games against an adversary $\mathcal{A}$ against the IND-CU-CPA security of $\mathsf{CU}.\Pi$. Let $\Pr[\mathbf{G}_i]$ be the probability that the adversary wins the i^{th} game.

Game $\mathbf{G}_0$: The IND-CU-CPA game against $\mathcal{A}$. Everything proceeds as normal. In addition to the oracles in the game, $\mathcal{A}$ also has access to two random oracles H and G. This game is shown in Figure 3.5.

By definition, we have:

$$|\Pr[\mathbf{G}_0] - 1/2| = \mathsf{Adv}^{\text{S-IND-CU-CPA}}_{\mathcal{A},\mathsf{CU}.\Pi}(\lambda).$$

Game $\mathbf{G}_1$: Same as $\mathbf{G}_0$, except we abort the game if the adversary had submitted an honestly generated faulty update token at any point in the game. Since the base scheme has $1 - \mathsf{negl}(\lambda)$ correctness, there is a chance that a ciphertext outputs a faulty payload upon decryption. Such an event can result in a honestly generated update token, created by querying the corresponding ROs, failing the re-encryption check, and being rejected. We check if we encountered such tokens at the end of the game. The modified main function of the game is shown in Figure 3.6.

The probability of encountering such a faulty update token is directly tied to the decryption failure probability of underlying PKE. If the adversary makes q_{up} queries to the update oracle, then we get:

$$|\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_0]| \leq q_{\mathsf{up}} \cdot \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$$

With this change, we can condition on the adversary never submitting such a faulty update token in the remaining games.

Game $\mathbf{G}_2$: Same as $\mathbf{G}_1$, except that update queries are rejected without knowledge of the secret key. The challenger can do this by observing the RO calls made by $\mathcal{A}$. If $\mathcal{A}$ calls UpdO on $\mathsf{up}', \mathsf{pk}'$, then the challenger looks for $r, s, \mathsf{up}, \mathsf{pk}', h$ such that $\big(s = H(r) \wedge (\mathsf{up}, \mathsf{pk}') \leftarrow \mathsf{UpdPk}(\mathsf{pk}; s) \wedge h = G(r, \mathsf{up}, \mathsf{pk}') \wedge \mathsf{up}' \leftarrow \mathsf{Enc}(\mathsf{pk}, r; h)\big)$. Since we now expect all update tokens to pass the re-encryption check, if such a tuple can be found, then it is considered a valid update with randomness r for Π; otherwise, it is rejected. Since $\mathcal{A}$ can only make polynomially many RO queries, we can do this operation in polynomial time. The modified oracles are shown in Figure 3.7.

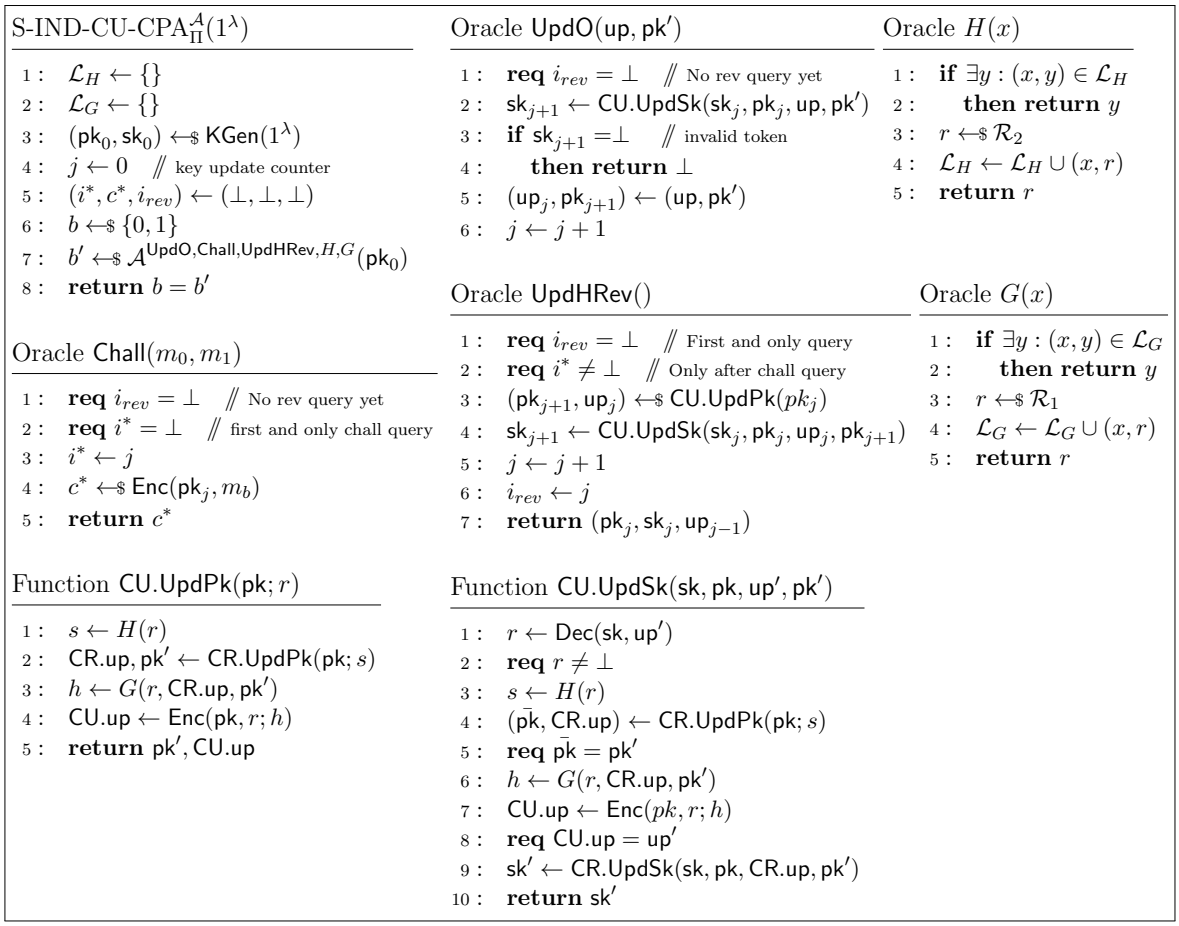


Figure 3.5: $\mathbf{G}_0$ for the proof of Theorem 1 for lifting a indistinguishability under chosen randomness CPA-secure UPKE to a indistinguishability under chosen updates CPA-secure one.

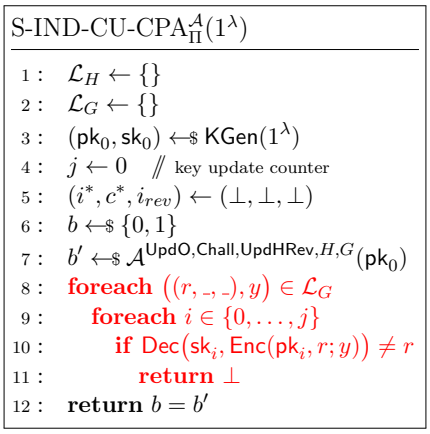


Figure 3.6: The modified $\mathbf{G}_1$ for the proof of Theorem 1 for lifting a indistinguishability under chosen randomness CPA-secure UPKE to a indistinguishability under chosen updates CPA-secure one.

The games continue identically until $\mathcal{A}$ produces an update token that is valid without first querying the corresponding ROs. Since G is a random oracle, its output is uniformly random till it is queried. For $\mathcal{A}$ to submit a valid token without querying it, it needs to guess the output of G, which is randomness of the ciphertext that forms the update token. Since $\mathsf{CR}.\Pi$ is γ-spread, the probability of a successful guess is at most $2^{-\gamma}$. To see why, recall that γ-spreadness implies that, for any public key pk, any message m, and any ciphertext c, $\Pr_{r \leftarrow\$ \mathcal{R}_1}[\mathsf{Enc}(\mathsf{pk}, m; r) = c] \leq 2^{-\gamma}$, where r is chosen uniformly at random and independently of (pk, m, c). Setting r to be the output of the underlying G query, if the query has never been made before, then r will indeed be both uniformly random and independent of (pk, m, c). Thus, the probability of successfully guessing the resulting ciphertext is at most $2^{-\gamma}$. If the adversary makes q_{up} queries to the update oracle, then by the union bound, we have:

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq q_{\mathsf{up}} \cdot 2^{-\gamma}.$$

Game $\mathbf{G}_3$: Same as $\mathbf{G}_2$, except the update token for the reveal query is generated without the knowledge of the secret key. This is done by sampling a random value r and setting $\mathsf{up}' \leftarrow \mathsf{Enc}(pk, r)$. The modified oracle is shown in Figure 3.8.

Note that until the adversary makes the query on $H(r)$ or $G(r, \mathsf{up}, \mathsf{pk}')$, the outputs of those queries are uniformly random and the adversary cannot tell that the token was generated incorrectly. So this game continues identically to the previous one until the ad-

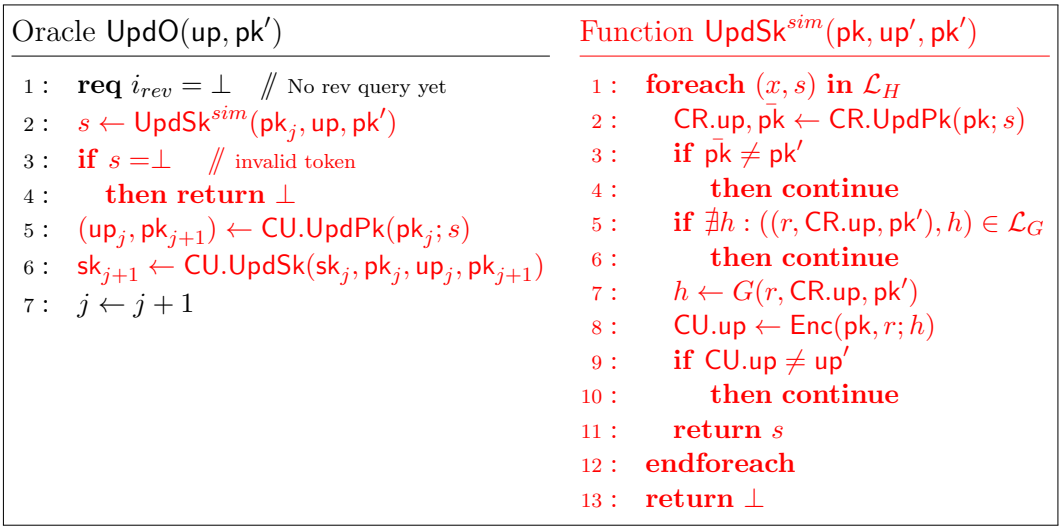


Figure 3.7: Modified oracles in $\mathbf{G}_2$ for the proof of Theorem 1 for lifting a chosen randomness CPA-secure UPKE to a chosen update CPA-secure one.

versary makes either of those queries. If any of the queries occur, then the adversary knows r, and it has broken the One-Wayness of the UPKE (which is related to IND-CR-CPA). So if $\mathcal{A}$ makes such a query, then we can construct an adversary $\mathcal{B}'$ that uses $\mathcal{A}$ to win the OW-CR-CPA game against $\mathsf{CR}.\Pi$.

Our constructed adversary $\mathcal{B}'$ interacts with the OW-CR-CPA challenger and interacts with $\mathcal{A}$. After receiving pk_0 from the challenger, $\mathcal{B}'$ passes it to $\mathcal{A}$. For every UpdO query that $\mathcal{A}$ makes, $\mathcal{B}'$ can either reject it or extract the randomness and make the corressponding query at the challenger using the random oracles. Upon $\mathcal{A}$'s challenge query on (m_0, m_1), $\mathcal{B}'$ chooses a one of the messages and returns its encryption under the current public key. It then continues handling UpdO queries like before. Finally when $\mathcal{A}$ asks for the $\mathsf{UpdHRev}$ query, $\mathcal{B}'$ first asks its challenger for the challege query on the same node and receives c^*. It then asks for the $\mathsf{UpdHRev}$ query itself to receive $(\mathsf{pk}_{rev}, \mathsf{sk}_{rev}, \mathsf{up}')$, $\mathcal{B}'$ then responds with $(\mathsf{pk}_{rev}, \mathsf{sk}_{rev}, c^*)$, providing the OW-CR-CPA challenge as the forged update token. After $\mathcal{A}$ outputs its guess, $\mathcal{B}'$ looks at all the H queries made by $\mathcal{A}$ and outputs one of them at uniformly at random. The advantage of $\mathcal{B}'$ is q_H^{-1} times the probability of $\mathcal{A}$ making the query $H(r)$ in $\mathbf{G}_3$, where q_H is the number of queries to the H oracle made by $\mathcal{A}$.

This gives us:

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq q_H \cdot \mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B}', \mathsf{CR}.\Pi}(\lambda).$$

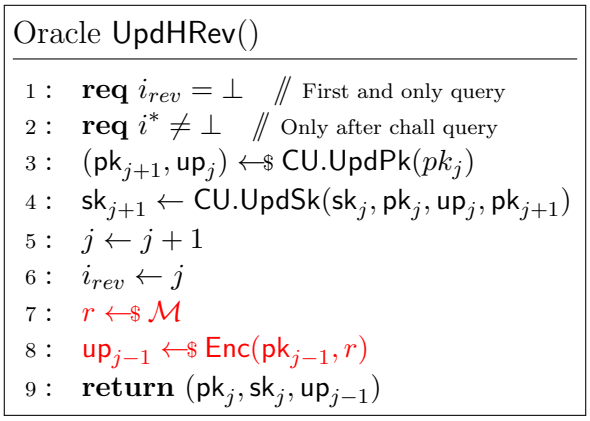


Figure 3.8: Modified oracle in $\mathbf{G}_3$ for the proof of Theorem 1 for lifting a chosen randomness CPA-secure UPKE to a chosen update CPA-secure one.

Analysis of Game $\mathbf{G}_3$: Finally, if an adversary can beat $\mathbf{G}_3$, then it can be used by an adversary $\mathcal{B}$ to beat the IND-CR-CPA game of $\mathsf{CR}.\Pi$ because $\mathcal{B}$ can simulate $\mathbf{G}_3$ perfectly by interacting with a IND-CR-CPA challenger for $\mathsf{CR}.\Pi$. The construction of this adversary is shown in Figure 3.9.

The adversary $\mathcal{B}$ has access to IND-CR-CPA oracles $\mathsf{Ch.UpdO}, \mathsf{Ch.Chall}, \mathsf{Ch.UpdHRev}$ from its challenger that it uses to simulate the $\mathbf{G}_3$ oracles $\mathsf{UpdO}, \mathsf{Chall}, \mathsf{UpdHRev}$ for $\mathcal{A}$.

Note that $\mathcal{B}$ perfectly simulates $\mathcal{A}$'s view from $\mathbf{G}_3$. The Chall oracles are identical between the reduction and $\mathbf{G}_3$. For the UpdO oracle, $\mathcal{B}$ still rejects tokens the same way as in $\mathbf{G}_3$, but calls the challenger's UpdO oracle to perform the update instead of using the randomness to call the associated functions itself. For the $\mathsf{UpdHRev}$ oracle, $\mathcal{B}$ calls the corresponding oracle provided by the challenger to get the updated public key and secret key, but then generates the update token like in $\mathbf{G}_3$.

It is clear that $\mathcal{B}$ wins the game against its challenger with the same probability that $\mathcal{A}$ wins in $\mathbf{G}_3$. Hence, we have:

$$|\Pr[\mathbf{G}_3] - 1/2| = \mathsf{Adv}^{\text{IND-CR-CPA}}_{\mathcal{B},\mathsf{CR}.\Pi}(\lambda).$$

Combining all the bounds accumulated during the game hops yields the desired bound in the theorem. □

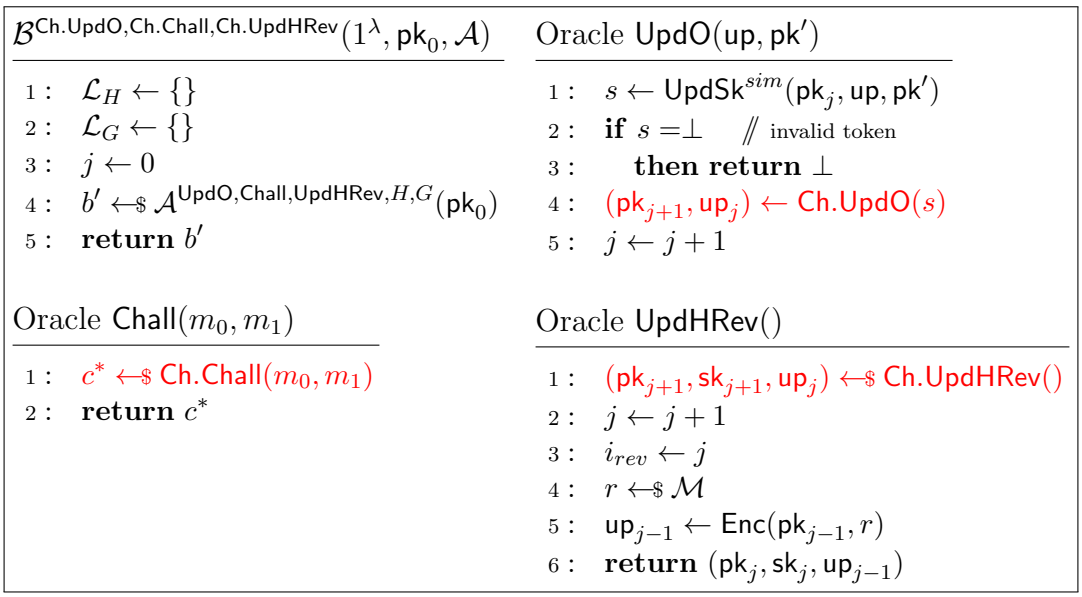


Figure 3.9: Construction of $\mathcal{B}$ that uses IND-CU-CPA adversary $\mathcal{A}$ to win the IND-CR-CPA game for the proof of Theorem 1.

Chapter 4

Adding Forking And Public Verifiability

In this chapter, we add forking security and public verifiability to our system of transforms. We start by recalling the definitions of forking security and public verifiability (PV) as introduced in [AFM24]. Then, in Section 4.1 we define our intermediate half-forking security notion and present our transform to achieve in Section 4.2. Similarly, in Section 4.3 we define the full forking security notion with public verifiability and show the transform achieving it in Section 4.4.

Forking. MLS and TreeKEM enforce some level of synchronization to ensure message ordering. This synchronization ensures that updates are ordered and received in a straight line. However, we must consider an adversary that may control the network and reorder messages differently for each participant in the chat. In such scenarios, while the updates appear to be in a straight line from the users' perspectives, there is actually forking from the adversary's perspective. Hence, security must hold even in the presence of forking/concurrent updates.

Whereas the straight-line games involve a straight chain of updates, the forking game allows the adversary to update any node i (not necessarily the latest one), thereby inducing a tree structure with forks. The adversary may request the challenge ciphertext at a node i^*, issue further updates, and request key revelation after an honest update on another arbitrarily chosen node.

Public verifiability. Since the adversary may update keys for an offline receiver, we want any party—not just the receiver—to be able to verify the validity of a newly formed public key; otherwise, the adversary could trivially win by registering arbitrary keys. To enable this, every update message, including adversarially issued ones, must include a publicly verifiable proof attesting to the validity of the new public key. This allows any party to verify the new public key independently of the secret-key holder. Note that the goal here is to protect the confidentiality of messages encrypted under updated keys, but we do not aim to provide any guarantees regarding the validity or correctness of the update itself.

To achieve public verifiability, we modify the UpdPk algorithm to output a proof attesting to the linkage between the old and new public keys. This proof is then checked by a verification algorithm $\mathsf{VerifyPk}$. Receivers must also verify this proof before updating their secret keys. Formally, we add the following algorithms:

- $(\mathsf{pk}_{i+1}, \mathsf{up}_i, \pi_i) \leftarrow\!\!\$\ \mathsf{UpdPk}(\mathsf{pk}_i)$: The public update algorithm takes in a public key pk_i and outputs the updated public key pk_{i+1}, the update token up_i, and a proof π_i.
- $0/1 \leftarrow \mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}_{i+1}, \pi_i)$: The verify algorithm takes in the old public key pk_i, the new public key pk_{i+1}, and the proof π_i, and outputs 0/1.

With the correctness requirement that for any $(\mathsf{pk}_{i+1}, \mathsf{up}_i, \pi_i) \leftarrow\!\!\$\ \mathsf{UpdPk}(\mathsf{pk}_i)$, $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}_{i+1}, \pi_i)$ should output 1.

Modelling security. With PV, we want to protect other members who do not have the secret key (non-receivers) against maliciously generated public key updates. These other members use the proofs to decide whether they can trust the updated key; if a proof is valid, then the confidentiality of the messages encrypted under the key should be maintained. To capture this idea, we extend the security game by introducing a new oracle called $\mathsf{VerifyO}$, which models the behavior of non-receivers in the presence of potentially malicious updates.

- $\mathsf{VerifyO}(i, \mathsf{pk}', \pi')$: The oracle runs $\mathsf{VerifyPk}$ on $(\mathsf{pk}_i, \mathsf{pk}', \pi')$. If the output is 0, it returns $\perp$. If the output is 1, it creates a new child of node i and sets its public key as pk'.

Since the child is created using only the proof, not the update token, the challenger is not expected to know the child's secret key. Such nodes created by $\mathsf{VerifyO}$ are called *half*

nodes, since they only have public keys. Other nodes created by the UpdO oracle that have both public and secret keys are called *full nodes*. The adversary can ask for the challenge ciphertext on these half nodes.

Naturally, the adversary cannot ask for corrupt queries or honest-update-and-reveal queries on half nodes. Hence, if an adversary ever creates a half node in a straight-line game, it loses the ability to ask for a $\mathsf{UpdHRev}$ query and consequently gets less information to win the game. Therefore, half nodes are useless in the straight-line security games, which is why we add forking before or alongside adding public verifiability.

4.1 HF-IND-CU-CPA-PV: Half-Forking, Chosen Updates

As we have shown so far, public verifiability and forking are closely intertwined. Hence, in our series of transforms, we choose to add PV first by introducing an intermediate notion called *half-forking*, which we later upgrade to full forking security in subsequent transforms. In half-forking security, we augment the straight-line game by allowing the adversary to create forks using the $\mathsf{VerifyO}$ oracle. The result is that our update tree consists of a straight chain of full nodes with half nodes forming branches. In this game the adversary can ask for the challenge query at any node, including full nodes it previously created, in contrast to the straight-line game. This game is shown in Figure 4.1. This game does not correspond to any real-world adversary since half-forking is an arbitrary restriction on forking, but it serves as a convenient intermediate notion to make our transforms and their respective proofs easier.

