A preliminary version of this work appears in the EUROCRYPT 2023 proceedings (© IACR 2023). This is the full version.

Let Attackers Program Ideal Models: Modularity and Composability for Adaptive Compromise

Joseph Jaeger[1]

May 12, 2024

Abstract

We show that the adaptive compromise security definitions of Jaeger and Tyagi (Crypto '20) cannot be applied in several natural use-cases. These include proving multi-user security from single-user security, the security of the cascade PRF, and the security of schemes sharing the same ideal primitive. We provide new variants of the definitions and show that they resolve these issues with composition. Extending these definitions to the asymmetric settings, we establish the security of the modular KEM/DEM and Fujisaki-Okamoto approaches to public key encryption in the full adaptive compromise setting. This allows instantiations which are more efficient and standard than prior constructions.

[1] School of Cybersecurity and Privacy, Georgia Tech, Atlanta, GA, USA. Email: `josephjaeger@gatech.edu`. URL: `https://cc.gatech.edu/~josephjaeger/`.

Contents

1 Introduction

Definitions lie at the heart of modern cryptography. They allow us to mathematically specify what should be achieved by a scheme in practice and give modular, proof-based analyses to ensure these properties are achieved. Studying and