Definition 14 (HF-IND-CU-CPA-PV security of UPKE)**.** *An UPKE scheme satisfies indistinguishability under half-forking, chosen update, chosen plaintext attacks with public verifiability, if for any PPT* $\mathcal{A}$, $\mathsf{Adv}^{\text{HF-IND-CU-CPA-PV}}_{\mathcal{A},\Pi}(\lambda) = \left|\Pr[\text{HF-IND-CU-CPA-PV}^{\mathcal{A}}_{\Pi}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$*, where the* HF-IND-CU-CPA-PV *game is described in Figure 4.1.*

4.2 Lifting Straightline to Half-Forking Security

Ciphertext updatability. To lift a straightline-secure UPKE to the half-forking security setting, we will use a property called *ciphertext updatability*. Informally, this property captures the ability to update ciphertexts using the same shift value that was used to update the corresponding keys. Specifically, if a keypair $(\mathsf{pk}_1, \mathsf{sk}_1)$ was updated to $(\mathsf{pk}_2, \mathsf{sk}_2)$ using a shift value d, we should be able to convert a ciphertext encrypting a message under

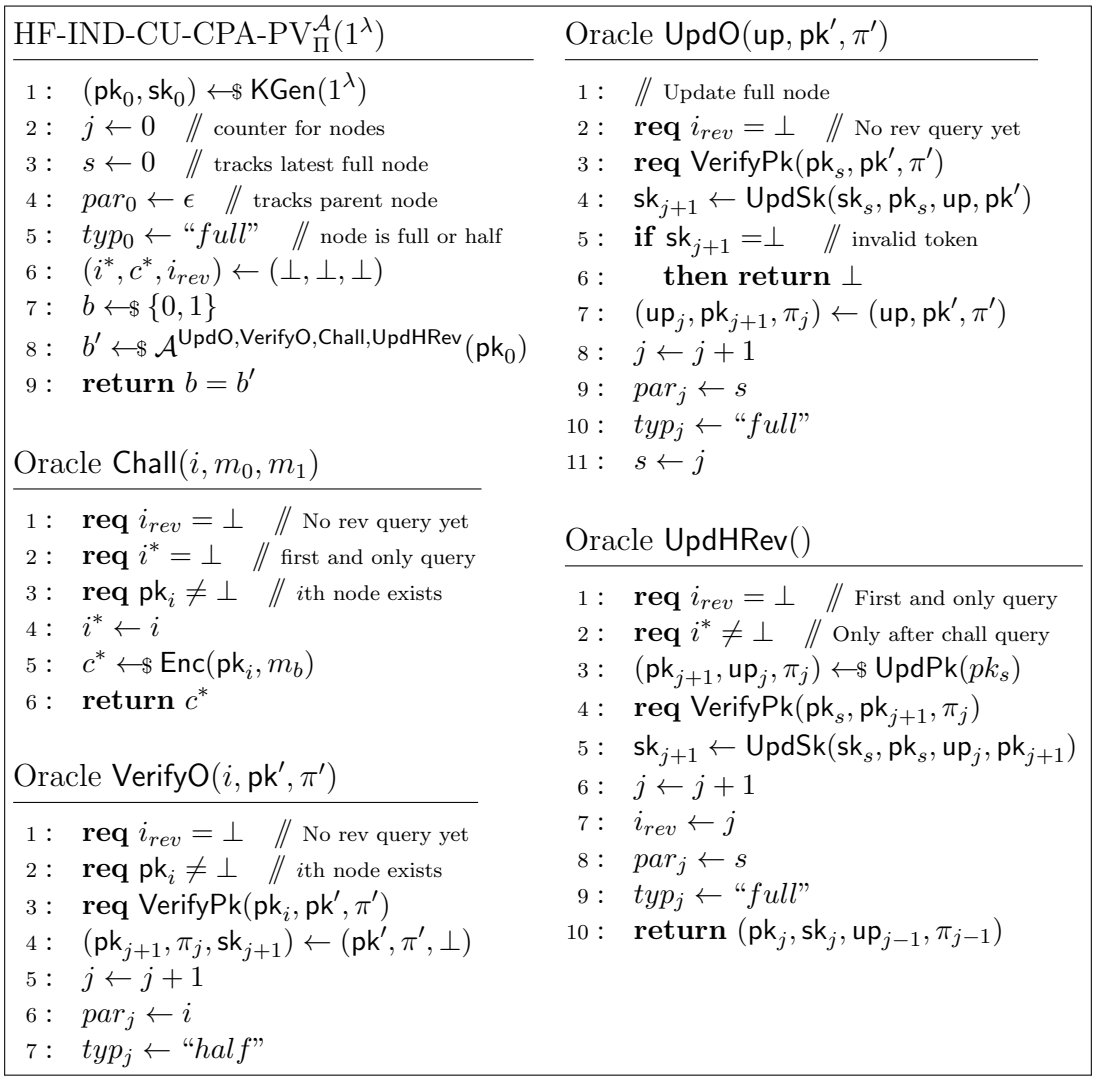


Figure 4.1: The HF-IND-CU-CPA-PV (half-forking indistinguishability under chosen updates and chosen plaintext attacks with public verifiability) game for a UPKE scheme.

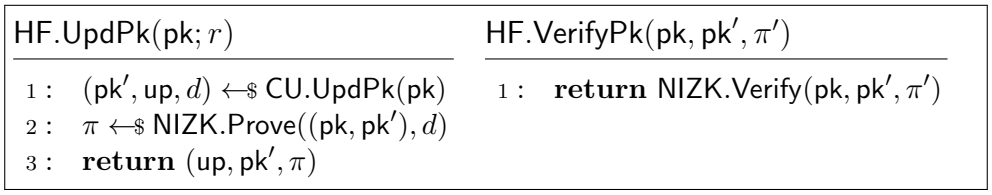


Figure 4.2: The transform for lifting a straightline chosen update and CPA-secure UPKE to a half-forking chosen update CPA-secure with public verifiability one. The NIZK-PoK is a proof for the relation $R((\mathsf{pk}, \mathsf{pk}'), d) = 1$ iff $\mathsf{pk}' = \mathsf{ShiftPK}(\mathsf{pk}, d)$, where d is the witness.

pk_1 to another ciphertext encrypting the same message under pk_2, using the same shift value d, while maintaining correctness.

We will assume that this is possible unconditionally, meaning the shifted ciphertexts are statistically indistinguishable from fresh encryptions under pk_2. However, our methodology could easily be extended to work with updated ciphertexts that are only computationally indistinguishable from non-updated ones, or with update procedures that introduce a small correctness penalty.

Definition 15. *Let* Π *be a shiftable UPKE.* Π *is said to have* ciphertext updatability *if there exists an algorithm* ShiftCt *such that, for any pair* $(\mathsf{pk}_1, \mathsf{sk}_1) \in \mathtt{Corr}$*, shift value* d *and message* m*, letting* $\mathsf{pk}_2 \leftarrow \mathsf{ShiftPK}(\mathsf{pk}_1, d)$*,* $c_1 \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{pk}_1, m)$*, and* $c_2' \leftarrow \mathsf{ShiftCt}(c_1, d)$*, we have* (pk_2, c_2') *is statistically close to* $(\mathsf{pk}_2, \mathsf{Enc}(\mathsf{pk}_2, m))$*.*

We will use ciphertext updatability in our reduction to update the challenge ciphertext using the shift values extracted from the NIZK-PoKs used for PV.

Our transform for lifting an S-IND-CU-CPA secure UPKE to an HF-IND-CU-CPA-PV secure one only changes the UpdPk algorithm, and introduces a new Verify algorithm. Recall that the private output shift value of UpdPk is preserved by the previous transform used for CU security. We make use of this shift value as a witness to generate the NIZK-PoK proof. The verify algorithm for the resultant UPKE is the NIZK-PoK verify algorithm. The remaining algorithms remain unchanged, and the transform is shown in Figure 4.2.

Theorem 2. *Let* CU.Π *be a* S-IND-CU-CPA *secure UPKE with* $1 - \mathsf{negl}(\lambda)$ *correctness obtained using the transform in Figure 3.4 on a* IND-CR-CPA *secure UPKE* CR.Π *with the random oracles* $H : \{0,1\}^* \mapsto \mathcal{R}_2$ *and* $G : \{0,1\}^* \mapsto \mathcal{R}_1$*. Assume* CU.$\Pi$ *is* γ*-spread and is ciphertext updatable. Then* HF.Π*, the UPKE obtained by applying the transform in Figure 4.2 to* CU.Π *along with an* ϵ_{ZK}*-zero knowledge and* ϵ_{Ext}*-straightline simulation extractable NIZK-PoK, is* HF-IND-CU-CPA-PV *secure UPKE.*

For any HF-IND-CU-CPA-PV *adversary* $\mathcal{A}$*, making at most* q_{up} *queries to the* UpdO *oracle, and at most* q_{Verify} *queries to the* VerifyO *oracle, there exists a* S-IND-CU-CPA *adversary* $\mathcal{B}$ *such that*

$$\mathsf{Adv}^{\text{HF-IND-CU-CPA-PV}}_{\mathcal{A},\mathsf{HF}.\Pi}(\lambda) \leq q_{\mathsf{up}} \cdot 2^{-\gamma} + q_{\mathsf{Verify}} \cdot \epsilon_{\mathsf{Ext}} + \epsilon_{\mathsf{ZK}} + \mathsf{negl}(\lambda) + \mathsf{Adv}^{\text{S-IND-CU-CPA}}_{\mathcal{B},\mathsf{CU}.\Pi}(\lambda).$$

We first start by giving an **outline of the proof**. For the reduction, we need to be able to answer the adversary's challenge queries on half-nodes while having access only to challenge queries on full nodes. The idea then is to extract all shift values connecting the half node to its closest full node and use them, along with ciphertext updatability, to convert a challenge ciphertext on the full node to one on our desired half node. A subtle point to note: In the HF-IND-CU-CPA-PV game, the adversary can ask for a challenge ciphertext on any node it has created in the past, but in the straightline game, it can only ask for the challenge on the latest node. To bridge this gap, we do not create the challenge node at our challenger until the adversary asks for the challenge query.

For the games, we first start with the unmodified HF-IND-CU-CPA-PV game in $\mathbf{G}_0$. In game $\mathbf{G}_1$, we abort if we find a honestly generated, but faulty update token. In game $\mathbf{G}_2$, we then handle the update queries without knowing the secret key, by observing the RO queries made by the adversary, as we did in the previous theorem. Like before, there is a negligible chance here that the adversary generated a valid update token without making the corresponding RO queries. In game $\mathbf{G}_3$, if the challenge query is asked on a half node, we extract shift values connecting that node to its first ancestor that is full using the Ext algorithm, and abort if the extraction fails. In $\mathbf{G}_4$, when asked for a challenge query on a half node, we generate the challenge query on its first ancestor that is full and shift the ciphertext to the appropriate node using the extracted shift values. In $\mathbf{G}_5$, we generate the final proof in the UpdHRev query using the simulator. Finally, we show that any adversary that beats $\mathbf{G}_5$ can be used in a reduction to beat the S-IND-CU-CPA game against CU.Π using the idea described above.

Proof. Now we give the **full proof**. It proceeds as a sequence of games against an adversary $\mathcal{A}$ for the HF-IND-CU-CPA-PV security of HF.Π. Let $\Pr[\mathbf{G}_i]$ be the probability that the adversary wins the i^{th} game.

Game $\mathbf{G}_0$: The HF-IND-CU-CPA-PV game against $\mathcal{A}$. Everything proceeds as normal. In addition to the oracles in the game, $\mathcal{A}$ also has access to two random oracles H and G. This game is shown in Figure 4.3.

Clearly, we have:

$$|\Pr[\mathbf{G}_0] - 1/2| = \mathsf{Adv}^{\text{HF-IND-CU-CPA-PV}}_{\mathcal{A},\mathsf{HF}.\Pi}(\lambda).$$

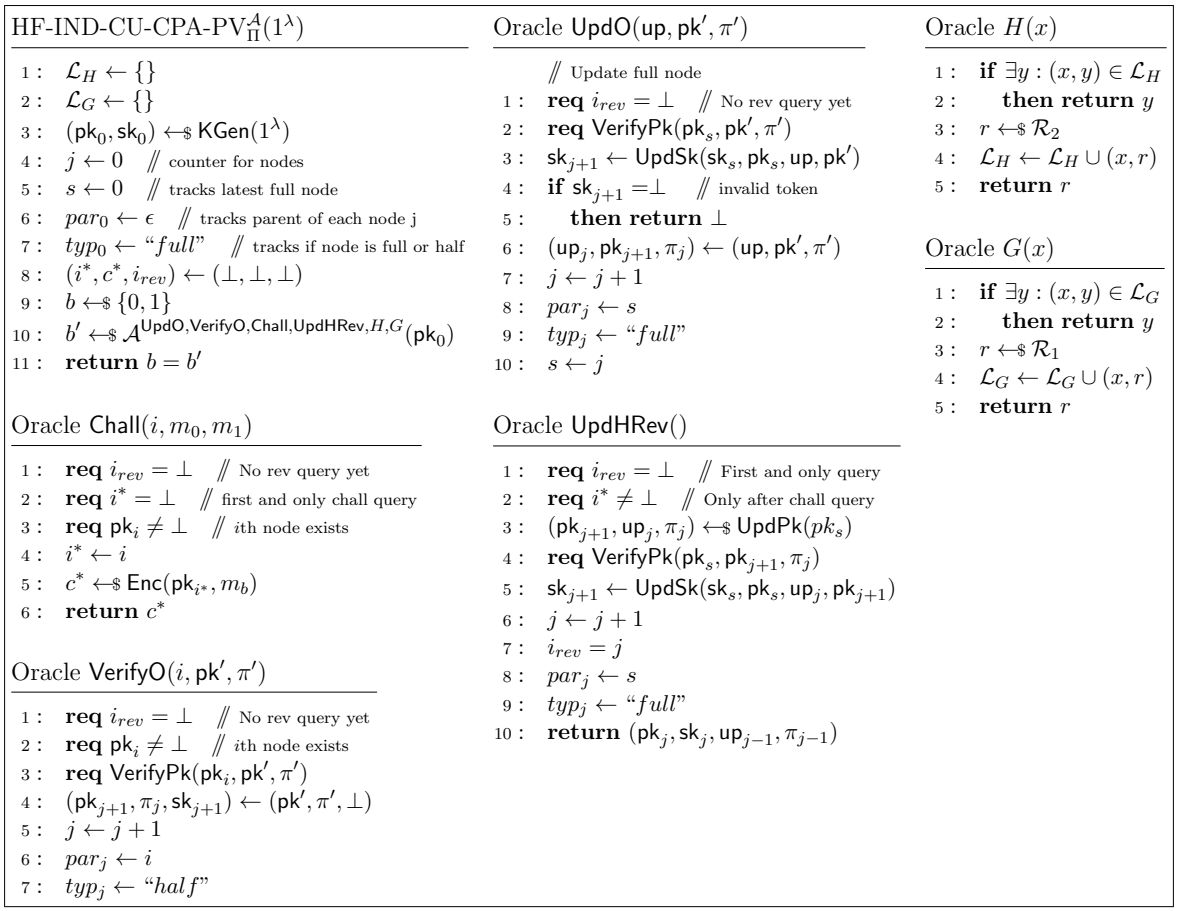


Figure 4.3: The game $\mathbf{G}_0$ for the proof of Theorem 2 for lifting straightline chosen update and CPA-secure UPKE to a half-forking chosen update CPA-secure one with public verifiability.

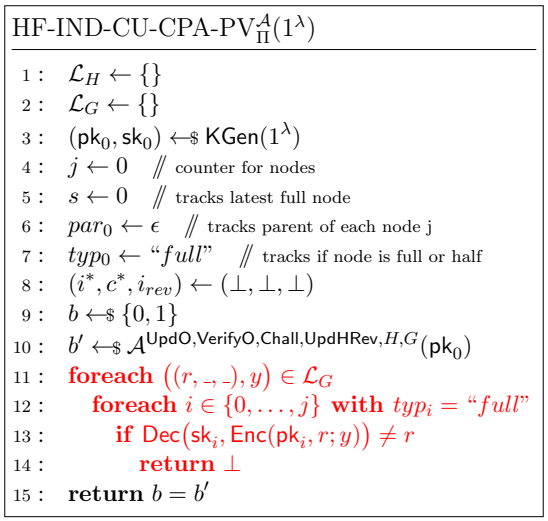


Figure 4.4: The modified $\mathbf{G}_1$ for the proof of Theorem 2 for lifting straightline chosen update and CPA-secure UPKE to a half-forking chosen update CPA-secure one with public verifiability.

Game $\mathbf{G}_1$: Same as $\mathbf{G}_0$, except we abort the game if the adversary had submitted an honestly generated faulty update token at any point in the game. Since the base scheme has $1 - \mathsf{negl}(\lambda)$ correctness, there is a chance that a ciphertext outputs a faulty payload upon decryption. Such an event can result in a honestly generated update token, created by querying the corresponding ROs, failing the re-encryption check, and being rejected. We check if we encountered such tokens at the end of the game. The modified main function of the game is shown in Figure 4.4.

The probability of encountering such a faulty update token is directly tied to the decryption failure probability of underlying PKE. If the adversary makes q_{up} queries to the update oracle, then we get:

$$|\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_0]| \leq q_{\mathsf{up}} \cdot \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$$

With this change, we can condition on the adversary never submitting such a faulty update token in the remaining games.

Game $\mathbf{G}_2$: Same as $\mathbf{G}_1$, except that malformed update queries are rejected without knowledge of the secret key. The challenger can do this by observing the RO calls made by $\mathcal{A}$. If $\mathcal{A}$ calls UpdO on $\mathsf{up}', \mathsf{pk}', \pi'$, then the challenger first verifies π' and if that

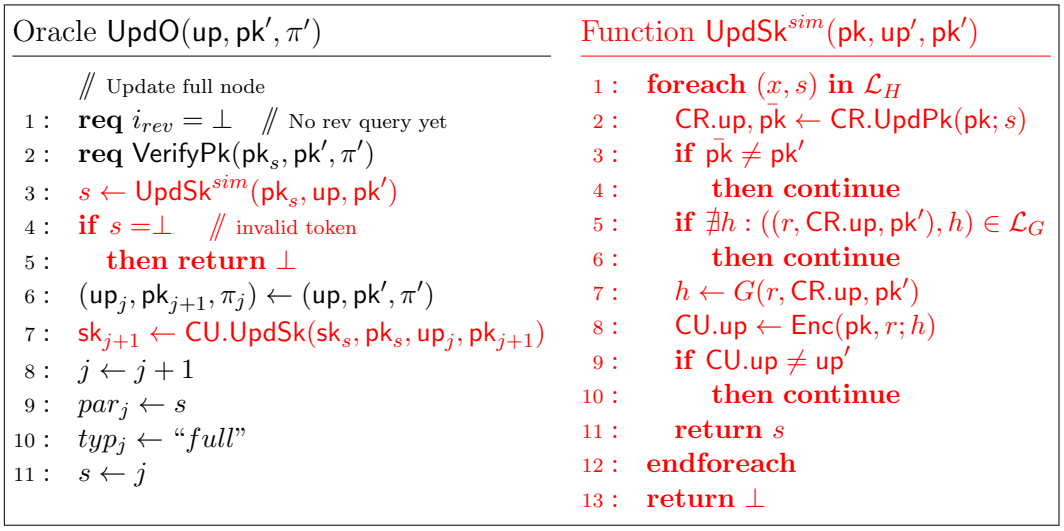


Figure 4.5: Modified oracles in $\mathbf{G}_2$ for the proof of Theorem 2 for lifting straightline chosen update and CPA-secure UPKE to a half-forking chosen update CPA-secure one with public verifiability.

passes, it looks for $r, s, \mathsf{up}, \mathsf{pk}', h$ such that $\big(s = H(r) \wedge (\mathsf{up}, \mathsf{pk}') \leftarrow \mathsf{UpdPk}(\mathsf{pk}; s) \wedge h = G(r, \mathsf{up}, \mathsf{pk}') \wedge \mathsf{up}' \leftarrow \mathsf{Enc}(\mathsf{pk}, r; h)\big)$. Since we now expect all update tokens to pass the re-encryption check, if such a tuple can be found, then it is considered a valid update with randomness r for $\mathsf{CU}.\Pi$, otherwise not. Since $\mathcal{A}$ can only make polynomially many RO queries, we can do this operation in polynomial time. The modified oracles are shown in Figure 4.5.

The games continue identically until $\mathcal{A}$ produces an update token that is valid without first querying the corresponding ROs. This occurs with probability $2^{-\gamma}$ due to the γ-spread property of $\mathsf{CU}.\Pi$, as we have shown before in Section 3.4. If the adversary makes q_{up} queries to the UpdO oracle, then by the union bound we have:

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq q_{\mathsf{up}} \cdot 2^{-\gamma}.$$

Game $\mathbf{G}_3$: Same as $\mathbf{G}_2$, except that if the adversary asks for the challenge query on a half node i, we extract the shift value along the path from i's first full ancestor to i using the NIZK-PoK extractor. The extractor only fails with a probability ϵ_{Ext} since the NIZK is ϵ_{Ext} proof of knowledge. If the extractor fails, we abort. Otherwise, the query response is generated normally, and the rest of the game remains unchanged. The modified oracle is shown in Figure 4.6.

Oracle $\mathsf{Chall}(i, m_0, m_1)$ (in $\mathbf{G}_3$)

1 : **req** $i_{rev} = \bot$ // No rev query yet
2 : **req** $i^* = \bot$ // first and only chall query
3 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
4 : $i^* \leftarrow i$
5 : $A \leftarrow \mathit{Stack}()$
6 : **while** $\mathit{typ}_i = \text{“}\mathit{half}\text{”}$
7 : $d \leftarrow\!\$\ \mathsf{Ext}(\pi_{i-1}, \mathsf{pk}_{par_i}, \mathsf{pk}_i)$
8 : **req** $d \neq \bot$
9 : $A.\mathit{push}(d)$
10 : $i \leftarrow \mathit{par}_i$
11 : $c^* \leftarrow\!\$\ \mathsf{Enc}(\mathsf{pk}_{i^*}, m_b)$
12 : **return** c^*

Oracle $\mathsf{Chall}(i, m_0, m_1)$ (in $\mathbf{G}_4$)

1 : **req** $i_{rev} = \bot$
2 : **req** $i^* = \bot$
3 : **req** $\mathsf{pk}_i \neq \bot$
4 : $i^* \leftarrow i$
5 : $A \leftarrow \mathit{Stack}()$
6 : **while** $\mathit{typ}_i = \text{“}\mathit{half}\text{”}$
7 : $d \leftarrow\!\$\ \mathsf{Ext}(\pi_{i-1}, \mathsf{pk}_{par_i}, \mathsf{pk}_i)$
8 : **req** $d \neq \bot$
9 : $A.\mathit{push}(d)$
10 : $i \leftarrow \mathit{par}_i$
11 : $c^* \leftarrow\!\$\ \mathsf{Enc}(\mathsf{pk}_i, m_b)$
12 : **while** $A.\mathit{is_not_empty}()$
13 : $c^* \leftarrow \mathsf{ShiftCt}(c^*, A.\mathit{pop}())$
14 : **return** c^*

Figure 4.6: The modified oracles in game $\mathbf{G}_3$ (left) and $\mathbf{G}_4$ (right) for the proof of Theorem 2 for lifting straightline chosen update and CPA-secure UPKE to a half-forking chosen update CPA-secure one with public verifiability.

The games continue identically until $\mathcal{A}$ asks for a challenge query on a half node and the extraction procedure fails. This occurs with very low probability due to the straightline simulation extractability property of the NIZK-PoK. If the adversary makes q_{Verify} queries to the $\mathsf{VerifyO}$ oracle, then by the union bound we have:

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq q_{\mathsf{Verify}} \cdot \epsilon_{\mathsf{Ext}}.$$

Game $\mathbf{G}_4$: Same as $\mathbf{G}_3$ except the response to the challenge query on a half node is handled differently. If the adversary asks for the challenge query on a half node i, we generate the ciphertext on the first full ancestor of i. Then, using the shift values that were extracted, and $\mathsf{ShiftCt}$, we update the ciphertext to one that is encrypted under the public key of i. We send this updated ciphertext to the adversary. The modified oracle is shown in Figure 4.6.

The games continue identically until the adversary asks for the challenge query on a half node. The response here is generated differently, but using our assumption of ciphertext updatability, the adversary should not be able to notice the difference. Hence,

the adversary's success rate in this game is the same as in the previous game.

$$\Pr[\mathbf{G}_4] = \Pr[\mathbf{G}_3].$$

Game $\mathbf{G}_5$: Same as $\mathbf{G}_4$, except for the final-update-and-reveal query, the proof associated with the update is generated using a simulator. Since the NIZK-PoK is ϵ_{ZK}-zero knowledge, the adversary can only notice this change with probability ϵ_{ZK}. This is a simple modification and we omit an explicit pseudo code description of the modified oracle.

The games continue identically until this final update, where the simulator is used. The difference in the success probabilities of the adversary between the two games is tied to the zero knowledge property of the underlying NIZK-PoK.

$$|\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_4]| \leq \epsilon_{\mathsf{ZK}}.$$

Analysis of Game $\mathbf{G}_5$: Finally, if an adversary can beat $\mathbf{G}_5$, then it can be used by an adversary $\mathcal{B}$ to beat the S-IND-CU-CPA game of $\mathsf{CU}.\Pi$ because $\mathcal{B}$ can simulate $\mathcal{A}$'s view in $\mathbf{G}_5$ perfectly by interacting with a S-IND-CU-CPA challenger for $\mathsf{CU}.\Pi$. The modified oracles in this reduction are shown in Figure 4.7. The verify oracle $\mathsf{VerifyO}$ remains unchanged.

In the HF-IND-CU-CPA-PV game, the adversary can ask for the challenge query on any node it has created in the past, but in the S-IND-CU-CPA game, we can only ask for the challenge query on the latest node. To address this discrepancy, the reduction $\mathcal{B}$ uses a Queue to store the update tokens and public keys until it is ready to create those nodes at the challenger. So if an update is accepted by the UpdO oracle, the update token and public key get stored in the Queue. When the adversary asks for the challenge query, we recreate the desired full node at the challenger by using the tokens and public keys from the queue and calling the $\mathsf{Ch.UpdO}$ oracle. Note that this use of a Queue to delay updates is undetectable to the adversary, as all tokens will remain valid and be accepted by the challenger. Hence, this does not change $\mathcal{A}$'s success probability. Similarly, for the $\mathsf{UpdHRev}$ query, we recreate the final node by using up all the tokens and public keys in the queue and then calling $\mathsf{Ch.UpdHRev}$.

Overall, $\mathcal{B}$ wins the game with the same probability that $\mathcal{A}$ wins $\mathbf{G}_5$.

$$|\Pr[\mathbf{G}_5] - 1/2| \leq \mathsf{Adv}^{\text{S-IND-CU-CPA}}_{\mathcal{B},\mathsf{CU}.\Pi}(\lambda).$$

Combining all the bounds accumulated during the game hops yields the desired bound in the theorem. □

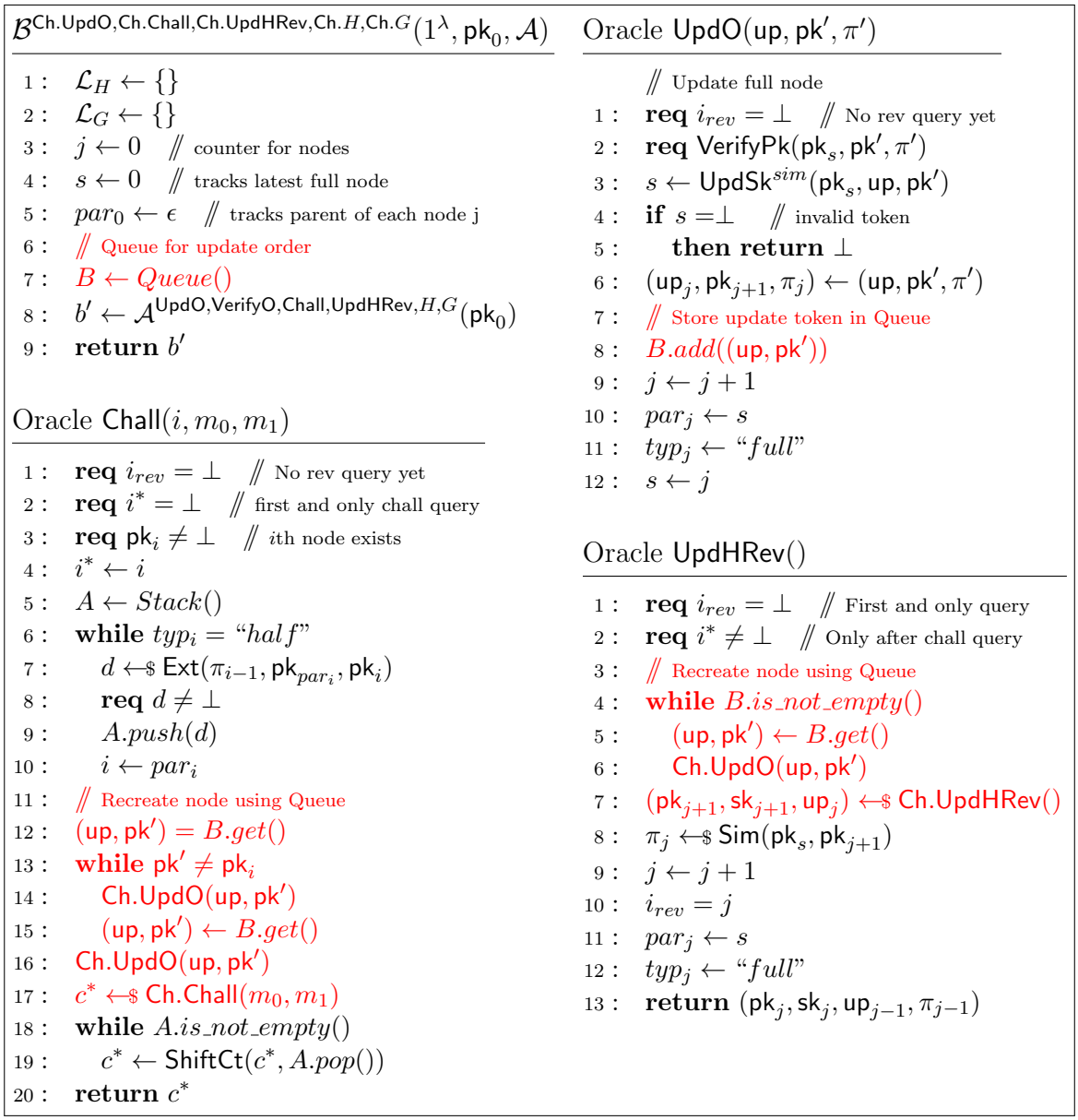


Figure 4.7: Modified oracles in the construction of $\mathcal{B}$ that uses HF-IND-CU-CPA-PV adversary $\mathcal{A}$ to win the S-IND-CU-CPA game for the proof of Theorem 2.

4.3 F-IND-CU-CPA-PV **Security**

Next, we will upgrade half-forking security to full forking. The F-IND-CU-CPA-PV security game for a UPKE is quite similar to the HF-IND-CU-CPA-PV one, with the only major difference being the UpdO and $\mathsf{UpdHRev}$ oracles. In the half-forking game, the update oracle would only allow the adversary to update the latest full node; with full forking, we now also accept an index i and enable the adversary to update any full node it had created in the past. With this change, the update tree does not have a fixed structure and branches arbitrarily depending on the calls made by the adversary. We also allow the adversary to ask the honest-update-and-reveal query on any full node of its choice, so the $\mathsf{UpdHRev}$ oracle in the full forking game accepts an index i as part of its arguments. This game is shown in Figure 4.8.

Definition 16 (F-IND-CU-CPA-PV security)**.** *An UPKE scheme satisfies indistinguishability under forking chosen update, chosen plaintext attacks with public verifiability, if for any PPT* $\mathcal{A}$, $\mathsf{Adv}^{\text{F-IND-CU-CPA-PV}}_{\mathcal{A},\Pi}(\lambda) = \left|\Pr[\text{F-IND-CU-CPA-PV}^{\mathcal{A}}_{\Pi}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$ *where the* F-IND-CU-CPA-PV *game is as described in Figure 4.8.*

4.4 Lifting Half-Forking to Forking Security

To enable our transform from half-forking to full-forking security, we introduce a new property: *update uniformity*. At a high level, this captures the idea that the result of a single update can be any key in the update closure. While there always exists a series of shift values connecting a key to any other key in the closure, it is not necessarily true that there is a *single* update randomness that does this. Update uniformity means that such a randomness always exists, or at least that the adversary cannot detect when it does not exist.

If update uniformity holds (and the scheme is OW-CR-CPA-secure), we can use it to forge the response to the "honest-update-and-reveal" query on a node by querying a completely different node at the challenger. We do so by giving out the keypair obtained from the same query on another node, forging the update token, and simulating the proof. We will show that the adversary does not detect this forgery in our security proofs, but first, we define update uniformity below.

Definition 17 (Update Uniformity)**.** *A UPKE scheme* Π *is said to have* update uniformity

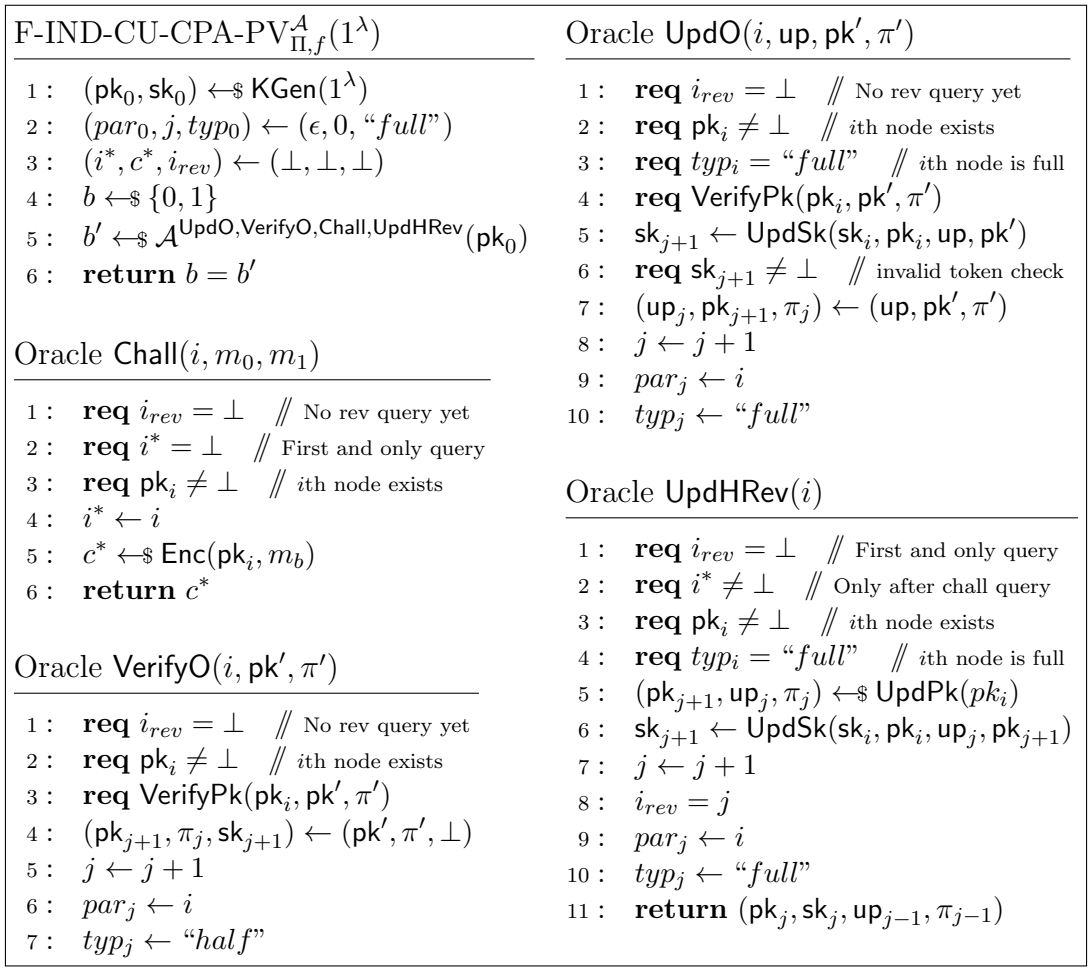


Figure 4.8: The F-IND-CU-CPA-PV (forking indistinguishability under chosen updates and chosen plaintext attacks with public verifiability) game for a UPKE scheme.

if for any two keypairs $(\mathsf{pk}_1, \mathsf{sk}_1)$ *and* $(\mathsf{pk}_2, \mathsf{sk}_2)$ *from the same update closure, and for*

$$(\mathsf{pk}'_1, \mathsf{up}_1, \pi_1) \leftarrow\!\$\ \mathsf{UpdPk}(\mathsf{pk}_1),\ \ \mathsf{sk}'_1 \leftarrow \mathsf{UpdSk}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{up}_1, \mathsf{pk}'_1)$$
$$\textit{and}\ (\mathsf{pk}'_2, \mathsf{up}_2, \pi_2) \leftarrow\!\$\ \mathsf{UpdPk}(\mathsf{pk}_2),\ \ \mathsf{sk}'_2 \leftarrow \mathsf{UpdSk}(\mathsf{sk}_2, \mathsf{pk}_2, \mathsf{up}_2, \mathsf{pk}'_2)$$

no PPT adversary $\mathcal{A}$ *can distinguish with non-negligible advantage between:*

$$(\mathsf{pk}_1, \mathsf{pk}_2, \mathsf{sk}'_1, \mathsf{pk}'_1) \quad \textit{and} \quad (\mathsf{pk}_1, \mathsf{pk}_2, \mathsf{sk}'_2, \mathsf{pk}'_2),$$

where the probability is taken over the randomness of UpdPk.

With this property, we can show that our UPKE is secure in the full forking setting.

Theorem 3. *Let* $\mathsf{HF}.\Pi$ *be a* HF-IND-CU-CPA-PV *UPKE with* $1 - \mathsf{negl}(\lambda)$ *correctness that is* γ*-spread and obtained by applying the transforms described in Figure 3.4 and Figure 4.2 on a* OW-CR-CPA *secure UPKE* $\mathsf{CR}.\Pi$ *and* $H : \{0,1\}^* \mapsto \mathcal{R}_2$ *&* $G : \{0,1\}^* \mapsto \mathcal{R}_1$ *modelled as random oracles. If* $\mathsf{HF}.\Pi$ *has update uniformity, and the NIZK-PoK is* ϵ_{ZK}*-zero knowledge, then* $\mathsf{HF}.\Pi$ *is also a* F-IND-CU-CPA-PV *secure UPKE.*

For any F-IND-CU-CPA-PV *adversary* $\mathcal{A}$ *that makes at most* q_{up} *queries to the update oracle and* q_H *queries to* H*, there exists a* OW-CR-CPA *adversary* $\mathcal{B}'$ *and a* HF-IND-CU-CPA-PV *adversary* $\mathcal{B}$ *such that*

$$\begin{aligned}\mathsf{Adv}^{\text{F-IND-CU-CPA-PV}}_{\mathcal{A},\mathsf{HF}.\Pi}(\lambda) \leq\ & q_{\mathsf{up}} \cdot 2^{-\gamma} + q_H \cdot \mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B}',\mathsf{CR}.\Pi}(\lambda) \\ & + \epsilon_{\mathsf{ZK}} + \mathsf{negl}(\lambda) + \mathsf{Adv}^{\text{HF-IND-CU-CPA-PV}}_{\mathcal{B},\mathsf{HF}.\Pi}(\lambda).\end{aligned}$$

We start with an **outline of the proof**. There are two main ideas in the reduction: the first is that we do not need to commit to creating a challenge node until the adversary asks for the challenge ciphertext. We can handle all queries before the challenge node ourselves and then recreate the exact challenge node at our challenger once the adversary asks for a challenge query. A small point to keep in mind when doing this is that the indices used for the nodes will differ between the reduction and the challenger. The second idea is that using update uniformity, we can "forge" honest updates. If we provide a fake update token and a simulated proof connecting two public keys within the same update closure, the one-wayness of the UPKE and the ZK property of the NIZK-PoK ensure that the adversary cannot distinguish it from real updates with properly generated tokens and proofs. So when $\mathcal{A}$ asks for the $\mathsf{UpdHRev}$ query, we return an updated keypair from a different full node, forge the update token by encrypting a random value, and provide the associated NIZK-PoK proof using the simulator.

For the games, we first start with the unmodified F-IND-CU-CPA-PV game in $\mathbf{G}_0$. In game $\mathbf{G}_1$, we abort if we find a honestly generated, but faulty update token. In game $\mathbf{G}_2$, we handle the rejection of update queries without knowledge of the secret key, like in the previous proof. In game $\mathbf{G}_3$, the associated proof of the final update is created using the simulator. In game $\mathbf{G}_4$, the update token of the final update is created by encrypting a random value, and the adversary cannot tell the difference due to the random oracles and One-Wayness of the UPKE. In game $\mathbf{G}_5$ the keys in the final update are derived from the first full ancestor of the challenge node instead of the one asked, and the adversary cannot tell the difference due to our assumption of update uniformity. Finally we show that if an adversary can beat the game $\mathbf{G}_5$, it can beat the HF-IND-CU-CPA-PV against $\mathsf{HF}.\Pi$.

The idea of the reduction is to handle all queries ourselves after receiving the public key from the challenger, until the adversary makes the challenge query. When the challenge query is made, we recreate the node at the challenge and pass the query. Finally we handle all the remaining queries ourselves and the final honest update and reveal query is handled as in $\mathbf{G}_5$ by calling the corresponding query on the latest full node at the challenger.

Proof. We will now give the full proof. It proceeds as a sequence of games against an adversary $\mathcal{A}$ that breaks the F-IND-CU-CPA-PV security of $\mathsf{HF}.\Pi$. Let $\Pr[\mathbf{G}_i]$ be the probability that the adversary wins the i^{th} game.

Game $\mathbf{G}_0$: The forking IND-CU-CPA with PV game against $\mathcal{A}$. Everything proceeds as normal. In addition to the oracles in the game, $\mathcal{A}$ also has access to two random oracles H and G. This game is shown in Figure 4.9.

By definition, we have:

$$|\Pr[\mathbf{G}_0] - 1/2| = \mathsf{Adv}^{\text{F-IND-CU-CPA-PV}}_{\mathcal{A},\mathsf{HF}.\Pi}(\lambda).$$

Game $\mathbf{G}_1$: Same as $\mathbf{G}_0$, except we abort the game if the adversary had submitted an honestly generated faulty update token at any point in the game. Since the base scheme has $1 - \mathsf{negl}(\lambda)$ correctness, there is a chance that a ciphertext outputs a faulty payload upon decryption. Such an event can result in a honestly generated update token, created by querying the corresponding ROs, failing the re-encryption check, and being rejected. We check if we encountered such tokens at the end of the game. The modified main function of the game is shown in Figure 4.10.

The probability of encountering such a faulty update token is directly tied to the decryption failure probability of underlying PKE. If the adversary makes q_{up} queries to the

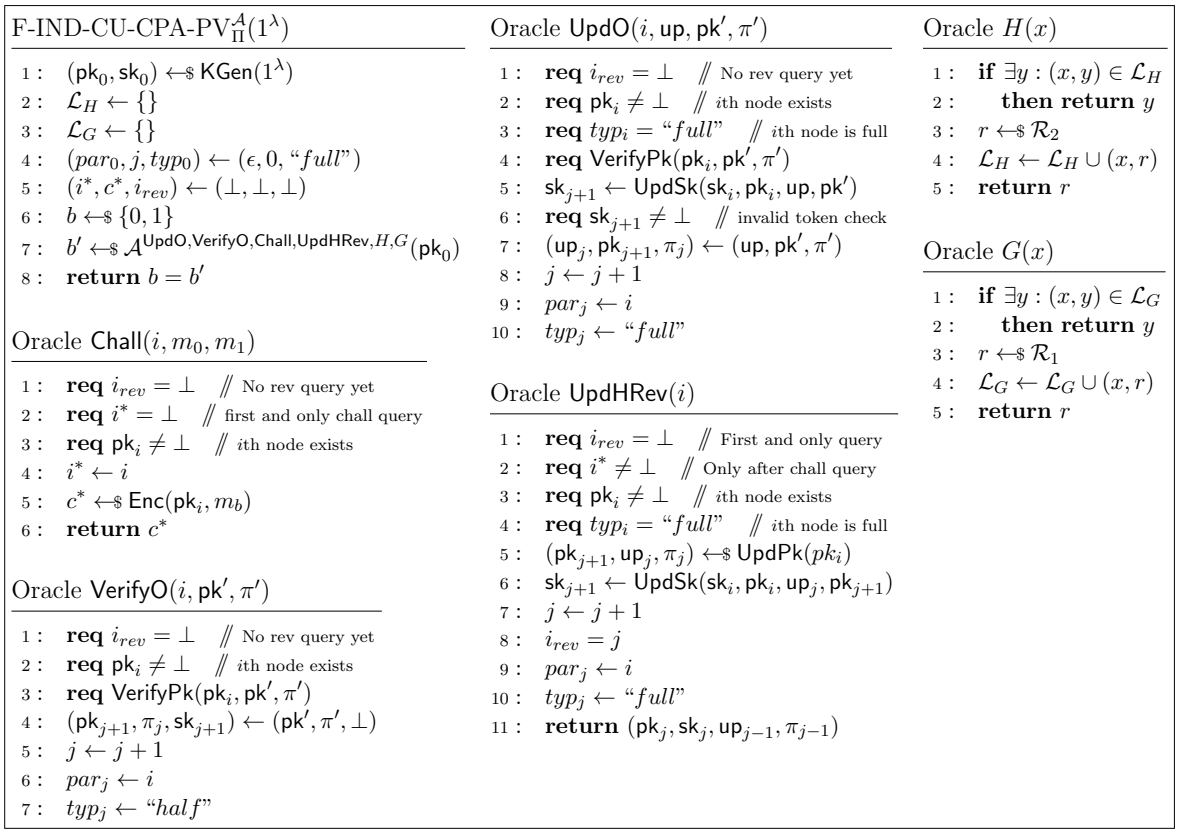


Figure 4.9: The game $\mathbf{G}_0$ for the proof of Theorem 3 for lifting half-forking chosen update and CPA-secure UPKE with public verifiability to a forking chosen update CPA-secure one with public verifiability.

update oracle, then we get:

$$|\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_0]| \leq q_{\mathsf{up}} \cdot \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$$

With this change, we can condition on the adversary never submitting such a faulty update token in the remaining games.

Game $\mathbf{G}_2$: Same as $\mathbf{G}_1$, except that malformed update queries are rejected without knowledge of the secret key. The challenger can do this by observing the RO calls made by $\mathcal{A}$. If $\mathcal{A}$ calls update on $\mathsf{up}', \mathsf{pk}', \pi'$, the challenger first verifies π' and if that passes, it looks for $r, s, \mathsf{up}, \mathsf{pk}', h$ such that $\big(s = H(r) \wedge (\mathsf{up}, \mathsf{pk}') \leftarrow \mathsf{UpdPk}(\mathsf{pk}; s) \wedge h = G(r, \mathsf{up}, \mathsf{pk}') \wedge \mathsf{up}' \leftarrow$

F-IND-CU-CPA-PV$^{\mathcal{A}}_{\Pi}(1^\lambda)$

$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$
$\mathcal{L}_H \leftarrow \{\}$
$\mathcal{L}_G \leftarrow \{\}$
$(par_0, j, typ_0) \leftarrow (\epsilon, 0, \text{“}full\text{”})$
$(i^*, c^*, i_{rev}) \leftarrow (\bot, \bot, \bot)$
$b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{UpdO},\mathsf{VerifyO},\mathsf{Chall},\mathsf{UpdHRev},H,G}(\mathsf{pk}_0)$
foreach $\big((r,_,_), y\big) \in \mathcal{L}_G$
  foreach $i \in \{0,\ldots,j\}$ **with** $typ_i = \text{“}full\text{”}$
    if $\mathsf{Dec}\big(\mathsf{sk}_i, \mathsf{Enc}(\mathsf{pk}_i, r; y)\big) \neq r$
      return $\bot$
return $b = b'$

Figure 4.10: Modified oracles in $\mathbf{G}_1$ for the proof of Theorem 3 for lifting half-forking chosen update and CPA-secure UPKE with public verifiability to a forking chosen update CPA-secure one with public verifiability.

$\mathsf{Enc}(\mathsf{pk}, r; h)\big)$. Since we now expect all update tokens to pass the re-encryption check, if such a tuple can be found, then it is considered a valid update with randomness r for $\mathsf{HF}.\Pi$, otherwise not. The modified oracles are shown in Figure 4.11.

The games continue identically until $\mathcal{A}$ produces an update token which is valid without querying the corresponding ROs first. This occurs with probability $2^{-\gamma}$ due to the γ-spread property of $\mathsf{HF}.\Pi$, as we have shown before in Section 3.4. If the adversary makes q_{up} queries to the update oracle, then by the union bound we have:

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq q_{\mathsf{up}} \cdot 2^{-\gamma}.$$

Game $\mathbf{G}_3$: Same as $\mathbf{G}_2$ except the response to the final update and reveal query is slightly altered. Instead of responding with the correct proof π from the update, we use a simulated proof π' for connecting the two public keys. So the response consists of the correctly generated update token up, correctly generated keypair $(\mathsf{pk}, \mathsf{sk})$ and the simulated proof π'. This is a simple modificiation so we omit an explicit pseudocode description of the modified oracle.

It is clear that the games continue identically until this final update where the simulator is used. The difference in success probability of the adversary between the two games is

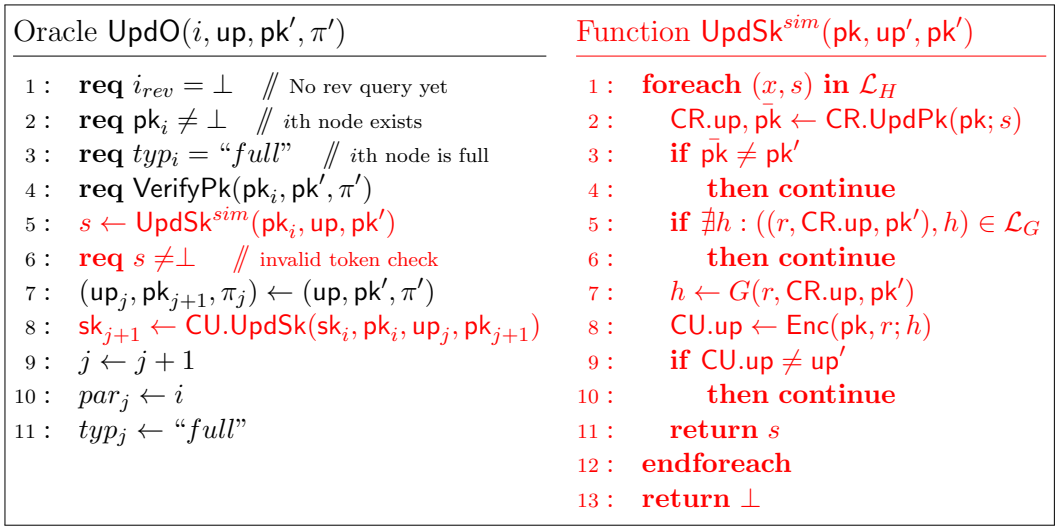


Figure 4.11: Modified oracles in $\mathbf{G}_2$ for the proof of Theorem 3 for lifting half-forking chosen update and CPA-secure UPKE with public verifiability to a forking chosen update CPA-secure one with public verifiability.

tied to the zero knowledge property of the NIZK-PoK.

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq \epsilon_{\mathsf{ZK}}.$$

Game $\mathbf{G}_4$: Same as $\mathbf{G}_3$ but we yet again modify the response in the final update and reveal query. Let this query be on node i. This time instead of outputting the correct update token, we sample a random value r and set $\mathsf{up}' \leftarrow \mathsf{Enc}(pk_i, r)$. So the response consists of the incorrectly generated update token up', correctly generated keypair $(\mathsf{pk}, \mathsf{sk})$ and the simulated proof π'. This is yet another simple modification and the pseudocode description is omitted.

Note that until the adversary makes the query on $H(r)$ or $G(r, \mathsf{up}', \mathsf{pk})$, their outputs are uniformly random and the adversary cannot tell that the token was generated incorrectly. So this game continues identically to the previous one until the adversary makes either of those queries. If any of the queries occur, then the adversary has broken the One-Wayness of the UPKE (which is related to IND-CR-CPA). So if $\mathcal{A}$ makes such a query, then we can have an adversary $\mathcal{B}'$ that uses $\mathcal{A}$ to win the OW-CR-CPA game against $\mathsf{CR}.\Pi$.

The construction of $\mathcal{B}'$ is similar to the one we have shown before in Section 3.4. If q_H

Oracle $\mathsf{UpdHRev}(i)$

1 : **req** $i_{rev} = \bot$ // First and only query
2 : **req** $i^* \neq \bot$ // Only after chall query
3 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
4 : **req** $typ_i =$ "$full$" // ith node is full
5 : $i' \leftarrow i;\ i \leftarrow i^*$
6 : **while** $typ_i \neq$ "$full$"
7 : $\quad i \leftarrow par_i$
8 : $(\mathsf{pk}_{j+1}, \mathsf{up}_j, \pi_j) \leftarrow\!\$\ \mathsf{UpdPk}(pk_i)$
9 : $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_j, \mathsf{pk}_{j+1})$
10 : $\pi_j \leftarrow\!\$\ \mathsf{Sim}(\mathsf{pk}_{i'}, \mathsf{pk}_{j+1})$
11 : $r \leftarrow\!\$\ \mathcal{R}_1$
12 : $\mathsf{up}_j \leftarrow\!\$\ \mathsf{Enc}(\mathsf{pk}_{i'}, r)$
13 : $j \leftarrow j + 1$
14 : $i_{rev} = j$
15 : $par_j \leftarrow i$
16 : $typ_j \leftarrow$ "$full$"
17 : **return** $(\mathsf{pk}_j, \mathsf{sk}_j, \mathsf{up}_{j-1}, \pi_{j-1})$

Figure 4.12: The modified oracle $\mathbf{G}_5$ for the proof of Theorem 3 for lifting half-forking chosen update and CPA-secure UPKE with public verifiability to a forking chosen update CPA-secure one with public verifiability.

is the number of queries to H, this gives us:

$$|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_3]| \leq q_H \cdot \mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B}',\mathsf{CR}.\Pi}(\lambda).$$

Game $\mathbf{G}_5$: Same as $\mathbf{G}_4$ but we modify the response in the final update and reveal query one last time. This time instead of outputting the correct keypair $(\mathsf{pk}, \mathsf{sk})$, we run a honest update on the first full ancestor of the challenge node to get the keypair $(\mathsf{pk}', \mathsf{sk}')$ and output that instead. So the response now consists of the incorrectly generated update token up', incorrectly generated keypair $(\mathsf{pk}', \mathsf{sk}')$ and the simulated proof π'. The final modified $\mathsf{UpdHRev}$ oracle is shown in Figure 4.12.

The games continue identically until the adversary receives this final response. From our assumption of update uniformity, the adversary should not be able to tell the difference between the games, and hence its success probability remains unchanged.

$$\Pr[\mathbf{G}_5] = \Pr[\mathbf{G}_4].$$

Oracle $\mathsf{Chall}(i, m_0, m_1)$

```
1 :  req i_rev = ⊥   // No rev query yet
2 :  req i* = ⊥   // first and only chall query
3 :  req pk_i ≠ ⊥   // ith node exists
4 :  i* ← i
5 :  A ← Stack()   // Init Stack
6 :  // Store indices from i* to 0 in stack
7 :  while i ≠ 0
8 :     A.push(i)
9 :     i ← par_i
10 :  // While popping from stack
11 :  k ← 0   // Index of nodes at Challenger
12 :  while A.is_not_empty()
13 :     i ← A.pop()
14 :     // Call UpdO or VerifyO based on typ
15 :     if typ_i = "full"
16 :        Ch.UpdO(up_{i-1}, pk_i, π_{i-1})
17 :     else
18 :        Ch.VerifyO(k, pk_i, π_{i-1})
19 :     k ← k + 1
20 :  c* ←$ Ch.Chall(k, m_0, m_1)
21 :  return c*
```

Figure 4.13: The modified oracle in the reduction for the proof of Theorem 3 for winning against a half-forking chosen update and CPA-secure UPKE challenger using an adversary that wins in game $\mathbf{G}_5$.

Analysis of Game $\mathbf{G}_5$: Finally, if an adversary can beat $\mathbf{G}_5$, then it can beat the HF-IND-CU-CPA-PV game of $\mathsf{HF}.\Pi$ because we can simulate $\mathcal{A}$'s view in $\mathbf{G}_5$ perfectly by interacting with the HF-IND-CU-CPA-PV challenger. This modified challenge oracle for this reduction is shown in Figure 4.13.

From the adversary's view, the UpdO, and $\mathsf{VerifyO}$ oracles are identical to those in $\mathbf{G}_5$. For the challenge oracle, we first need to recreate the node since we do not pass any updates to the challenge in the UpdO oracle. To do this, we create a stack which stores all the indices form the challenge node to the root. Then by popping an index from the stack, we create that node at the challenger. If the popped index is that of a full node, we call $\mathsf{Ch.UpdO}$ with the appropriate arguments and we call $\mathsf{Ch.VerifyO}$ otherwise. This creates the required node at the challenger, albeit with a different index, which we track with k. Hence, Chall oracle in this reduction also behaves identically to the one in $\mathbf{G}_5$.

Lastly, we have the $\mathsf{UpdHRev}$ oracle. The only time we create nodes at the challenger is in the Chall oracle. So the when the $\mathsf{UpdHRev}$ oracle is called, after Chall has already been called, the challenger contains all the nodes from the root to the challenger. The last full node at the challenger is the first full ancestor of the challenge node, so calling $\mathsf{Ch.UpdHRev}$ directly, performs the update and reveal on that node. Hence even $\mathsf{UpdHRev}$ in the reduction behaves identically to the one in $\mathbf{G}_5$. Hence we have

$$\Pr[\mathbf{G}_5] - 1/2| = \mathsf{Adv}^{\text{HF-IND-CU-CPA-PV}}_{\mathcal{B},\mathsf{HF}.\Pi}(\lambda).$$

Combining all the bounds accumulated during the game hops yields the desired bound in the theorem. □

Chapter 5

Chosen Ciphertext Security

In this chapter we introduce security against chosen ciphertext attacks (CCA security). As a part of the transform for achieving CCA security, we also convert our UPKE to an updatable KEM (UKEM). We introduce the relevant definitions in Section 5.1 and then present our transform along with its proof of security in Section 5.2. The transform we use is an adaptation of the FO transform used by PKEs [HHK17] to the UPKE setting.

5.1 CCA-Secure UKEM

Our next transform lifts CPA security to chosen ciphertext attack (CCA) security. CCA security is an important property for most PKE schemes, including UPKEs, because it rules out a large class of attacks. At the same time, the KEM-DEM paradigm [CS03] has proven to be a useful framework in the public key setting and there exist many efficient and generic transforms that lift a CPA-secure PKE to a CCA-secure KEM [HHK17], which is our approach as well. We provide the syntax for a UKEM with public verifiability, adapted from [AFM24].

Definition 18 (UKEM). *An updatable key encapsulation mechanism (UKEM) with key space $\mathcal{K}$ is a set of six* PPT *algorithms:*

- $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$*: takes as input 1^λ, where λ is the security parameter, and outputs an initial public-private key pair $(\mathsf{pk}_0, \mathsf{sk}_0)$.*
- $(K, c) \leftarrow\!\!\$\ \mathsf{Encaps}(\mathsf{pk}_i)$*: takes as input a public key pk_i, and outputs a key $K \in \mathcal{K}$ and a ciphertext c.*

– $K/\perp \leftarrow \mathsf{Decaps}(\mathsf{sk}_i, \mathsf{pk}_i, c)$: *takes as input a secret key* sk_i *and a ciphertext* c*, and outputs a key* $K \in \mathcal{K}$ *or* $\perp$ *on failure.*

– $(\mathsf{pk}_{i+1}, \mathsf{up}_i, \pi_i) \leftarrow\!\!\$\ \mathsf{UpdPk}(\mathsf{pk}_i)$: *takes as input a public key* pk_i*, and outputs a new public key* pk_{i+1}*, an update token* up_i *and a proof* π_i*.*

– $0/1 \leftarrow \mathsf{Verify}(\mathsf{pk}_i, \mathsf{pk}_{i+1}, \pi_i)$: *takes as input the old public key* pk_i*, the new public key* pk_{i+1}*, the proof* π_i *and outputs 0 or 1.*

– $\mathsf{sk}_{i+1}/\perp \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_i, \mathsf{pk}_{i+1})$: *takes as input a secret key* sk_i*, public key* pk_i*, an update token* up_i*, and the new public key* pk_{i+1}*, and outputs a new secret key* sk_{i+1} *or* $\perp$*.*

For correctness, we require that for any $(\mathsf{pk}_0, \mathsf{sk}_0) \in \mathsf{KGen}(1^\lambda)$*,* $i \in \{0, 1, 2, \dots\}$*,* $(\mathsf{pk}_{j+1}, \mathsf{up}_j) \leftarrow\!\!\$\ \mathsf{UpdPk}(\mathsf{pk}_j)$ *and* $\mathsf{sk}_{j+1} \leftarrow\!\!\$\ \mathsf{UpdSk}(\mathsf{sk}_j, \mathsf{pk}_j, \mathsf{up}_j, \mathsf{pk}_{j+1})$ *for all* $0 \le j \le i$*, we have*

$$\Pr\left[\mathsf{Decaps}(\mathsf{sk}_{i+1}, \mathsf{pk}_{i+1}, c) = K; (K, c) \leftarrow\!\!\$\ \mathsf{Encaps}(\mathsf{pk}_{i+1})\right] \ge 1 - \mathsf{negl}(\lambda)$$

and

$$\Pr\left[\mathsf{Verify}(\mathsf{pk}_i, \mathsf{pk}_{i+1}, \pi_i) = 1\right] = 1$$

where the probability is taken over the randomness of Encaps*.*

The security game for our CCA-secure UKEM remains largely the same as that of our F-IND-CU-CPA-PV-secure UPKE, with the update oracle UpdO, the verify oracle VerifyO, and the honest-update-and-reveal oracle UpdHRev remaining unchanged. Like in the CCA-security game for KEMs, we introduce a new decapsulation oracle DecO that accepts a ciphertext and outputs the shared key or failure. Unlike the game for KEMs, DecO additionally accepts an index i to specify the node and the public key for the query. To avoid trivial wins, we disallow the adversary from asking for a decapsulation query for the challenge ciphertext on a node with the same public key as the challenge node. The challenge oracle Chall now only accepts an index i and outputs the challenge ciphertext along with the challenge key, exactly like the challenge oracle in CCA-security for KEMs. The game is described in Figure 5.1.

Definition 19 (F-IND-CU-CCA-PV security of UKEM)**.** *A UKEM satisfies indistinguishability under forking, chosen update, chosen ciphertext attacks, with public verifiability, if for any PPT* $\mathcal{A}$*,* $\mathsf{Adv}^{\text{F-IND-CU-CCA-PV}}_{\mathcal{A},\Pi}(\lambda) = \left|\Pr[\text{F-IND-CU-CCA-PV}^{\mathcal{A}}_{\Pi}(1^\lambda) = 1] - \frac{1}{2}\right| \le \mathsf{negl}(\lambda)$ *where the* F-IND-CU-CCA-PV *game is defined in Figure 5.1.*

UKEM F-IND-CU-CCA-PV$^{\mathcal{A}}_{\Pi}(1^\lambda)$

1 : $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\$\ \mathsf{KGen}(1^\lambda)$
2 : $(par_0, j, typ_j) \leftarrow (\epsilon, 0, \text{“}full\text{”})$
3 : $(i^*, c^*) \leftarrow (\bot, \bot)$
4 : $b \leftarrow\!\$\ \{0,1\}$
5 : $b' \leftarrow\!\$\ \mathcal{A}^{\mathsf{UpdO},\mathsf{DecO},\mathsf{VerifyO},\mathsf{Chall},\mathsf{UpdHRev}}(\mathsf{pk}_0)$
6 : **return** $b = b'$

Oracle $\mathsf{UpdO}(i, \mathsf{up}, \mathsf{pk}', \pi')$

1 : **req** $i_{rev} = \bot$ // No rev query yet
2 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
3 : **req** $typ_i = \text{“}full\text{”}$ // ith node is full
4 : **req** $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
5 : $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}, \mathsf{pk}')$
6 : **req** $\mathsf{sk}_{j+1} \neq \bot$ // invalid token check
7 : $(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow (\mathsf{up}, \mathsf{pk}', \pi')$
8 : $j \leftarrow j + 1$
9 : $par_j \leftarrow i$
10 : $typ_j \leftarrow \text{“}full\text{”}$

Oracle $\mathsf{DecO}(i, c)$

1 : **req** $i_{rev} = \bot$ // No rev query yet
2 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
3 : **req** $typ_i = \text{“}full\text{”}$ // ith node is full
4 : **req** $(\mathsf{pk}_i, c) \neq (\mathsf{pk}_{i^*}, c^*)$
5 : $\mathsf{K} \leftarrow \mathsf{Decaps}(\mathsf{sk}_i, \mathsf{pk}_i, c)$
6 : **return** K

Oracle $\mathsf{VerifyO}(i, \mathsf{pk}', \pi')$

1 : **req** $i_{rev} = \bot$ // No rev query yet
2 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
3 : **req** $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
4 : $(\mathsf{pk}_{j+1}, \pi_j, \mathsf{sk}_{j+1}) \leftarrow (\mathsf{pk}', \pi', \bot)$
5 : $j \leftarrow j + 1$
6 : $par_j \leftarrow i$
7 : $typ_j \leftarrow \text{“}half\text{”}$

Oracle $\mathsf{Chall}(i)$

1 : **req** $i_{rev} = \bot$ // No rev query yet
2 : **req** $i^* = \bot$ // first and only query
3 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
4 : $i^* \leftarrow i$; $\mathsf{K}^0 \leftarrow\!\$\ \mathcal{K}$
5 : $(\mathsf{K}^1, c^*) \leftarrow\!\$\ \mathsf{Encaps}(\mathsf{pk}_i)$
6 : **return** (K^b, c^*)

Oracle $\mathsf{UpdHRev}(i)$

1 : **req** $i_{rev} = \bot$ // First and only query
2 : **req** $i^* \neq \bot$ // Only after chall query
3 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
4 : **req** $typ_i = \text{“}full\text{”}$ // ith node is full
5 : $(\mathsf{pk}_{j+1}, \mathsf{up}_j, \pi_j) \leftarrow\!\$\ \mathsf{UpdPk}(pk_i)$
6 : **req** $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}_{j+1}, \pi_j)$
7 : $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_j, \mathsf{pk}_{j+1})$
8 : $j \leftarrow j + 1$
9 : $i_{rev} = j$
10 : $par_j \leftarrow i$
11 : $typ_j \leftarrow \text{“}full\text{”}$
12 : **return** $(\mathsf{pk}_j, \mathsf{sk}_j, \mathsf{up}_{j-1}, \pi_{j-1})$

Figure 5.1: The F-IND-CU-CCA-PV (forking indistinguishability under chosen updates and chosen ciphertext attacks with public verifiability) game for a UKEM scheme.

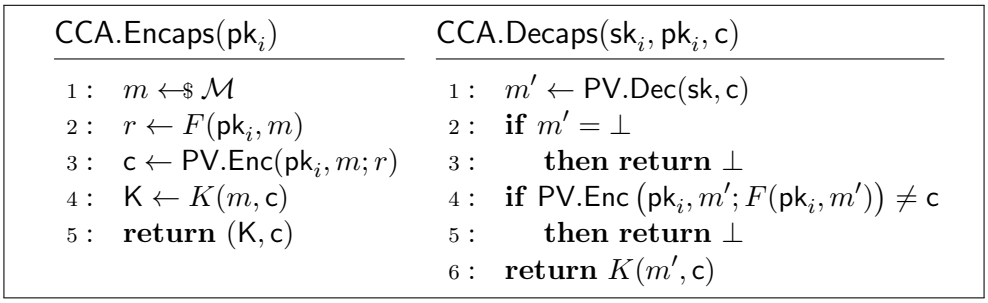


Figure 5.2: The FO transform for lifting a forking chosen update and chosen plaintext attack secure with public verifiability UPKE to a forking chosen update and chosen ciphertext attack with public verifiability secure UKEM .

5.2 Lifting CPA UPKE To CCA UKEM

We can use a modification of [HHK17]'s modular FO transform to obtain a CCA-secure UKEM. There are two potential issues to worry about when introducing CCA security: the adversary using the decryption oracle on update tokens, and dealing with well formed ciphertexts that are a result of shifting an existing ciphertext. Due to the domain separation between the random oracles in our construction, querying the decryption oracle on update tokens should almost always return $\perp$, which takes care of the first issue. To deal the latter issue, we include the public key in the oracle call used to derive the randomness of the ciphertext. With this modification, even if a shifted ciphertext preserves the message and the randomnes, it will not be considered a well formed ciphertext for the new public key. The transform is shown in Figure 5.2.

Theorem 4. *If* $\mathsf{PV}.\Pi$ *is a* F-IND-CU-CPA-PV *UPKE that has* $1 - \mathsf{negl}(\lambda)$ *correctness, is* γ*-spread, and* $F : \{0,1\}^* \mapsto \mathcal{R}_1$ *&* $K : \{0,1\}^* \mapsto \mathcal{K}$ *are modelled as random oracles, then* $\mathsf{CCA}.\Pi$*, the UKEM obtained by applying the FO transform on* $\mathsf{PV}.\Pi$ *as described in Figure 5.2 is a* F-IND-CU-CCA-PV *secure UKEM.*

For any F-IND-CU-CCA-PV *adversary* $\mathcal{A}$ *that issues at most* q_{Decaps} *queries to the decapsulation oracle, there exists a* F-IND-CU-CPA-PV *adversary* $\mathcal{B}$ *such that*

$$\mathsf{Adv}^{\text{F-IND-CU-CCA-PV}}_{\mathcal{A},\mathsf{CCA}.\Pi}(\lambda) \leq q_{\mathsf{Decaps}} \cdot 2^{-\gamma} + \mathsf{Adv}^{\text{F-IND-CU-CPA-PV}}_{\mathcal{B},\mathsf{PV}.\Pi}(\lambda) + \mathsf{negl}(\lambda).$$

We start by giving an outline of the proof. The proof here is almost exactly the same as the one used for ordinary PKEs. The main idea is that we can simulate the decapsulation oracle just by observing the random oracle calls made by the adversary. The only way

an adversary can create a well-formed ciphertext is by querying the ROs to create it, and hence, the reduction can see what was encrypted in each well-formed ciphertext.

For the games, we start with the unmodified game $\mathbf{G}_0$. In game $\mathbf{G}_1$, we abort if we find a honestly generated, but faulty ciphertext. In game $\mathbf{G}_2$, we handle the decapsulation queries without the use of the secret key. There is a negligible chance that the adversary generates a valid ciphertext without querying the corresponding ROs. Finally, we show that any adversary that beats $\mathbf{G}_2$, can be used to construct an adversary that beats the F-IND-CU-CPA-PV game for $\mathsf{PV}.\Pi$.

Proof. We now give the complete proof. It proceeds as a sequence of games against an adversary $\mathcal{A}$, which beats the F-IND-CU-CCA-PV game for UKEMs. The proof proceeds as a sequence of games, and let $\Pr[\mathbf{G}_i]$ be the probability that the adversary wins the i^{th} game.

Game $\mathbf{G}_0$: The F-IND-CU-CCA-PV game for UKEMs against $\mathcal{A}$. Everything proceeds normally. This game is shown in Figure 5.3.

By definition, we have:

$$|\Pr[\mathbf{G}_0] - 1/2| = \mathsf{Adv}^{\text{F-IND-CU-CCA-PV}}_{\mathcal{A},\mathsf{CCA}.\Pi}(\lambda).$$

Game $\mathbf{G}_1$: Same as $\mathbf{G}_0$, except we abort the game if the adversary had submitted an honestly generated faulty KEM ciphertext at any point in the game. Since the base scheme has $1 - \mathsf{negl}(\lambda)$ correctness, there is a chance that a ciphertext outputs a faulty payload upon decryption. Such an event can result in a honestly generated KEM ciphertext, created by querying the corresponding ROs, failing the re-encryption check, and being rejected. We check if we encountered such ciphertexts at the end of the game. The modified main function of the game is shown in Figure 5.4.

The probability of encountering such a faulty KEM ciphertext is directly tied to the decryption failure probability of underlying PKE. If the adversary makes q_{Decaps} queries to the decapsulation oracle, then we get:

$$|\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_0]| \leq q_{\mathsf{Decaps}} \cdot \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$$

With this change, we can condition on the adversary never submitting such a faulty KEM ciphertext in the remaining games.

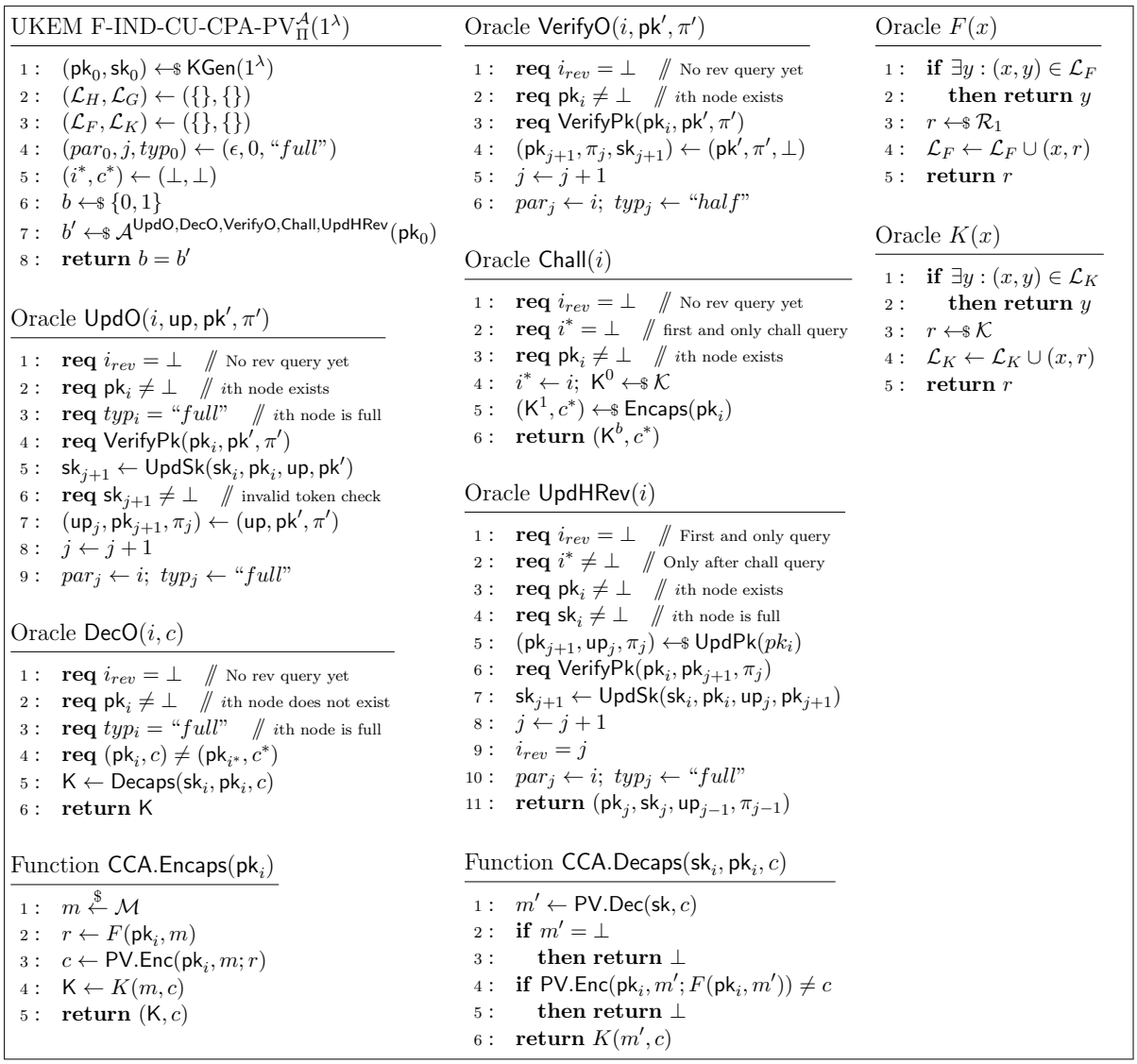


Figure 5.3: The game $\mathbf{G}_0$ for the proof of Theorem 4 for lifting forking chosen update and CPA-secure UPKE with public verifiability to a forking chosen update CCA-secure one with public verifiability.

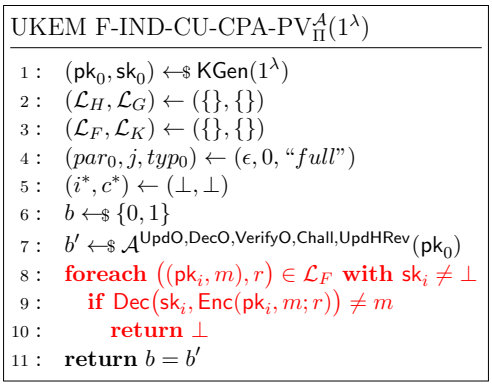


Figure 5.4: The modified game $\mathbf{G}_1$ for the proof of Theorem 4 for lifting forking chosen update and CPA-secure UPKE with public verifiability to a forking chosen update CCA-secure one with public verifiability.

Game $\mathbf{G}_2$: Same as the previous game, but we handle the decapsulation queries without the use of the secret key. To do this, we will look at the random oracle queries made by the adversary. Whenever the adversary queries the decapsulation oracle with the public key pk_i on c', we will look at the record of queries on the F oracle to find an m such that $c' = \mathsf{Enc}(\mathsf{pk}_i, \mathsf{m}; F(\mathsf{pk}_i, \mathsf{m}))$. Since we condition on the fact that honestly generated ciphertexts submitted by the adversarya always pass the re-encryption check, we can answer the decapsulation query on c' by returning $K(\mathsf{m}, c')$. If we do not find such an m, we return $\bot$. The modified oracles are shown in Figure 5.5.

This game continues identically to the previous one, until the event where the adversary accidentally submits a valid ciphertext without querying the corresponding oracles first. Focusing on F, the random oracle that outputs the ciphertext randomness, we know that its output is uniformly random till it is queried. For the adversary to submit a valid ciphertext without querying it, it must correctly guess the output of F, which is the randomness of the ciphertext. Since the scheme is γ-spread, the probability of a successful guess is at most $2^{-\gamma}$. To see why, recall that γ-spreadness implies that, for any public key pk, any message m, and any ciphertext c, $\Pr_{r \leftarrow\$ \mathcal{R}_1}[\mathsf{Enc}(\mathsf{pk}, m; r) = c] \leq 2^{-\gamma}$, where r is chosen uniformly at random and independently of (pk, m, c). Setting r to be the output of the underlying F query, if the query has never been made before, then r will indeed be both uniformly random and independent of (pk, m, c). Thus, the probability of successfully guessing the resulting ciphertext is at most $2^{-\gamma}$. If the adversary makes q_{Decaps} queries to the decapsulation oracle, then by the union bound, we have:

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq q_{\mathsf{Decaps}} \cdot 2^{-\gamma}.$$

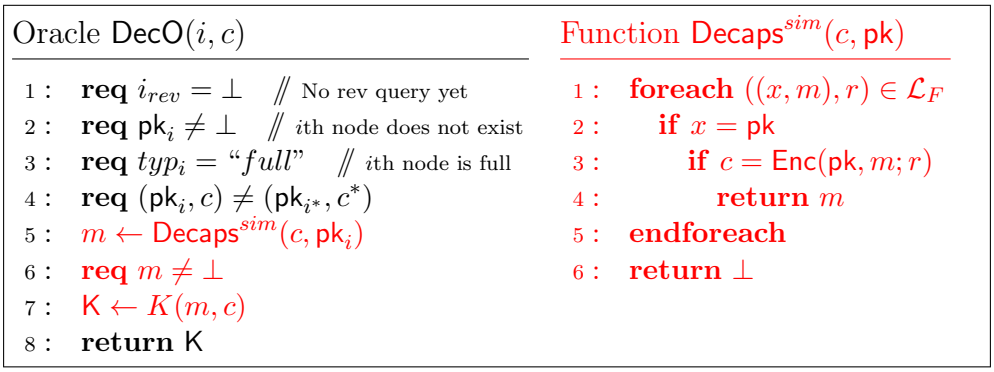


Figure 5.5: Modified oracles in $\mathbf{G}_2$ for the proof of Theorem 4 for lifting forking chosen update and CPA-secure UPKE with public verifiability to a forking chosen update CCA-secure one with public verifiability.

Analysis of Game $\mathbf{G}_2$: Finally, we show that if an adversary can beat $\mathbf{G}_2$, it can be used by another adversary $\mathcal{B}$ to win the F-IND-CU-CPA-PV game against $\mathsf{PV}.\Pi$. The modified challenge oracle in the construction of $\mathcal{B}$ is shown in Figure 5.6.

Like the previous game, we handle the Decapsulation queries ourselves using the random oracles. The update oracle UpdO, verify oracle $\mathsf{VerifyO}$ and honest-update-then-reveal oracle $\mathsf{UpdHRev}$ are directly handled by the challenger. The only non-straightforward part is the challenge query and how we use the adversary's response to win our own game.

We generate the challenge query using the challenger. When the adversary asks for a challenge query, we select two messages m_0, m_1 that the adversary has not queried the random oracles on before. We send the challenge query on the same node to the challenger and get c^* in return. We then sample a randomly uniform K^* and send (K^*, c^*) to $\mathcal{A}$.

For the adversary to determine if K^* is sampled correctly or is uniformly random, it must query the oracle K on $(\mathsf{Dec}(c^*), c^*)$. Similarly, it cannot know that the ciphertext is formed incorrectly unless it queries the F oracle on $\mathsf{Dec}(c^*)$. So we watch the random oracle queries to spot m_b and output b.

Note that $\mathcal{B}$ perfectly simulates $\mathbf{G}_2$ for $\mathcal{A}$ till it makes the aforementioned RO queries, so it must make either of those queries. Since $\mathcal{B}$ cannot properly simulate the response to those queries, $\mathcal{A}$'s behaviour after that is unpredicatble, and we ignore it. Once $\mathcal{A}$ outputs its guess bit and terminates, we ignore its output and look through the RO queries it made to search for m_b and output that. Using this strategy, $\mathcal{B}$ wins the game against the challenger with the same probability that the adversary makes those queries, which is the

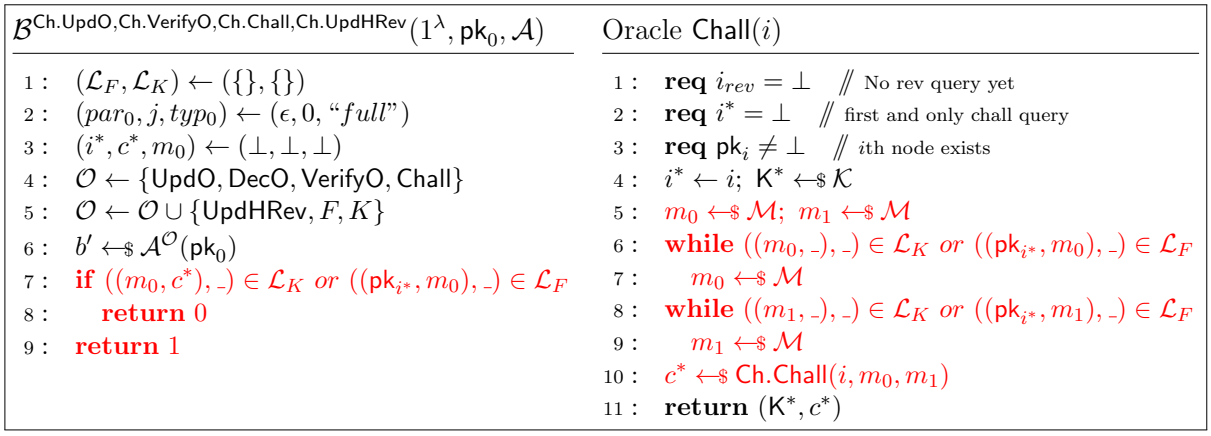


Figure 5.6: Modified oracles in the construction of $\mathcal{B}$ that uses F-IND-CU-CCA-PV adversary $\mathcal{A}$ to win the F-IND-CU-CPA-PV game for the proof of Theorem 4.

same probability with which it wins $\mathbf{G}_2$.

$$|\Pr[\mathbf{G}_2] - 1/2| = \mathsf{Adv}^{\text{F-IND-CU-CPA-PV}}_{\mathcal{B},\text{PV}.\Pi}(\lambda).$$

Combining all the bounds accumulated during the game hops yields the desired bound in the theorem. □

Chapter 6

Corruptions

The last property we add is security against corruptions, which is a generalization of the UpdHRev query. Instead of a single honest-update-and-reveal query, we will allow the adversary to make many independent honest update queries and many reveal (also called corruption) queries. We need to have some restrictions on which nodes the adversary can corrupt to avoid trivial wins. We will refer to the set of nodes that the adversary cannot corrupt as the *challenge set*. The challenge set for a game with the challenge node i^* consists of:

1. All nodes i' on the path from node 0 to node i^* including both of them.
2. All nodes i' such that there exists i in the set with $\mathsf{pk}_{i'} = \mathsf{pk}_i$.
3. All descendants of the nodes in the set created from adversarial updates

We include the nodes in point 1 because if the adversary learns any of the secret keys on the path, it can use the update tokens it has seen throughout the game to update that secret key to the challenge secret key. After including the nodes from point 1, we must include nodes from point 2 since forking updates allow the adversary to re-use update tokens and proofs to create nodes that are identical to nodes in the challenge set in all aspects except their index at the challenger. The nodes from the point 3 in the above description are included because generally, we can assume that if an update was generated adversarially, then the adversary knows the shift value used to generate it. Knowing the shift value could allow the adversary to roll back the update and learn the parent node's secret key.

Corruption F-IND-CU-CCA-PV$^{\mathcal{A}}_{\Pi}(1^\lambda)$

$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\$\ \mathsf{KGen}(1^\lambda)$
$(par_0, rev_0, j) \leftarrow (\epsilon, 0, 0)$
$(typ_0, upd_0) \leftarrow (\text{“}full\text{”}, \epsilon)$
$(i^*, c^*) \leftarrow (\perp, \perp)$
$b \leftarrow\!\$\ \{0,1\}$
$\mathcal{O} \leftarrow \{\mathsf{UpdHO}, \mathsf{UpdO}, \mathsf{DecO}, \}$
$\mathcal{O} \leftarrow \mathcal{O} \cup \{\mathsf{RevO}, \mathsf{VerifyO}, \mathsf{Chall}\}$
$b' \leftarrow\!\$\ \mathcal{A}^{\mathcal{O}}(\mathsf{pk}_0)$
$S \leftarrow$ **chall-set**(i^*)
if not$(\forall j \in S : \neg rev_j)$
  then $x \leftarrow\!\$\ \{0,1\}$; **return** x
return $(b = b')$

Oracle $\mathsf{Chall}(i)$

req $i^* = \perp$ // first and only chall query
req $\mathsf{pk}_i \neq \perp$ // ith node exists
$i^* \leftarrow i$; $\mathsf{K}^0 \leftarrow\!\$\ \mathcal{K}$
$(\mathsf{K}^1, c^*) \leftarrow\!\$\ \mathsf{Encaps}(\mathsf{pk}_i)$
return (K^b, c^*)

Oracle $\mathsf{RevO}(\mathrm{i})$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
$rev_i \leftarrow 1$
return sk_i

Oracle $\mathsf{VerifyO}(i, \mathsf{pk}', \pi')$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
$(\mathsf{pk}_{j+1}, \pi_j, \mathsf{sk}_{j+1}) \leftarrow (\mathsf{pk}', \pi', \perp)$
$j \leftarrow j + 1$
$(par_j, rev_j) \leftarrow (i, 0)$
$(typ_j, upd_j) \leftarrow (\text{“}half\text{”}, \text{“}DisHon\text{”})$

Oracle $\mathsf{UpdHO}(i)$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
$(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow\!\$\ \mathsf{UpdPk}(\mathsf{pk}_i)$
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}_{j+1}, \pi_j)$
$typ_{j+1} = \text{“}half\text{”}$
if $typ_i = \text{“}full\text{”}$
  $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_j, \mathsf{pk}_{j+1})$
  $typ_{j+1} = \text{“}full\text{”}$
$j \leftarrow j + 1$
$(par_j, rev_j, upd_j) \leftarrow (i, 0, \text{“}Hon\text{”})$
return $(\mathsf{pk}_j, \mathsf{up}_{j-1}, \pi_{j-1})$

Oracle $\mathsf{UpdO}(i, \mathsf{up}, \mathsf{pk}', \pi')$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
$\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}, \mathsf{pk}')$
req $\mathsf{sk}_{j+1} \neq \perp$
$(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow (\mathsf{up}, \mathsf{pk}', \pi')$
$j \leftarrow j + 1$
$(par_j, rev_j) \leftarrow (i, 0)$
$(typ_j, upd_j) \leftarrow (\text{“}full\text{”}, \text{“}DisHon\text{”})$
return 1

Oracle $\mathsf{DecO}(i, c)$

req $pk_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
req $(\mathsf{pk}_i, c) \neq (\mathsf{pk}_{i^*}, c^*)$
$\mathsf{K} \leftarrow \mathsf{Decaps}(\mathsf{sk}_i, \mathsf{pk}_i, c)$
if $\mathsf{K} = \perp$ // Decapsulation failure
  then return $\perp$
return K

Figure 6.1: The F-IND-CU-CCA-PV game with adaptive corruptions for an UKEM.

Figure 6.1 shows the game. Notably, this version of the game permits *adaptive* corruptions: the adversary may choose which nodes to corrupt and construct the tree based on the challenger's responses. Our transforms do not achieve full adaptive security, so we introduce a weaker notion that restricts the adversary's freedom by requiring commitment to parts of its strategy before seeing any responses, analogous to selective security in threshold schemes. We call this security against *selective corruptions*. In the next section, we define this notion. However, before we achieve selective corruption resilience, we define another intermediate security notion called "re-ordered forking security" which makes our proof of security for selective corruptions easier. In Section 6.2 we define this notion, followed by showing how to achieve it in Section 6.3. Finally, in Section 6.4 we show how to achieve selective corruptions security.

6.1 Selective Corruptions

The rationale behind this definition of selective corruptions is to have the adversary commit to as few details as possible, while still enabling us to achieve a construction. In our proof, our reduction technique requires that we know the challenge set in advance, before the adversary makes any oracle calls. Keeping this in mind, the least information that the adversary can commit to is the tree structure (in the form of all the par_j variables) and the challenge-set itself (as a set of indices). This commitment fixes the indices for each node and, therefore, places a partial restriction on the ordering of the oracle calls since the nodes are indexed by the order in which they are created. With this in mind, we can define the selective security game, which is a simple modification of the adaptive corruptions game as just described; the game is shown in Figure 6.2.

Definition 20 (Sel-F-IND-CU-CCA-PV security of UKEM)**.** *A UKEM satisfies indistinguishability under forking, chosen update, chosen ciphertext attacks, with public verifiability and selective corruptions, if for any PPT* $\mathcal{A}$, $\mathsf{Adv}^{\text{Sel-F-IND-CU-CCA-PV}}_{\mathcal{A},\Pi}(\lambda) = \left|\Pr[\text{Sel-F-IND-CU-CCA-PV}^{\mathcal{A}}_{\Pi}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$ *where the* Sel-F-IND-CU-CCA-PV *game is defined in Figure 6.2.*

UKEM Sel-F-IND-CU-CCA-PV$^{\mathcal{A}}_{\Pi}(1^\lambda)$

$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\$\ \mathsf{KGen}(1^\lambda)$
$(par_0, rev_0, j) \leftarrow (\epsilon, 0, 0)$
$(typ_0, upd_0) \leftarrow (\text{“}full\text{”}, \epsilon)$
$(i^*, c^*) \leftarrow (\perp, \perp)$
$b \leftarrow\!\$\ \{0,1\}$
$(\{par'_i\}_{\forall i}, S') \leftarrow\!\$\ \mathcal{A}(\mathsf{pk}_0)$
$\mathcal{O} \leftarrow \{\mathsf{UpdHO}, \mathsf{UpdO}, \mathsf{DecO}\}$
$\mathcal{O} \leftarrow \mathcal{O} \cup \{\mathsf{RevO}, \mathsf{Chall}, \mathsf{VerifyO}\}$
$b' \leftarrow\!\$\ \mathcal{A}^{\mathcal{O}}(\mathsf{pk}_0)$
$S \leftarrow$ **chall-set**(i^*)
if not$\big((S = S') \wedge (par_i = par'_i)_{\forall i}\big)$
  then $x \leftarrow\!\$\ \{0,1\}$; **return** x
return $(b = b')$

Oracle $\mathsf{Chall}(i)$

req $i^* = \perp$ // first and only chall query
req $\mathsf{pk}_i \neq \perp$ // ith node exists
$i^* \leftarrow i$; $\mathsf{K}^0 \leftarrow\!\$\ \mathcal{K}$
$(\mathsf{K}^1, c^*) \leftarrow\!\$\ \mathsf{Encaps}(\mathsf{pk}_i)$
return (K^b, c^*)

Oracle $\mathsf{RevO}(\mathrm{i})$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
$rev_i \leftarrow 1$
return sk_i

Oracle $\mathsf{VerifyO}(i, \mathsf{pk}', \pi')$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
$(\mathsf{pk}_{j+1}, \pi_j, \mathsf{sk}_{j+1}) \leftarrow (\mathsf{pk}', \pi', \perp)$
$j \leftarrow j + 1$
$(par_j, rev_j) \leftarrow (i, 0)$
$(typ_j, upd_j) \leftarrow (\text{“}half\text{”}, \text{“}DisHon\text{”})$

Oracle $\mathsf{UpdHO}(i)$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
$(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow\!\$\ \mathsf{UpdPk}(\mathsf{pk}_i)$
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}_{j+1}, \pi_j)$
$typ_{j+1} \leftarrow typ_i$
if $typ_i = \text{“}full\text{”}$
  $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_j, \mathsf{pk}_{j+1})$
  $typ_{j+1} \leftarrow \text{“}full\text{”}$
$j \leftarrow j + 1$
$(par_j, rev_j, upd_j) \leftarrow (i, 0, \text{“}Hon\text{”})$
return $(\mathsf{pk}_j, \mathsf{up}_{j-1}, \pi_{j-1})$

Oracle $\mathsf{UpdO}(i, \mathsf{up}, \mathsf{pk}', \pi')$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
$\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}, \mathsf{pk}')$
req $\mathsf{sk}_{j+1} \neq \perp$
$(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow (\mathsf{up}, \mathsf{pk}', \pi')$
$j \leftarrow j + 1$
$(par_j, rev_j) \leftarrow (i, 0)$
$(typ_j, upd_j) \leftarrow (\text{“}full\text{”}, \text{“}DisHon\text{”})$
return 1

Oracle $\mathsf{DecO}(i, c)$

req $pk_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
req $(\mathsf{pk}_i, c) \neq (\mathsf{pk}_{i^*}, c^*)$
$\mathsf{K} \leftarrow \mathsf{Decaps}(\mathsf{sk}_i, \mathsf{pk}_i, c)$
if $\mathsf{K} = \perp$ // Decapsulation failure
  then return $\perp$
return K

Figure 6.2: The Sel-F-IND-CU-CCA-PV game for an UKEM.

6.2 An Intermediate Step: F-IND-CU-CCA-PV2 Security ("Reordered Forking")

For our reduction, we cannot directly prove selective security from a F-IND-CU-CCA-PV secure UKEM. Instead, we introduce another intermediate security definition, which is a minor modification of F-IND-CU-CCA-PV security. In this new security goal, which we will call F-IND-CU-CCA-PV2, the adversary is allowed to ask for the "honest-update-and-reveal" query before it has asked the challenge query. This is not a natural definition based on any real adversarial scenario, but it does help make our proof of selective security easier.

In this new notion, we allow the adversary to ask for the honest update and reveal query before it asks for the challenge query. There is just one restriction: the adversary cannot query any oracles on a node that has the same public key as the revealed node. This restriction prevents the adversary from recreating the revealed node and using it as a part of its winning strategy. This security game is a minor modification of the F-IND-CU-CCA-PV game for UKEM, and it is described in Figure 6.3.

Definition 21 (F-IND-CU-CCA-PV2 security of UKEM)**.** *A UKEM satisfies indistinguishability under re-ordered forking, chosen update, chosen ciphertext attacks, with public verifiability, if for any PPT* $\mathcal{A}$, $\mathsf{Adv}^{\text{F-IND-CU-CCA-PV2}}_{\mathcal{A},\Pi}(\lambda) = \left|\Pr[\text{F-IND-CU-CCA-PV2}^{\mathcal{A}}_{\Pi}(1^\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$ *where the* F-IND-CU-CCA-PV2 *game is defined in Figure 6.3.*

6.3 Lifting CCA UKEM To Reordered Forking Security

To lift our UKEM from full-forking security (F-IND-CU-CCA-PV) to reordered-forking security (F-IND-CU-CCA-PV2), we need to only assume that it has ciphertext updatability and update uniformity, which in fact is already assumed in our previous transformations.

Theorem 5. *Let* $\mathsf{CCA}.\Pi$ *be a* F-IND-CU-CCA-PV *secure UKEM obtained by applying the transforms described in Figure 3.4, Figure 4.2, and Figure 5.2 to a* IND-CR-CPA *UPKE* $\mathsf{CR}.\Pi$*, with* $1 - \mathsf{negl}(\lambda)$ *correctness, message space* $\mathcal{M}$*, and* $H : \{0,1\}^* \mapsto \mathcal{R}_2$*,* $G : \{0,1\}^* \mapsto \mathcal{R}_1$*,* $F : \{0,1\}^* \mapsto \mathcal{R}_1$*, and* $K : \{0,1\}^* \mapsto \mathcal{K}$ *modelled as random oracles, and the associated NIZK-PoK being* ϵ_{ZK}*-zero knowledge and* ϵ_{Ext}*-straightline simulation extractable. Then if* $\mathsf{CR}.\Pi$ *has ciphertext updatability and update uniformity, and* $\mathsf{CR}.\Pi$ *is* γ*-spread, then* $\mathsf{CCA}.\Pi$ *is* F-IND-CU-CCA-PV2 *secure.*

UKEM F-IND-CU-CCA-PV2$^{\mathcal{A}}_{\Pi}(1^{\lambda})$

$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\$\ \mathsf{KGen}(1^{\lambda})$
$(par_0, j, typ_j) \leftarrow (\epsilon, 0, \text{“}full\text{”})$
$(i^*, c^*) \leftarrow (\perp, \perp)$
$b \leftarrow\!\$\ \{0,1\}$
$b' \leftarrow\!\$\ \mathcal{A}^{\mathsf{UpdO},\mathsf{DecO},\mathsf{VerifyO},\mathsf{Chall},\mathsf{UpdHRev}}(\mathsf{pk}_0)$
return $b = b'$

Oracle $\mathsf{UpdO}(i, \mathsf{up}, \mathsf{pk}', \pi')$

req $\mathsf{pk}_i \neq \mathsf{pk}_{i_{rev}}$ // No query on i_{rev}
req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
$\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}, \mathsf{pk}')$
req $\mathsf{sk}_{j+1} \neq \perp$ // invalid token check
$(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow (\mathsf{up}, \mathsf{pk}', \pi')$
$j \leftarrow j + 1$
$par_j \leftarrow i$
$typ_j \leftarrow \text{“}full\text{”}$

Oracle $\mathsf{DecO}(i, c)$

req $\mathsf{pk}_i \neq \mathsf{pk}_{i_{rev}}$ // No query on i_{rev}
req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
req $(\mathsf{pk}_i, c) \neq (\mathsf{pk}_{i^*}, c^*)$
$\mathsf{K} \leftarrow \mathsf{Decaps}(\mathsf{sk}_i, \mathsf{pk}_i, c)$
return K

Oracle $\mathsf{VerifyO}(i, \mathsf{pk}', \pi')$

req $\mathsf{pk}_i \neq \mathsf{pk}_{i_{rev}}$ // No query on i_{rev}
req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
$(\mathsf{pk}_{j+1}, \pi_j, \mathsf{sk}_{j+1}) \leftarrow (\mathsf{pk}', \pi', \perp)$
$j \leftarrow j + 1$
$par_j \leftarrow i$
$typ_j \leftarrow \text{“}half\text{”}$

Oracle $\mathsf{Chall}(i)$

req $\mathsf{pk}_i \neq \mathsf{pk}_{i_{rev}}$ // No query on i_{rev}
req $i^* = \perp$ // first and only chall query
req $\mathsf{pk}_i \neq \perp$ // ith node exists
$i^* \leftarrow i$; $\mathsf{K}^0 \leftarrow\!\$\ \mathcal{K}$
$(\mathsf{K}^1, c^*) \leftarrow\!\$\ \mathsf{Encaps}(\mathsf{pk}^*_i)$
return (K^b, c^*)

Oracle $\mathsf{UpdHRev}(i)$

req $i_{rev} = \perp$ // First and only query
req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
$(\mathsf{pk}_{j+1}, \mathsf{up}_j, \pi_j) \leftarrow\!\$\ \mathsf{UpdPk}(pk_i)$
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}_{j+1}, \pi_j)$
$\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_j, \mathsf{pk}_{j+1})$
$j \leftarrow j + 1$
$i_{rev} = j$
$par_j \leftarrow i$
$typ_j \leftarrow \text{“}full\text{”}$
return $(\mathsf{pk}_j, \mathsf{sk}_j, \mathsf{up}_{j-1}, \pi_{j-1})$

Figure 6.3: The re-ordered F-IND-CU-CCA-PV game for an UKEM.

For any successful F-IND-CU-CCA-PV2 *adversary* $\mathcal{A}$ *against* $\mathsf{CCA}.\Pi$ *that issues at most* q_{up} *queries to the* UpdO *oracle,* q_{Decaps} *queries to the* DecO *oracle,* q_{VerifyO} *queries to the* $\mathsf{VerifyO}$ *oracle,* q_H *queries to* H*,* q_K *queries to* K*, and* q_F *queries to* F*, there exists a* F-IND-CU-CCA-PV *adversary* $\mathcal{B}$ *and* OW-CR-CPA *adversaries* $\mathcal{B}', \mathcal{B}''$ *such that*

$$\begin{aligned}\mathsf{Adv}^{\text{F-IND-CU-CCA-PV2}}_{\mathcal{A},\mathsf{CCA}.\Pi}(\lambda) \leq\ & (q_{\mathsf{up}} + q_{\mathsf{Decaps}}) \cdot 2^{-\gamma} + \epsilon_{\mathsf{ZK}} + \mathsf{negl}(\lambda) + q_K/\mathcal{M} \\ & + (q_{\mathsf{up}} + q_{\mathsf{VerifyO}}) \cdot \epsilon_{\mathsf{Ext}} + q_H \cdot \mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B}',\mathsf{CR}.\Pi}(\lambda) \\ & + q_F \cdot \mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B}'',\mathsf{CR}.\Pi}(\lambda) + \mathsf{Adv}^{\text{F-IND-CU-CCA-PV}}_{\mathcal{B},\mathsf{CCA}.\Pi}(\lambda).\end{aligned}$$

We start by giving a **proof outline**. The F-IND-CU-CCA-PV2 adversary could potentially want to ask for the $\mathsf{UpdHRev}$ query before the challenge query, but the challenger does not allow this. The interesting point here is that the challenge query needs no input beyond specifying the node on which it is asked. So the main idea will be to immediately ask for the challenge query and the $\mathsf{UpdHRev}$ query at the root node during initialization, and then handle all queries made by the adversary ourselves. Then we can forge the update for the $\mathsf{UpdHRev}$ query that the adversary asks for, and update the ciphertext to the correct node when the adversary asks for the challenge query. One thing to note is that when updating ciphertexts, we have no control over the randomness of the resulting ciphertext, so we need to specify abort conditions to ensure the adversary does not realize the ciphertext is malformed.

For the games, we start with the unmodified game $\mathbf{G}_0$. In game $\mathbf{G}_1$, we abort if we find a honestly generated, but faulty update token. In game $\mathbf{G}_2$, we handle the rejection of malformed update queries without the knowledge of the secret key, using the random oracles. In game $\mathbf{G}_3$, we abort if we find a honestly generated, but faulty ciphertext. In $\mathbf{G}_4$, we handle decapsulation queries without knowledge of the secret, again using random oracles. In $\mathbf{G}_5$, we begin forging the update in the $\mathsf{UpdHRev}$ query by replacing the proof with one generated by the simulator. In $\mathbf{G}_6$, we continue this forgery by replacing the update token with a forged one. In $\mathbf{G}_7$, we finish this forgery by replacing the correct keypair with one obtained by updating the root node.

In $\mathbf{G}_8$, when the adversary asks for the challenge query on node i, we extract all the shift values on the path from i to the root node. In $\mathbf{G}_9$, we add an abort condition when oracle F is queried on the payload in the challenge ciphertext. In $\mathbf{G}_{10}$, we replace the abort condition to instead trigger when any message that re-encrypts to the challenge ciphertext is queried. In $\mathbf{G}_{11}$, we add a similar abort condition in the oracle K to abort when queried on m^*. In $\mathbf{G}_{12}$ we replace that condition with a more general one as we did in $\mathbf{G}_{10}$. Finally, we show that if an adversary $\mathcal{A}$ can beat game $\mathbf{G}_{12}$, then it can be used to beat the F-IND-CU-CCA-PV game against $\mathsf{CCA}.\Pi$.

Proof. We will now give the **full proof**. It proceeds as a sequence of games against an adversary $\mathcal{A}$ for the F-IND-CU-CCA-PV2 security of $\mathsf{CCA}.\Pi$. Let $\Pr[\mathbf{G}_i]$ be the probability that the adversary wins the i^{th} game.

Game $\mathbf{G}_0$: The F-IND-CU-CCA-PV2 game against $\mathsf{CCA}.\Pi$. Everything proceeds as normal. In addition to the oracles in the game, $\mathcal{A}$ also has access to the random oracles H, G, F, and K. This game is shown in Figure 6.4.

By definition, we have:

$$|\Pr[\mathbf{G}_0] - 1/2| = \mathsf{Adv}^{\text{F-IND-CU-CCA-PV2}}_{\mathcal{A},\mathsf{CCA}.\Pi}(\lambda).$$

Game $\mathbf{G}_1$: Same as $\mathbf{G}_0$, except we abort the game if the adversary had submitted an honestly generated faulty update token at any point in the game. Since the base scheme has $1 - \mathsf{negl}(\lambda)$ correctness, there is a chance that a ciphertext outputs a faulty payload upon decryption. Such an event can result in a honestly generated update token, created by querying the corresponding ROs, failing the re-encryption check, and being rejected. We check if we encountered such tokens at the end of the game. The modified main function of the game is shown in Figure 6.5.

The probability of encountering such a faulty update token is directly tied to the decryption failure probability of underlying PKE. If the adversary makes q_{up} queries to the update oracle, then we get:

$$|\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_0]| \leq q_{\mathsf{up}} \cdot \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$$

With this change, we can condition on the adversary never submitting such a faulty update token in the remaining games.

Game $\mathbf{G}_2$: Same as $\mathbf{G}_1$, except that malformed update queries are rejected without knowledge of the secret key. The challenger can do this by observing the RO calls made by $\mathcal{A}$. If $\mathcal{A}$ calls UpdO on $\mathsf{up}', \mathsf{pk}', \pi'$, then the challenger first verifies π' and if that passes, it looks for $r, s, \mathsf{up}, \mathsf{pk}', h$ such that $\big(s = H(r) \wedge \mathsf{up}, \mathsf{pk}' \leftarrow \mathsf{UpdPk}(\mathsf{pk}; s) \wedge h = G(r, \mathsf{up}, \mathsf{pk}') \wedge \mathsf{up}' \leftarrow \mathsf{Enc}(\mathsf{pk}, r; h)\big)$. If such a tuple can be found, then it is considered a valid update with randomness r for $\mathsf{CR}.\Pi$, otherwise not. The modified oracles are shown in Figure 6.6.

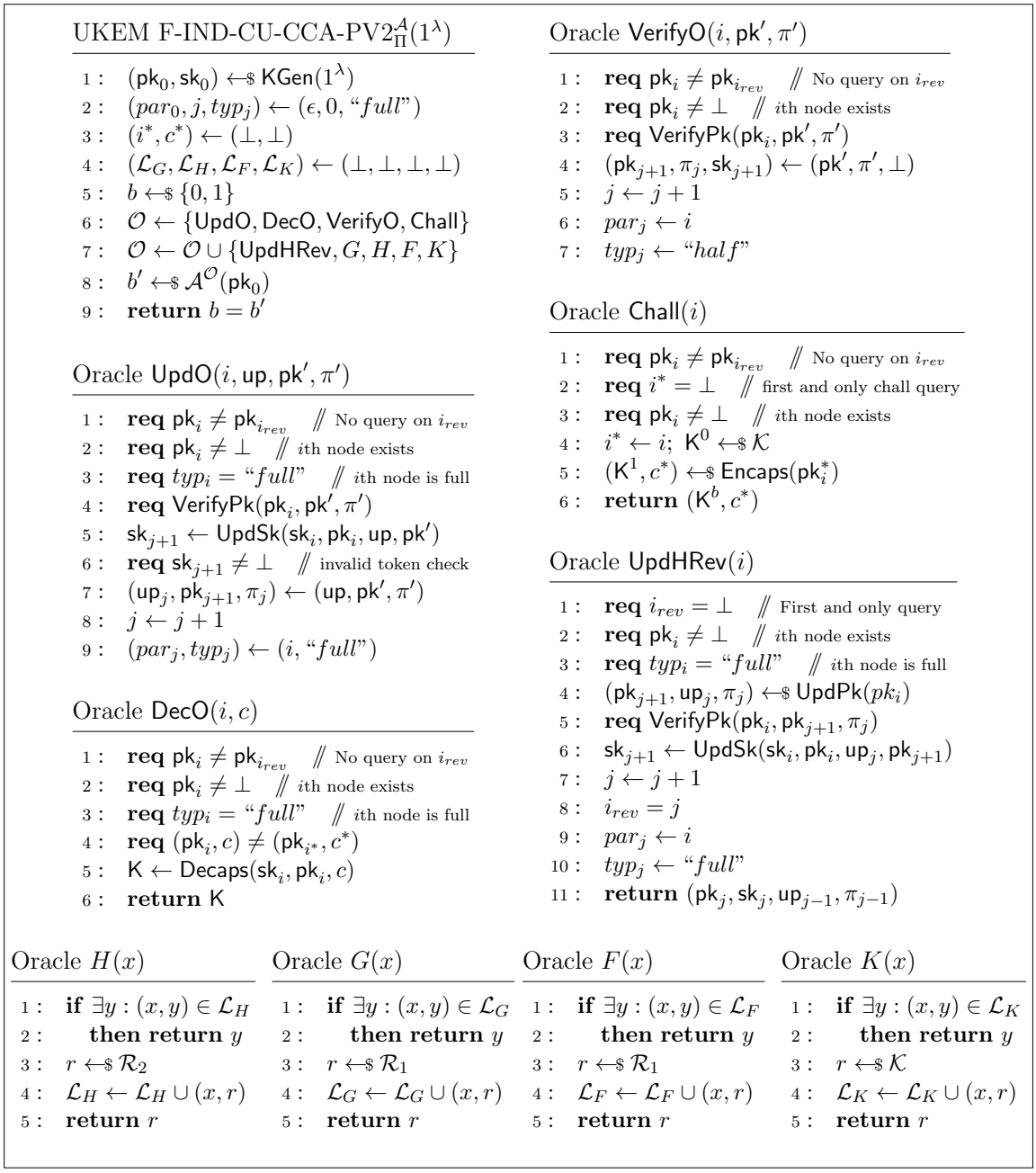


Figure 6.4: The game $\mathbf{G}_0$ for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

UKEM F-IND-CU-CCA-PV2$^{\mathcal{A}}_{\Pi}(1^\lambda)$

$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\$\ \mathsf{KGen}(1^\lambda)$
$(par_0, j, typ_j) \leftarrow (\epsilon, 0, \text{“}full\text{”})$
$(i^*, c^*) \leftarrow (\bot, \bot)$
$(\mathcal{L}_G, \mathcal{L}_H, \mathcal{L}_F, \mathcal{L}_K) \leftarrow (\bot, \bot, \bot, \bot)$
$b \leftarrow\!\$\ \{0,1\}$
$\mathcal{O} \leftarrow \{\mathsf{UpdO}, \mathsf{DecO}, \mathsf{VerifyO}, \mathsf{Chall}\}$
$\mathcal{O} \leftarrow \mathcal{O} \cup \{\mathsf{UpdHRev}, G, H, F, K\}$
$b' \leftarrow\!\$\ \mathcal{A}^{\mathcal{O}}(\mathsf{pk}_0)$
foreach $\big((r, _, _), y\big) \in \mathcal{L}_G$
  foreach $i \in \{0, \ldots, j\}$ **with** $typ_i = \text{“}full\text{”}$
    if $\mathsf{Dec}\big(\mathsf{sk}_i, \mathsf{Enc}(\mathsf{pk}_i, r; y)\big) \neq r$
      return $\bot$
return $b = b'$

Figure 6.5: The modified oracles in $\mathbf{G}_1$ for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

Oracle $\mathsf{UpdO}(i, \mathsf{up}, \mathsf{pk}', \pi')$

req $\mathsf{pk}_i \neq \mathsf{pk}_{i_{rev}}$ // No query on i_{rev}
req $\mathsf{pk}_i \neq \bot$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
$s \leftarrow \mathsf{UpdSk}^{sim}(\mathsf{pk}_i, \mathsf{up}, \mathsf{pk}')$
req $s \neq \bot$ // invalid token check
$\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}, \mathsf{pk}')$
$(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow (\mathsf{up}, \mathsf{pk}', \pi')$
$j \leftarrow j + 1$
$(par_j, typ_j) \leftarrow (i, \text{“}full\text{”})$

Function $\mathsf{UpdSk}^{sim}(\mathsf{pk}, \mathsf{up}', \mathsf{pk}')$

foreach (x, s) **in** $\mathcal{L}_H$
  $\mathsf{CR.up}, \bar{\mathsf{pk}} \leftarrow \mathsf{CR.UpdPk}(\mathsf{pk}; s)$
  if $\bar{\mathsf{pk}} \neq \mathsf{pk}'$
    then continue
  if $\nexists h : ((r, \mathsf{CR.up}, \mathsf{pk}'), h) \in \mathcal{L}_G$
    then continue
  $h \leftarrow G(r, \mathsf{CR.up}, \mathsf{pk}')$
  $\mathsf{CU.up} \leftarrow \mathsf{Enc}(\mathsf{pk}, r; h)$
  if $\mathsf{CU.up} \neq \mathsf{up}'$
    then continue
  return s
endforeach
return $\bot$

Figure 6.6: The modified oracles in $\mathbf{G}_2$ for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

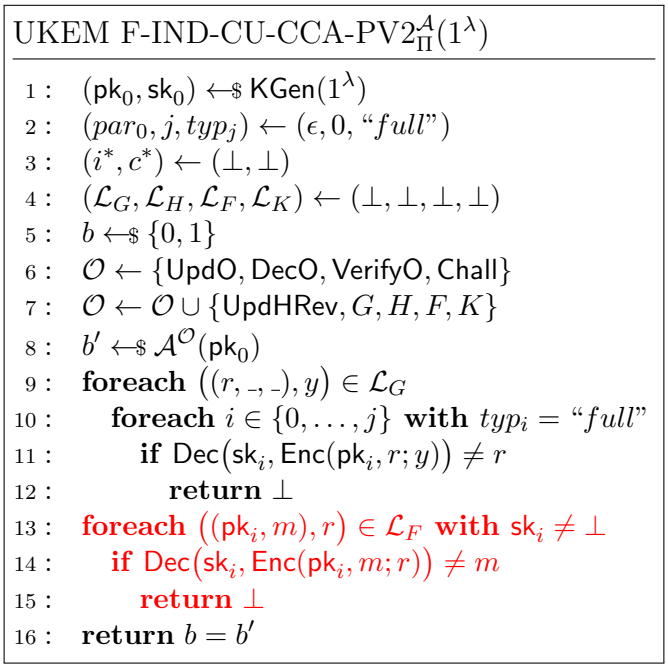


Figure 6.7: The modified oracles in $\mathbf{G}_3$ for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

The games continue identically until $\mathcal{A}$ produces an update token that is valid without first querying the corresponding ROs. This occurs with probability $2^{-\gamma}$ due to the γ-spread property of $\mathsf{CR}.\Pi$, as we have shown before in Section 3.4. If the adversary makes q_{up} queries to the update oracle, then by the union bound, we have:

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq q_{\mathsf{up}} \cdot 2^{-\gamma}.$$

Game $\mathbf{G}_3$: Same as $\mathbf{G}_2$, except we abort the game if the adversary had submitted an honestly generated faulty KEM ciphertext at any point in the game. Since the base scheme has $1 - \mathsf{negl}(\lambda)$ correctness, there is a chance that a ciphertext outputs a faulty payload upon decryption. Such an event can result in a honestly generated KEM ciphertext, created by querying the corresponding ROs, failing the re-encryption check, and being rejected. We check if we encountered such ciphertexts at the end of the game. The modified main function of the game is shown in Figure 6.7.

The probability of encountering such a faulty KEM ciphertext is directly tied to the decryption failure probability of underlying PKE. If the adversary makes q_{Decaps} queries to

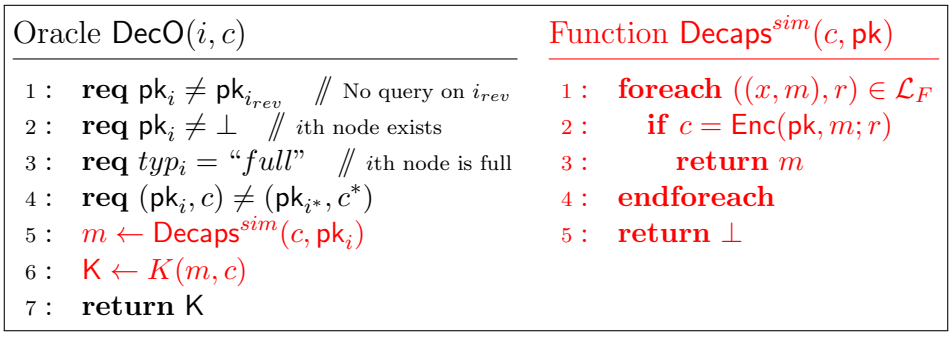


Figure 6.8: The modified oracles in $\mathbf{G}_4$ for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

the decapsulation oracle, then we get:

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq q_{\mathsf{Decaps}} \cdot \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$$

With this change, we can condition on the adversary never submitting such a faulty KEM ciphertext in the remaining games.

Game $\mathbf{G}_4$: Same as $\mathbf{G}_3$, but we handle the decapsulation queries without the use of the secret key. To do this, we will look at the random oracle queries made by the adversary. Whenever the adversary queries the decapsulation oracle with the public key pk_i on c', we will look at the record of queries on the F oracle to find an m such that $c' = \mathsf{Enc}(\mathsf{pk}_i, m; F(\mathsf{pk}_i, m))$. Since we condition on the fact that honestly generated ciphertexts submitted by the adversarya always pass the re-encryption check, we can answer the decapsulation query on c' by returning $K(\mathsf{m}, c')$. If we do not find such an m, we return $\perp$. The modified oracles are shown in Figure 6.8.

This game continues identically to the previous one until the event in which the adversary accidentally submits a valid ciphertext without first querying the corresponding oracles. The probability of this happening is related to the size of the randomness space, and it happens with probability $2^{-\gamma}$ if the scheme is γ-spread, as we have shown before in Section 5.2. If the adversary makes q_{Decaps} queries to the decapsulation oracle, then by the union bound, we have:

$$|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_3]| \leq q_{\mathsf{Decaps}} \cdot 2^{-\gamma}.$$

Game $\mathbf{G}_5$: Same as $\mathbf{G}_4$, except when $\mathcal{A}$ asks for the $\mathsf{UpdHRev}$ query, the associated NIZK-PoK is generated using the simulator. So the response consists of the correctly

Oracle $\mathsf{UpdHRev}(i)$ (in $\mathbf{G}_6$)	Oracle $\mathsf{UpdHRev}(i)$ (in $\mathbf{G}_7$)
1 : **req** $i_{rev} = \bot$ // First and only query	1 : **req** $i_{rev} = \bot$ // First and only query
2 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists	2 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
3 : **req** $typ_i =$ "$full$" // ith node is full	3 : **req** $typ_i =$ "$full$" // ith node is full
4 : $(\mathsf{pk}_{j+1}, \mathsf{up}_j, \pi_j) \leftarrow\$ \mathsf{UpdPk}(pk_i)$	4 : $(\mathsf{pk}_{j+1}, \mathsf{up}_j, \pi_j) \leftarrow\$ \mathsf{UpdPk}(pk_0)$
5 : **req** $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}_{j+1}, \pi_j)$	5 : **req** $\mathsf{VerifyPk}(\mathsf{pk}_0, \mathsf{pk}_{j+1}, \pi_j)$
6 : $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_j, \mathsf{pk}_{j+1})$	6 : $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{up}_j, \mathsf{pk}_{j+1})$
7 : $r_{\mathsf{up}} \leftarrow\$ \mathcal{R}_1$	7 : $r_{\mathsf{up}} \leftarrow\$ \mathcal{R}_1$
8 : $\mathsf{up}_j \leftarrow\$ \mathsf{Enc}(\mathsf{pk}_i, r_{\mathsf{up}})$	8 : $\mathsf{up}_j \leftarrow\$ \mathsf{Enc}(\mathsf{pk}_i, r_{\mathsf{up}})$
9 : $\pi_j \leftarrow\$ \mathsf{Sim}(\mathsf{pk}_i, \mathsf{pk}_{j+1})$	9 : $\pi_j \leftarrow\$ \mathsf{Sim}(\mathsf{pk}_i, \mathsf{pk}_{j+1})$
10 : $j \leftarrow j + 1$	10 : $j \leftarrow j + 1$
11 : $i_{rev} = j$	11 : $i_{rev} = j$
12 : $par_j \leftarrow i$	12 : $par_j \leftarrow i$
13 : $typ_j \leftarrow$ "$full$"	13 : $typ_j \leftarrow$ "$full$"
14 : **return** $(\mathsf{pk}_j, \mathsf{sk}_j, \mathsf{up}_{j-1}, \pi_{j-1})$	14 : **return** $(\mathsf{pk}_j, \mathsf{sk}_j, \mathsf{up}_{j-1}, \pi_{j-1})$

Figure 6.9: The modified oracles in game $\mathbf{G}_6$ (left) and $\mathbf{G}_7$ (right) for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

generated token up, correctly generated keypair $(\mathsf{pk}, \mathsf{sk})$, and the simulated proof π'. This is a minor modification and we omit the pseudocode description.

The games continue identically until this $\mathsf{UpdHRev}$ query, where the simulator is used. The difference in the success probability of the adversary between the two games is tied to the zero-knowledge property of the NIZK-PoK.

$$|\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_4]| \leq \epsilon_{\mathsf{ZK}}.$$

Game $\mathbf{G}_6$: Same as $\mathbf{G}_5$, except when $\mathcal{A}$ asks for the $\mathsf{UpdHRev}$ query, instead of returning the correctly generated update token, we sample a random value r_{up} and return $\mathsf{up}' \leftarrow \mathsf{Enc}(\mathsf{pk}_i, r_{\mathsf{up}})$. So the response now consists of the incorrectly generated update token up', correctly generated keypair $(\mathsf{pk}, \mathsf{sk})$, and the simulated proof π'. The modified update oracle is shown in Figure 6.9.

Note that until the adversary makes the query on $H(r)$ or $G(r, \mathsf{up}', \mathsf{pk})$, their outputs are uniformly random and the adversary cannot tell that the token was generated incorrectly. So this game continues identically to the previous one until the adversary makes either

of those queries. If any of these queries occur, then the adversary has broken the One-Wayness of the UPKE (which is related to IND-CR-CPA). So if $\mathcal{A}$ makes such a query, then we can have an adversary $\mathcal{B}'$ that uses $\mathcal{A}$ to win the OW-CR-CPA game against $\mathsf{CR}.\Pi$. We construct $\mathcal{B}'$ similar to how we did in Section 3.4, but we need to hold our updates before sending them to the challenger to account for the forking structure in this game. If the adversary makes q_H queries to H, this gives us:

$$|\Pr[\mathbf{G}_6] - \Pr[\mathbf{G}_5]| \leq q_H \cdot \mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B}',\mathsf{CR}.\Pi}(\lambda).$$

Now we can assume that this event does not occur in the following games.

Game $\mathbf{G}_7$: Same as $\mathbf{G}_6$, except when $\mathcal{A}$ asks for the $\mathsf{UpdHRev}$ query, instead of returning the correct keypair $(\mathsf{pk}, \mathsf{sk})$, we run an update on the root node and return the resulting keypair $(\mathsf{pk}', \mathsf{sk}')$. Now the response consists of the incorrectly generated update token up', the incorrectly generated keypair $(\mathsf{pk}', \mathsf{sk}')$, and the simulated proof π'. The modified update oracle is shown in Figure 6.9.

The games continue identically until the adversary receives this response. From our assumption of update uniformity, the adversary cannot tell the difference between the games, so its success probability remains unchanged.

$$\Pr[\mathbf{G}_7] = \Pr[\mathbf{G}_6].$$

Game $\mathbf{G}_8$: Same as $\mathbf{G}_7$, except when $\mathcal{A}$ asks for the challenge query at a node i, we extract all the shift values from it to the root node, and abort if this extraction fails. The modified challenge oracle is shown in Figure 6.10.

As long as we do not abort, there is no difference in the adversary's view between the two games. The probability of this abort is tied to the proof of knowledge property of the NIZK-PoK. If $\mathcal{A}$ makes q_{VerifyO} queries to the $\mathsf{VerifyO}$ oracle, then by the union bound we have:

$$|\Pr[\mathbf{G}_8] - \Pr[\mathbf{G}_7]| \leq (q_{\mathsf{up}} + q_{\mathsf{VerifyO}}) \cdot \epsilon_{\mathsf{Ext}}.$$

Game $\mathbf{G}_9$: Same as $\mathbf{G}_8$, except when $\mathcal{A}$ queries the random oracle F on (pk_{i^*}, m^*), the message encrypted in the challenge ciphertext, we abort. The modified oracles are shown in Figure 6.11.

As long as the adversary does not make this specific query, the games continue identically. If the adversary does make this query, then it has broken the One-Wayness of

Oracle $\mathsf{Chall}(i)$

1 : **req** $\mathsf{pk}_i \neq \mathsf{pk}_{i_{rev}}$ // No query on i_{rev}
2 : **req** $i^* = \bot$ // first and only chall query
3 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
4 : $i^* \leftarrow i$; $\mathsf{K}^0 \leftarrow\!\$\ \mathcal{K}$
5 : $A \leftarrow Stack()$
6 : **while** $i \neq 0$
7 : $\quad d \leftarrow\!\$\ \mathsf{Ext}(\pi_{i-1}, \mathsf{pk}_{par_i}, \mathsf{pk}_i)$
8 : $\quad$ **req** $d \neq \bot$
9 : $\quad A.push(d)$
10 : $\quad i \leftarrow par_i$
11 : $(\mathsf{K}^1, c^*) \leftarrow\!\$\ \mathsf{Encaps}(\mathsf{pk}_i^*)$
12 : **return** (K^b, c^*)

Figure 6.10: The modified oracle in $\mathbf{G}_8$ for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

Oracle $\mathsf{Chall}(i)$

1 : **req** $\mathsf{pk}_i \neq \mathsf{pk}_{i_{rev}}$ // No query on i_{rev}
2 : **req** $i^* = \bot$ // first and only chall query
3 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
4 : $i^* \leftarrow i$; $\mathsf{K}^0 \leftarrow\!\$\ \mathcal{K}$
5 : $A \leftarrow Stack()$
6 : **while** $i \neq 0$
7 : $\quad d \leftarrow\!\$\ \mathsf{Ext}(\pi_{i-1}, \mathsf{pk}_{par_i}, \mathsf{pk}_i)$
8 : $\quad$ **req** $d \neq \bot$
9 : $\quad A.push(d)$
10 : $\quad i \leftarrow par_i$
11 : $m \leftarrow\!\$\ \mathcal{M}$
12 : $c^* \leftarrow \mathsf{Enc}(\mathsf{pk}_{i^*}, m; F(\mathsf{pk}_{i^*}, m))$
13 : $\mathsf{K}^1 \leftarrow K(m, c^*)$
14 : $m^* \leftarrow m$
15 : **return** (K^b, c^*)

Oracle $F(x)$

1 : **if** $x = (\mathsf{pk}_{i*}, m^*)$
2 : $\quad$ **then** $ABORT$
3 : **if** $\exists y : (x, y) \in \mathcal{L}_F$
4 : $\quad$ **then return** y
5 : $r \leftarrow\!\$\ \mathcal{R}_1$
6 : $\mathcal{L}_F \leftarrow \mathcal{L}_F \cup (x, r)$
7 : **return** r

Figure 6.11: The modified oracles in $\mathbf{G}_9$ for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

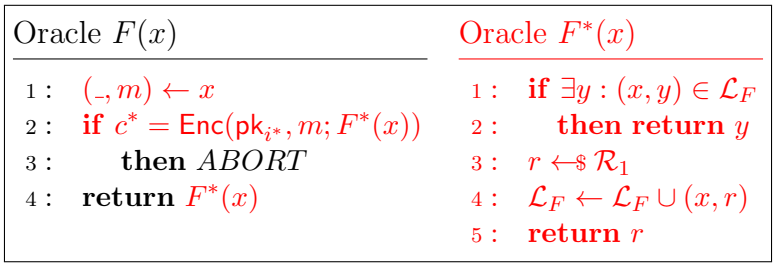


Figure 6.12: The modified oracles in $\mathbf{G}_{10}$ for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

the underlying encryption scheme and an adversary $\mathcal{B}''$ could use $\mathcal{A}$ as a subroutine to win the One-Wayness game. Once again, $\mathcal{B}''$ is constructed similar to our description in Section 3.4, but instead of embedding the the one-way challenge as the update token, we can embed it into the challenge ciphertext. Since we have already forged the update token for the $\mathsf{UpdHRev}$ query, $\mathcal{B}''$ can do so as well. If $\mathcal{A}$ makes q_F queries to F, then with this construction, we have:

$$|\Pr[\mathbf{G}_9] - \Pr[\mathbf{G}_8]| \leq q_F \cdot \mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B}'',\text{CR}.\Pi}(\lambda).$$

Game $\mathbf{G}_{10}$: Same as $\mathbf{G}_9$, except we modify the abort condition in the random oracle F to abort on any public key, message that re-encrypts to the challenge ciphertext. The modified oracles are shown in Figure 6.12.

The probability that there is a message $m \neq m^*$ that also encrypts to the challenge ciphertext is tied to the correctness of the underlying UPKE. Since we assume that the schemes have negligible correctness errors, we get:

$$|\Pr[\mathbf{G}_{10}] - \Pr[\mathbf{G}_9]| \leq \mathsf{negl}(\lambda).$$

Game $\mathbf{G}_{11}$: Same as $\mathbf{G}_{10}$, except for two changes:

- We generate the challenge query on the root node 0 to get K^b, c^*, and then shift the ciphertext to the appropriate node to get c'^*. This also requires a minor modification in F for correctness.
- We swap the responses of $K(m^*, c^*)$ and $K(m^*, c'^*)$.

As a result of shifting the challenge ciphertext, we destroy the relation between it and the challenge shared. To remedy this, we swap the query responses so that the adversary can still learn the information it needs to win the game, while maintaining the overall distribution of the responses from the random oracle. While performing this swap, we must ensure that the adversary does not notice any changes. If the adversary were to somehow guess the challenge payload before the challenge ciphertext is created, it could notice that K oracle behaves differently before and after the challenge query. So we move the sampling of the challenge payload to the main function and additionally add an abort condition in the K oracle as well. The modified oracles are shown in Figure 6.13.

Since we assume that our ciphertext updates are perfect and abort if the adversary tries to recover the corresponding randomness, our patch to the K oracle ensures that this game proceeds identically to the previous one, except if it can guess the challenge payload before making the challenge query. The probability of making such a guess depends on the size of the message space, and the number of queries to the K oracle. With this change, if the adversary makes q_K queries to the K oracle, we get:

$$|\Pr[\mathbf{G}_{11}] - \Pr[\mathbf{G}_{10}]| \leq q_K/|\mathcal{M}|.$$

Game $\mathbf{G}_{12}$: Same as $\mathbf{G}_{11}$, except we modify the patch in the K oracle from m^* to any message that encrypts to c^*. The modified oracle K in this game is shown in Figure 6.14.

As we have argued previously, the probability that there is a message $m \neq m^*$ that also encrypts to the same ciphertext is tied to the correctness of the underlying UPKE. Since we assume that the UPKE has negligible correctness errors, we have:

$$|\Pr[\mathbf{G}_{12}] - \Pr[\mathbf{G}_{11}]| \leq \mathsf{negl}(\lambda).$$

Analysis of Game $\mathbf{G}_{12}$: Lastly, we show that if an adversary can beat $\mathbf{G}_{12}$, then it can be used to win the F-IND-CU-CCA-PV game. We construct an adversary $\mathcal{B}$ that uses $\mathcal{A}$ to win against a F-IND-CU-CCA-PV challenger. The adversary $\mathcal{B}$ starts off by immediately asking for the challenge ciphertext c^* on the root node, followed by an $\mathsf{UpdHRev}$ query on the root node. With these two queries, it gets the unshifted challenge ciphertext and the keypair to forge the honest update. It has everything it needs to simulate all the oracles for $\mathcal{A}$. After this, $\mathcal{B}$ passes the public key to $\mathcal{A}$ and is able to handle all its oracle queries without interacting with the challenger. Finally, $\mathcal{B}$ outputs the same bit as $\mathcal{A}$ and wins with the same probability that $\mathcal{A}$ wins in $\mathbf{G}_{12}$. For an adversary $\mathcal{B}$ constructed in this way, we get

$$|\Pr[\mathbf{G}_{12}] - 1/2| = \mathsf{Adv}^{\text{F-IND-CU-CCA-PV}}_{\mathcal{B},\text{CCA}.\Pi}(\lambda).$$

UKEM F-IND-CU-CCA-PV2$^{\mathcal{A}}_{\Pi}(1^\lambda)$

$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\$\ \mathsf{KGen}(1^\lambda)$
$(par_0, j, typ_j) \leftarrow (\epsilon, 0, \text{“}full\text{”})$
$(i^*, c^*) \leftarrow (\bot, \bot)$
$(\mathcal{L}_G, \mathcal{L}_H, \mathcal{L}_F, \mathcal{L}_K) \leftarrow (\bot, \bot, \bot, \bot)$
$b \leftarrow\!\$\ \{0,1\};\ m^* \leftarrow\!\$\ \mathcal{M}$
$\mathcal{O} \leftarrow \{\mathsf{UpdO}, \mathsf{DecO}, \mathsf{VerifyO}, \mathsf{Chall}\}$
$\mathcal{O} \leftarrow \mathcal{O} \cup \{\mathsf{UpdHRev}, G, H, F, K\}$
$b' \leftarrow\!\$\ \mathcal{A}^{\mathcal{O}}(\mathsf{pk}_0)$
foreach $\big((r, _, _), y\big) \in \mathcal{L}_G$
  foreach $i \in \{0, \ldots, j\}$ **with** $typ_i = \text{“}full\text{”}$
    if $\mathsf{Dec}\big(\mathsf{sk}_i, \mathsf{Enc}(\mathsf{pk}_i, r; y)\big) \neq r$
      return $\bot$
foreach $\big((\mathsf{pk}_i, m), r\big) \in \mathcal{L}_F$ **with** $\mathsf{sk}_i \neq \bot$
  if $\mathsf{Dec}\big(\mathsf{sk}_i, \mathsf{Enc}(\mathsf{pk}_i, m; r)\big) \neq m$
    return $\bot$
return $b = b'$

Oracle $F(x)$

$(_, m) \leftarrow x$
if $c^* = \mathsf{Enc}(\mathsf{pk}_0, m; F^*(x))$
  then $ABORT$
return $F^*(x)$

Oracle $K^*(x)$

if $\exists y : (x, y) \in \mathcal{L}_K$
  then return y
$r \leftarrow\!\$\ \mathcal{K}$
$\mathcal{L}_K \leftarrow \mathcal{L}_K \cup (x, r)$
return r

Oracle $\mathsf{Chall}(i)$

req $i \neq i_{rev}$ // No query on i_{rev}
req $i^* = \bot$ // first and only chall query
req $\mathsf{pk}_i \neq \bot$ // ith node exists
req $\mathsf{pk}_i \neq \mathsf{pk}_{i_{rev}}$ // No trivial win
$i^* \leftarrow i;\ \mathsf{K}^0 \leftarrow\!\$\ \mathcal{K}$
$A \leftarrow Stack()$
while $i \neq 0$
  $d \leftarrow\!\$\ \mathsf{Ext}(\pi_{i-1}, \mathsf{pk}_{par_i}, \mathsf{pk}_i)$
  req $d \neq \bot$
  $A.push(d)$
  $i \leftarrow par_i$
$c^* \leftarrow \mathsf{Enc}(\mathsf{pk}_0, m^*; F(\mathsf{pk}_0, m^*))$
$\mathsf{K}^1 \leftarrow K^*(m^*, c^*);\ c'^* \leftarrow c^*$
while $A.is_not_empty()$
  $d \leftarrow A.pop()$
  $c'^* \leftarrow \mathsf{ShiftCt}(c'^*, d)$
return (K^b, c'^*)

Oracle $K(x)$

$(m, c) \leftarrow x$
if $m = m^*$
  if $i^* = \bot$ **then return** $\bot$
  if $c = c'^*$
    then return $K^*(m, c^*)$
  if $c = c^*$
    then return $K^*(m, c'^*)$
return $K^*(x)$

Figure 6.13: The modified oracles in $\mathbf{G}_{11}$ for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

Oracle $K(x)$

1 : $(m, c) \leftarrow x$
2 : **if** $c^* = \mathsf{Enc}(\mathsf{pk}_0, m; F^*(\mathsf{pk}_0, m))$
3 : **if** $i^* = \bot$ **then return** $\bot$
4 : **if** $c = c'^*$
5 : **then return** $K^*(m, c^*)$
6 : **if** $c = c^*$
7 : **then return** $K^*(m, c'^*)$
8 : **return** $K^*(x)$

Figure 6.14: The modified oracles in $\mathbf{G}_{12}$ for the proof of Theorem 5 for lifting forking chosen update CCA-secure UKEM with public verifiability to re-order security.

Combining all the bounds accumulated during the game hops yields the desired bound in the theorem. □

6.4 Lifting Reordered Forking Secure UKEM To Security Against Selective Corruptions

For this transform, which lifts reordered forking to full forking with selective corruptions, we do not need any additional assumptions beyond those made for our previous theorems. The following theorem for proving selective security will rely on the forging honest update trick from Theorem 3.

Theorem 6. *Let* $\mathsf{CCA}.\Pi$ *be a* F-IND-CU-CCA-PV2 *secure UPKE obtained by using the transforms described in Figure 3.4, Figure 4.2, and Figure 5.2 on a* S-IND-CR-CPA *secure UPKE* $\mathsf{CR}.\Pi$*, and let the associated NIZK-PoK be* ϵ_{ZK}*-zero knowledge. Then if* $\mathsf{CCA}.\Pi$ *has update uniformity, it is also* Sel-F-IND-CU-CCA-PV *secure.*

For any successful Sel-F-IND-CU-CCA-PV *adversary* $\mathcal{A}$ *that makes at most* q_{UpdHO} *queries to the* UpdHO *oracle, and* q_H *queries to* H*, there exists an* OW-CR-CPA *adversary* $\mathcal{B}'$ *and a* F-IND-CU-CCA-PV2 *adversary* $\mathcal{B}$ *such that*

$$\mathsf{Adv}^{\text{Sel-F-IND-CU-CCA-PV}}_{\mathcal{A},\mathsf{CCA}.\Pi}(\lambda) \leq q_{\mathsf{UpdHO}} \cdot \left(\epsilon_{\mathsf{ZK}} + q_H \cdot \mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B}',\mathsf{CR}.\Pi}(\lambda)\right) + \mathsf{Adv}^{\text{F-IND-CU-CCA-PV2}}_{\mathcal{B},\mathsf{CCA}.\Pi}(\lambda).$$

We start with the **proof outline**. The main idea of this proof is to split the nodes into two distinct sets and handle each set separately. We want the nodes inside the challenge set to be present at the challenger, and hence, we do not know the corresponding secret key. At the same time, we want to know the secret keys for the nodes outside the challenge set so that we can answer reveal queries. To achieve this, we will forge every full update that connects a node inside the challenge set to one outside it. Note that, by definition, any such update that connects a node inside the challenge set to one outside must be an honest update. This forgery is the same as the one we have used in the previous proofs, where we rely on OW-CR-CPA, zero knowledge, and update uniformity.

Note that this difference in how the two sets of nodes are handled results in the indices of the nodes differing between the reduction and the challenger. To remedy this, we make use of a map to find the corresponding index of a node at the challenger in our pseudocode.

We start with the unmodified Sel-F-IND-CU-CCA-PV game $\mathbf{G}_0$. In $\mathbf{G}_1$, we generate $(\mathsf{pk}_f, \mathsf{sk}_f)$ by updating the root node, and begin forging honest updates connecting nodes inside the challenge set to ones outside by replacing the proofs with ones generated using simulators. In $\mathbf{G}_2$, we continue the forgeries by replacing the update tokens with forged ones. In $\mathbf{G}_3$, we complete our forgeries by replacing the correct key with one resulting from updating $(\mathsf{pk}_f, \mathsf{sk}_f)$. Finally, we show that if an adversary $\mathcal{A}$ can beat $\mathbf{G}_3$, then it can be used to beat the F-IND-CU-CCA-PV2 game against $\mathsf{CCA}.\Pi$.

Proof. We now give the **full proof**. It proceeds as a sequence of games against an adversary $\mathcal{A}$ for the Sel-F-IND-CU-CCA-PV security of $\mathsf{CCA}.\Pi$. Let $\Pr[\mathbf{G}_i]$ be the probability that $\mathcal{A}$ wins the i^{th} game.

Game $\mathbf{G}_0$: The Sel-F-IND-CU-CCA-PV game against $\mathcal{A}$. Everything proceeds as normal. In addition to the oracles in the game, $\mathcal{A}$ also has access to random oracles H, G, F, and K. This game is shown in Figure 6.15.

By definition, we have:

$$|\Pr[\mathbf{G}_0] - 1/2| = \mathsf{Adv}^{\text{Sel-F-IND-CU-CCA-PV}}_{\mathcal{A},\mathsf{CCA}.\Pi}(\lambda).$$

Game $\mathbf{G}_1$: Same as $\mathbf{G}_0$, except for two changes:

- We update the root node to get $(\mathsf{pk}_f, \mathsf{sk}_f)$.
- Whenever there is a honest update connecting a full node in a challenge set to a full node outside it, we generate the associated NIZK-PoK using the simulator.

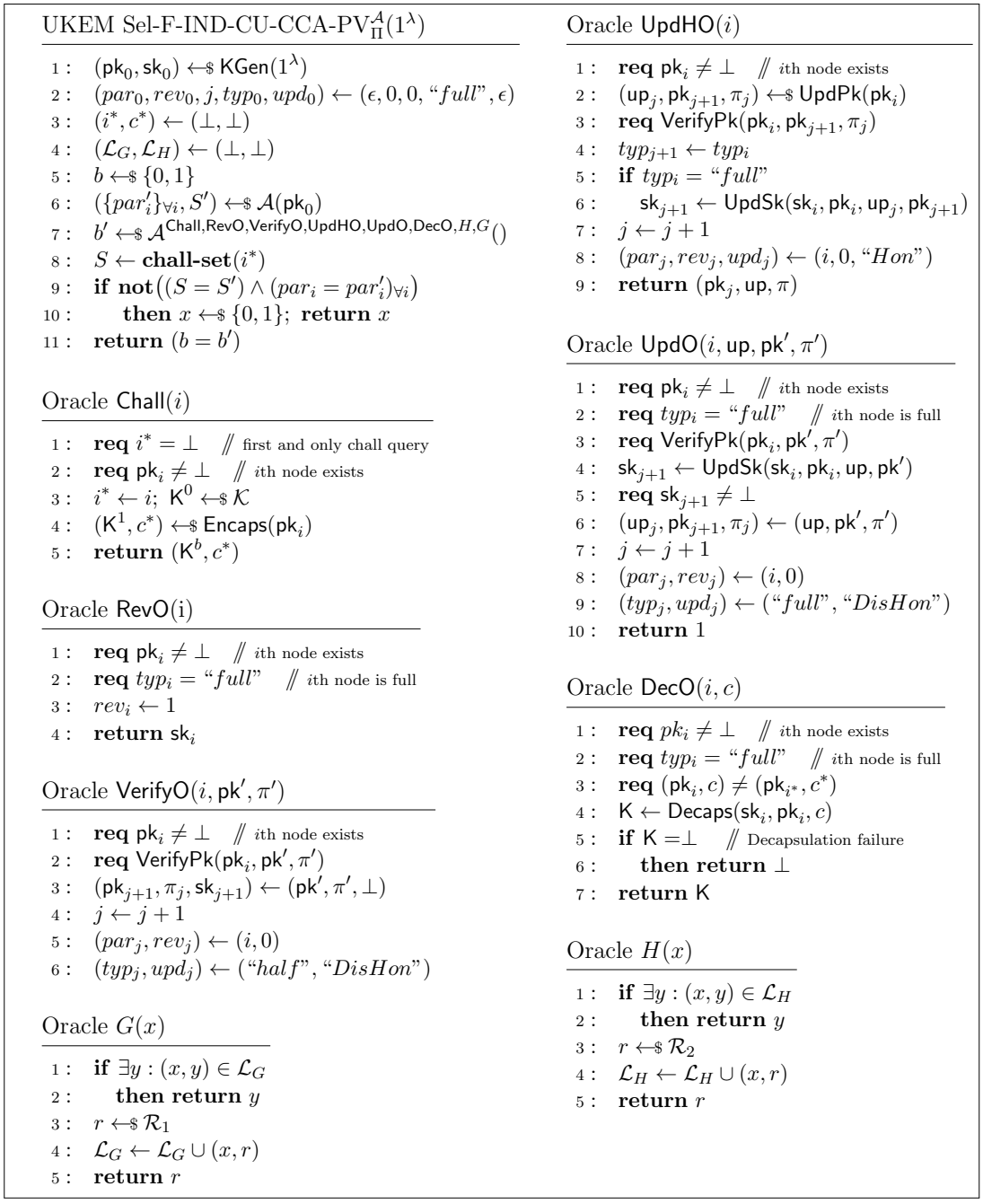


Figure 6.15: The game $\mathbf{G}_0$ for the proof of Theorem 6 for lifting a UKEM with reordered security to one secure in the presence of selective corruptions.

UKEM Sel-F-IND-CU-CCA-PV$^{\mathcal{A}}_{\Pi}(1^\lambda)$

1 : $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow\!\$\ \mathsf{KGen}(1^\lambda)$
2 : $(par_0, rev_0, j) \leftarrow (\epsilon, 0, 0)$
3 : $(typ_0, upd_0) \leftarrow (\text{“}full\text{”}, \epsilon)$
4 : $(i^*, c^*) \leftarrow (\bot, \bot)$
5 : $(\mathcal{L}_G, \mathcal{L}_H) \leftarrow (\bot, \bot)$
6 : $b \leftarrow\!\$\ \{0,1\}$
7 : $(\mathsf{up}_f, \mathsf{pk}_f, \pi_f) \leftarrow\!\$\ \mathsf{UpdPk}(\mathsf{pk}_0)$
8 : $\mathsf{sk}_f \leftarrow \mathsf{UpdSk}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{up}_f, \mathsf{pk}_f)$
9 : $(\{par'_i\}_{\forall i}, S') \leftarrow\!\$\ \mathcal{A}(\mathsf{pk}_0)$
10 : $b' \leftarrow\!\$\ \mathcal{A}^{\mathsf{Chall},\mathsf{RevO},\mathsf{VerifyO},\mathsf{UpdHO},\mathsf{UpdO},\mathsf{DecO},H,G}()$
11 : $S \leftarrow \textbf{chall-set}(i^*)$
12 : **if not**$\big((S = S') \wedge (par_i = par'_i)_{\forall i}\big)$
13 : **then** $x \leftarrow\!\$\ \{0,1\}$; **return** x
14 : **return** $(b = b')$

Oracle $\mathsf{UpdHO}(i)$

1 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
2 : $(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow\!\$\ \mathsf{UpdPk}(\mathsf{pk}_i)$
3 : **req** $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}_{j+1}, \pi_j)$
4 : $typ_{j+1} \leftarrow typ_i$
5 : **if** $typ_i = \text{“}full\text{”}$
6 : $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_j, \mathsf{pk}_{j+1})$
7 : **if** $(i \notin S') \wedge (j+1 \in S')$
8 : $\pi_j \leftarrow\!\$\ \mathsf{Sim}(\mathsf{pk}_i, \mathsf{pk}_{j+1})$
9 : $j \leftarrow j + 1$
10 : $(par_j, rev_j, upd_j) \leftarrow (i, 0, \text{“}Hon\text{”})$
11 : **return** $(\mathsf{pk}_j, \mathsf{up}, \pi)$

Figure 6.16: The modified oracles in $\mathbf{G}_1$ for the proof of Theorem 6 for lifting a UKEM with reordered security to one secure in the presence of selective corruptions.

The modified oracles are shown in Figure 6.16.

The first change has no effect on the adversary's view. The difference in probability due to the second change is tied to the zero knowledge property of the NIZK-PoK. If $\mathcal{A}$ makes at most q_{UpdHO} queries to the UpdHO oracle, then by the union bound we have:

$$|\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_2]| \leq q_{\mathsf{UpdHO}} \cdot \epsilon_{\mathsf{ZK}}.$$

Game $\mathbf{G}_2$: Same as $\mathbf{G}_1$, except whenever there is an honest update connecting a full node in the challenge set to a full node outside it, we forge the update token by encrypting a random value r. The modified oracles are shown in Figure 6.17.

Note that until the adversary makes the query on $H(r)$ or $G(r, \mathsf{up}, \mathsf{pk}')$, their outputs are uniformly random and the adversary cannot tell that the token was generated incorrectly. So this game continues identically to the previous one until the adversary makes either of those queries. If any of the queries occur, the adversary knows r and has broken the One-Wayness of the UPKE. So if $\mathcal{A}$ makes such a query, then we can construct an adversary $\mathcal{B}'$ that uses $\mathcal{A}$ to win the OW-CR-CPA game against $\mathsf{CR}.\Pi$. We construct $\mathcal{B}'$ similar to how we did in theorem 3 and Section 3.4. Since the adversary makes at most q_{UpdHO} queries to

Oracle $\mathsf{UpdHO}(i)$ (in $\mathbf{G}_2$)

1 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
2 : $(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow\!\$\ \mathsf{UpdPk}(\mathsf{pk}_i)$
3 : **req** $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}_{j+1}, \pi_j)$
4 : $typ_{j+1} \leftarrow typ_i$
5 : **if** $typ_i =$ "$full$"
6 : $\quad \mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_j, \mathsf{pk}_{j+1})$
7 : $\quad$ **if** $(i \notin S') \wedge (j+1 \in S')$
8 : $\quad\quad \pi_j \leftarrow\!\$\ \mathsf{Sim}(\mathsf{pk}_i, \mathsf{pk}_j)$
9 : $\quad\quad r \leftarrow\!\$\ \mathcal{M}$
10 : $\quad\quad \mathsf{up}_j \leftarrow\!\$\ \mathsf{Enc}(\mathsf{pk}_i, r)$
11 : $j \leftarrow j+1$
12 : $(par_j, rev_j, upd_j) \leftarrow (i, 0, \text{"}Hon\text{"})$
13 : **return** $(\mathsf{pk}_j, \mathsf{up}, \pi)$

Oracle $\mathsf{UpdHO}(i)$ (in $\mathbf{G}_3$)

1 : **req** $\mathsf{pk}_i \neq \bot$ // ith node exists
2 : $(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow\!\$\ \mathsf{UpdPk}(\mathsf{pk}_i)$
3 : **req** $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}_{j+1}, \pi_j)$
4 : $typ_{j+1} \leftarrow typ_i$
5 : **if** $typ_i =$ "$full$"
6 : $\quad \mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}_j, \mathsf{pk}_{j+1})$
7 : $\quad$ **if** $(i \notin S') \wedge (j+1 \in S')$
8 : $\quad\quad (\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow\!\$\ \mathsf{UpdPk}(\mathsf{pk}_f)$
9 : $\quad\quad \mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_f, \mathsf{pk}_f, \mathsf{up}_j, \mathsf{pk}_{j+1})$
10 : $\quad\quad \pi_j \leftarrow\!\$\ \mathsf{Sim}(\mathsf{pk}_i, \mathsf{pk}_{j+1})$
11 : $\quad\quad r \leftarrow\!\$\ \mathcal{M}$
12 : $\quad\quad \mathsf{up}_j \leftarrow\!\$\ \mathsf{Enc}(\mathsf{pk}_i, r)$
13 : $j \leftarrow j+1$
14 : $(par_j, rev_j, upd_j) \leftarrow (i, 0, \text{"}Hon\text{"})$
15 : **return** $(\mathsf{pk}_j, \mathsf{up}, \pi)$

Figure 6.17: The modified oracles in $\mathbf{G}_2$ (left) and $\mathbf{G}_3$ (right) for the proof of Theorem 6 for lifting a UKEM with reordered security to one secure in the presence of selective corruptions.

the UpdHO oracle, and q_H queries to H, we have:

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq q_{\mathsf{UpdHO}} \cdot q_H \cdot \mathsf{Adv}^{\text{OW-CR-CPA}}_{\mathcal{B}',\mathsf{CR}.\Pi}(\lambda).$$

Game $\mathbf{G}_3$: Same as $\mathbf{G}_2$, except whenever there is a honest update connecting a full node in the challenge set to a full a node outside it, instead of assigning the correct keypair to the newly created node, we run a update on $(\mathsf{pk}_f, \mathsf{sk}_f)$ and assign the resulting key as the keypair outputted by the honest update. The modified oracles are shown in Figure 6.17.

By our assumption of update uniformity, the adversary cannot distinguish between the two games, and its success probability remains unaffected. Hence, we have:

$$\Pr[\mathbf{G}_3] = \Pr[\mathbf{G}_2]$$

Analysis of Game $\mathbf{G}_3$: Finally, we show that if $\mathcal{A}$ can beat $\mathbf{G}_3$, then it can be used by another adversary $\mathcal{B}$ to beat the F-IND-CU-CCA-PV2 game against $\mathsf{CCA}.\Pi$.

$\mathcal{B}^{\mathsf{Ch}.\{\mathsf{Chall},\mathsf{VerifyO},\mathsf{UpdO},\mathsf{DecO},H,G\}}(1^\lambda, \mathcal{A}, \mathsf{pk}_0)$

$(par_0, rev_0, j) \leftarrow (\epsilon, 0, 0)$
$(typ_0, upd_0) \leftarrow (\text{“}full\text{”}, \epsilon)$
$(i^*, c^*) \leftarrow (\perp, \perp)$
$(\mathcal{L}_G, \mathcal{L}_H, \mathcal{L}_r) \leftarrow (\perp, \perp, \perp)$
$(\mathsf{pk}_f, \mathsf{sk}_f, \mathsf{up}_f, \pi_f) \leftarrow\!\$\ \mathsf{Ch.UpdHRev}(0)$
$(\{par'_i\}_{\forall i}, S') \leftarrow\!\$\ \mathcal{A}(\mathsf{pk}_0)$
$b' \leftarrow\!\$\ \mathcal{A}^{\mathsf{Chall},\mathsf{RevO},\mathsf{VerifyO},\mathsf{UpdHO},\mathsf{UpdO},\mathsf{DecO},H,G}()$
$S \leftarrow \textbf{chall-set}(i^*)$
return b'

Oracle $\mathsf{VerifyO}(i, \mathsf{pk}', \pi')$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
if $i \in S'$
  $id \leftarrow Id\text{-}Map(i)$
  $\mathsf{Ch.VerifyO}(id, \mathsf{pk}', \pi')$
$(\mathsf{pk}_{j+1}, \pi_j, \mathsf{sk}_{j+1}) \leftarrow (\mathsf{pk}', \pi', \perp)$
$j \leftarrow j + 1$
$(par_j, rev_j) \leftarrow (i, 0)$
$(typ_j, upd_j) \leftarrow (\text{“}half\text{”}, \text{“}DisHon\text{”})$

Oracle $\mathsf{Chall}(i)$

req $i^* = \perp$ // first and only chall query
req $\mathsf{pk}_i \neq \perp$ // ith node exists
$i^* \leftarrow i$; $id \leftarrow Id\text{-}Map(i)$
$(\mathsf{K}^*, c^*) \leftarrow\!\$\ \mathsf{Ch.Chall}(id)$
return (K^*, c^*)

Oracle $\mathsf{RevO}(\mathrm{i})$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
req $i \notin S'$
$rev_i \leftarrow 1$
return sk_i

Oracle $\mathsf{UpdHO}(i)$

// See Figure 6.19

Oracle $\mathsf{DecO}(i, c)$

req $pk_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
req $(\mathsf{pk}_i, c) \neq (\mathsf{pk}_{i^*}, c^*)$
if $i \notin S'$
  $id \leftarrow Id\text{-}Map(i)$
  $\mathsf{K} \leftarrow \mathsf{Ch.DecO}(id, c)$
else
  $\mathsf{K} \leftarrow \mathsf{Decaps}(\mathsf{sk}_i, \mathsf{pk}_i, c)$
  if $\mathsf{K} = \perp$ // Decapsulation failure
    then return $\perp$
return K

Oracle $\mathsf{UpdO}(i, \mathsf{up}, \mathsf{pk}', \pi')$

req $\mathsf{pk}_i \neq \perp$ // ith node exists
req $typ_i = \text{“}full\text{”}$ // ith node is full
req $\mathsf{VerifyPk}(\mathsf{pk}_i, \mathsf{pk}', \pi')$
if $i \in S'$
  $id \leftarrow Id\text{-}Map(i)$
  req $\mathsf{Ch.UpdO}(id, \mathsf{up}, \mathsf{pk}', \pi')$
else
  $\mathsf{sk}_{j+1} \leftarrow \mathsf{UpdSk}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{up}, \mathsf{pk}')$
  req $\mathsf{sk}_{j+1} \neq \perp$
$(\mathsf{up}_j, \mathsf{pk}_{j+1}, \pi_j) \leftarrow (\mathsf{up}, \mathsf{pk}', \pi')$
$j \leftarrow j + 1$
$(par_j, rev_j) \leftarrow (i, 0)$
$(typ_j, upd_j) \leftarrow (\text{“}full\text{”}, \text{“}DisHon\text{”})$
return 1

Figure 6.18: The construction of $\mathcal{B}$ that uses an adversary for the selective corruptions game for a UKEM to win the reordered security game for the proof of Theorem 6. A part of this description is given in Figure 6.19.

Oracle $\mathsf{UpdHO}(i)$

```
1 : req pk_i ≠ ⊥   // ith node exists
2 : if i ∈ S
3 :   id ← Id-Map(i)
4 :   if j+1 ∉ S
5 :     if typ_i = "full"
6 :       // Boundary and full node
7 :       // new forged update
8 :       (up', pk_{j+1}, π') ←$ UpdPk(pk_f)
9 :       sk_{j+1} ← UpdSk(sk_f, pk_f, up', pk_{j+1})
10 :      π_j ←$ Sim(pk_i, pk_{j+1})
11 :      r ←$ M
12 :      up_j ←$ Enc(pk_i, r)
13 :      L_r ← r ∪ L_r
14 :    else   // typ_i = "half"
15 :      // Boundary and half node
16 :      // Manual half update
17 :      (up_j, pk_{j+1}, π_j) ←$ UpdPk(pk_i)
18 :  else   // j+1 ∈ S
19 :    (up_j, pk_{j+1}, π_j) ←$ UpdPk(pk_i)
20 :    if typ_i = "full"
21 :      // Inside and full node
22 :      // Manual half and Ch.UpdO
23 :      Ch.UpdO(id, up_j, pk_{j+1}, π_j)
24 :    else   // typ_i = "half"
25 :      // Inside and half node
26 :      // Manual half and Ch.VerifyO
27 :      Ch.VerifyO(id, pk_{j+1}, π_j)
28 : else   // i ∉ S
29 :   // j+1 is always outside
30 :   (up_j, pk_{j+1}, π_j) ←$ UpdPk(pk_i)
31 :   if typ_i = "full"
32 :     // Outside and full node
33 :     // Manual half and manual full
34 :     sk_{j+1} ← UpdSk(sk_i, pk_i, up_j, pk_{j+1})
35 :   else   // typ_i = "half"
36 :     // Outside and half
37 :     // Manual half
38 : typ_{j+1} ← typ_i
39 : j ← j+1
40 : (par_j, rev_j, upd_j) ← (i, 0, "Hon")
41 : return (pk_j, up, π)
```

Figure 6.19: The UpdHO oracle in the construction of $\mathcal{B}$ that uses an adversary for the selective corruptions game for a UKEM to win the reordered security game for the proof of Theorem 6.

The reduction works as follows: we use the challenger to handle queries on the nodes inside the challenge set; we forge full honest updates connecting nodes inside the challenge set to a node outside it; and we handle all queries on the nodes outside the challenge set ourselves. These forged updates allow us to know the secret key for all nodes outside the challenge set, which enables us to answer all update, decapsulate, and reveal queries on these nodes. To implement this idea in the reduction, a simple check whether i is in the challenge set suffices for most oracles. If i is in the challenge set, we call the corresponding oracle at the challenger, and handle things ourselves otherwise.

The honest update oracle, however, is not present at the challenger and needs more care to be handled correctly. We handle each case in which the honest update oracle can

be called differently. For the full updates connecting a node inside the challenge set to one outside it, we forge the update as in the previous games. We ask for a $\mathsf{UpdHRev}$ query on the root node as soon as the game starts so that $(\mathsf{pk}_f, \mathsf{sk}_f)$ exist for these forged updates. For nodes that are completely inside the challenge set, we run an honest update and then call either the challenger's UpdO oracle (full update) or $\mathsf{VerifyO}$ oracle (half update). Lastly, we handle all types of updates that occur outside the challenge set by manually running the honest update.

Since the nodes are created in a random order at the reduction, and only a few of those nodes are recreated at the challenger, the indexing of the nodes at the challenger is different from that at the reduction. To remedy this, we make use of the $Id\text{-}Map$ function, which takes in the index at the reduction and outputs the corresponding index at the challenger. By designing oracles in this way, $\mathcal{B}$ perfectly simulates $\mathcal{A}$'s view from $\mathbf{G}_3$. Hence, we have:

$$|\Pr[\mathbf{G}_3] - 1/2| = \mathsf{Adv}^{\text{F-IND-CU-CCA-PV2}}_{\mathcal{B},\mathsf{CCA}.\Pi}(\lambda).$$

Combining all the bounds accumulated during the game hops yields the desired bound in the theorem. □

Chapter 7

Applicability

Our transformations show that if a $\mathsf{CR}.\Pi$ UPKE is S-IND-CR-CPA secure, and has both ciphertext updatability and update uniformity, then it can be lifted to $\mathsf{Sel}.\Pi$, a Sel-F-IND-CU-CCA-PV secure UKEM as shown in Figure 1.1. The scheme is transformed by applying the transformations shown in Figure 3.4, Figure 4.2, and Figure 5.2 while security follows from all the theorems we have proved. The combined transform can be found in Figure 7.1

We observe that many UPKE schemes in the literature already satisfy ciphertext updatability and update uniformity. In this chapter, we look at a few constructions based on various hard problems.

Discrete logarithm-based. For UPKEs based on the hardness of discrete logarithm, most schemes are built on ElGamal encryption. The scheme given in [DKW21] is one such example:

- The private key is a collection of one or many exponents and the public key consists of many generators raised to these exponents.
- The ciphertext consists of two parts: one containing the randomness in the exponent and the other containing the message.
- Updates to the keys are performed by adding values to the private exponents.

For a given keypair $((g, h), s)$ being updated using shift value d, the resultant keypair is $((g, h \cdot g^d), s + d)$. Any ciphertext $(c_1, c_2) = (g^r, h^r \cdot m)$ under the first keypair can be

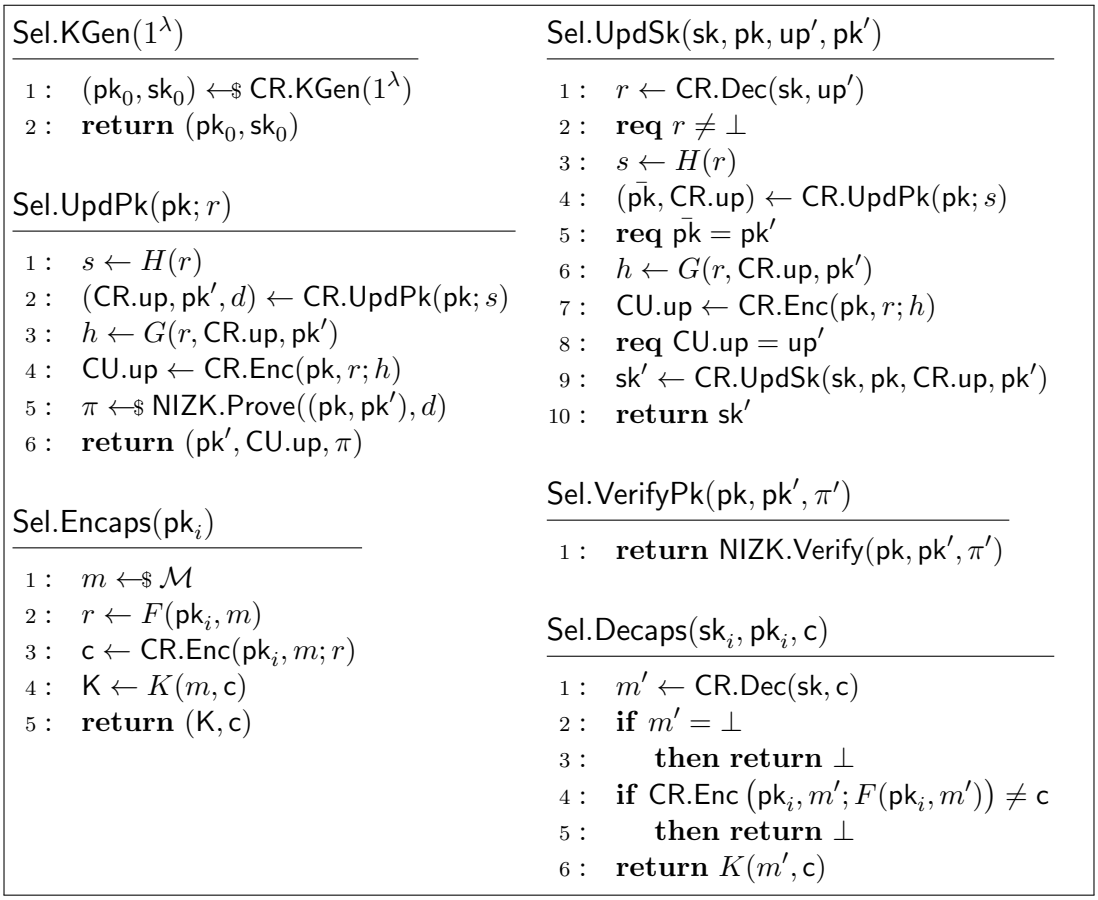


Figure 7.1: The transform for lifting a chosen randomness CPA-secure UPKE to a forking, chosen update, CCA-secure UKEM with public verifiability and selective corruptions resilience.

updated to be encrypted using the second keypair as: $(c_1', c_2') = (c_1, c_2 \cdot c_1^d)$. This technique is easily extended to [DKW21]. Update uniformity is provided by the fact that, for any two keypairs in the same update closure, there always exists a shift value d connecting them.

LWE-based. UPKEs based on LWE closely follow the structure of Regev's LWE scheme [Reg05], and we examine the UPKE of [HPS23]. The public key and encryption are similar in structure to Regev's scheme, and the updates are performed by adding a vector to the secret key and increasing the noise in the public key. For a given key pair $((\mathbf{A}, \mathbf{b}), \mathbf{s}) = ((\mathbf{A}, \mathbf{As} + \mathbf{e_1}), \mathbf{s})$ being updated using shift value $\mathbf{d}$, the resultant keypair is $((\mathbf{A}', \mathbf{b}'), \mathbf{s}') = ((\mathbf{A}, \mathbf{As} + \mathbf{Ad} + \mathbf{e_1} + \mathbf{e_2}), \mathbf{s} + \mathbf{d})$. Any ciphertext $(\mathbf{c_1}, c_2) = (\mathbf{xA}, \mathbf{xb} + e_3 + f \cdot m)$ encrypted under the first keypair can be updated to be encrypted using the second keypair as $(\mathbf{c_1}', c_2') = (\mathbf{c_1}, c_2 + \mathbf{c_1 d} + e_4)$. The updated ciphertexts are indistinguishable by the hardness of LWE. Update uniformity follows from the fact that there always exists a shift value $\mathbf{d}$ relating any two keypairs in the same update closure and the hardness of LWE.

LIP-based. The hardness of the lattice isomorphism problem (LIP) has also been used to make UPKEs. In Hollow LWE [ABL25], the secret key is a vector $\mathsf{sk} = \mathbf{r}$ and the public key is $\mathsf{pk} = (\mathbf{A}, \mathbf{r}^\mathsf{T}\mathbf{A})$ for some tall matrix $\mathbf{A}$. The ciphertexts are similar to Regev's LWE scheme as $(\mathbf{c_1}, c_2) = (\mathbf{Ax} + \mathbf{e}, \mathbf{r}^\mathsf{T}\mathbf{Ax} + e + f \cdot m)$ for some random vector $\mathbf{x}$. The shift values consist of a random orthogonal matrix $\mathbf{O}$ and a unimodular matrix $\mathbf{U}$. Keys are updated by setting $\mathsf{sk}' = \mathbf{Or}$ and $\mathsf{pk}' = (\mathbf{OAU}, \mathbf{r}^\mathsf{T}\mathbf{AU})$. Using the shift value, we are able to update any ciphertext $(\mathbf{c_1}, c_2)$ encrypted under pk using randomness $\mathbf{x}$ to a ciphertext encrypted under pk' using randomness $\mathbf{U}^{-1}\mathbf{x}$ by setting $(\mathbf{c_1}', c_2') = (\mathbf{Oc_1}, c_2)$. The updated ciphertexts are statistically indistinguishable from fresh encryptions under pk'. Update uniformity follows from the fact that there always exists a shift value $(\mathbf{O}, \mathbf{U})$ connecting any two keys in the same update closure.

As we have shown, our transformations are applicable to a wide range of UPKE schemes.

Chapter 8

Conclusions and Future Work

In this thesis, we have constructed generic transformations for UPKEs. These transformations lift any indistinguishability under chosen randomness and chosen plaintext attack (IND-CR-CPA) secure UPKE with ciphertext updatability and update uniformity to a UKEM that is secure under forking indistinguishability under chosen update, chosen ciphertext attacks with public verifiability and selective corruptions (Sel-F-IND-CU-CCA-PV). We hope these transformations make it easier to instantiate strongly secure UPKEs and improve their security in secure messaging contexts.

There are two few possible areas for future work:

- **QROM security**: Our transformations were proven secure in the random oracle model (ROM). However, a quantum adversary might be able to query the random oracle in superposition, so there is a need for further analysis in the quantum random oracle model (QROM) to assess the security of our transforms against such adversaries.
- **Adaptive corruptions**: Our transformations can only achieve security in the presence of selective corruptions. Proving security in the presence of adaptive corruptions remains an open problem, and it is unclear if it requires further assumptions.
- **Further analysis of similar transforms**: The transformations we present make no claims of being the most efficient or optimal. Further research could be conducted to construct transformations with stronger security guarantees or that require fewer assumptions.

References

[AAN+22] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 815–844. Springer, Cham, May / June 2022.

[ABL25] Martin R. Albrecht, Benjamin Benčina, and Russell W. F. Lai. Hollow LWE: A new spin - unbounded updatable encryption from LWE and PCE. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part VIII*, volume 15608 of *LNCS*, pages 363–392. Springer, Cham, May 2025.

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Cham, August 2020.

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 261–290. Springer, Cham, November 2020.

[AFM24] Joël Alwen, Georg Fuchsbauer, and Marta Mularczyk. Updatable public-key encryption, revisited. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part VII*, volume 14657 of *LNCS*, pages 346–376. Springer, Cham, May 2024.

[AFMR25] Joël Alwen, Georg Fuchsbauer, Marta Mularczyk, and Doreen Riepel. Lattice-based updatable public-key encryption for group messaging. Cryptology ePrint Archive, Report 2025/365, 2025.

[AW25] Kyoichi Asano and Yohei Watanabe. Updatable public key encryption with strong CCA security: Security analysis and efficient generic construction. In Arpita Patra, editor, *CT-RSA 2025*, volume 15598 of *LNCS*, pages 223–246. Springer, Cham, April / May 2025.

[BBR+23] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[CS03] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[DKW21] Yevgeniy Dodis, Harish Karthikeyan, and Daniel Wichs. Updatable public key encryption in the standard model. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 254–285. Springer, Cham, November 2021.

[EJKM22] Edward Eaton, David Jao, Chelsea Komlo, and Youcef Mokrani. Towards post-quantum key-updatable public-key encryption via supersingular isogenies. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 461–482. Springer, Cham, September / October 2022.

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, January 2013.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Cham, November 2017.

[HLP22] Calvin Abou Haidar, Benoît Libert, and Alain Passelègue. Updatable public key encryption from DCR: Efficient constructions with stronger security. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 11–22. ACM Press, November 2022.

[HPS23] Calvin Abou Haidar, Alain Passelègue, and Damien Stehlé. Efficient updatable public-key encryption from lattices. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part V*, volume 14442 of *LNCS*, pages 342–373. Springer, Singapore, December 2023.

[JMM19] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188. Springer, Cham, May 2019.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.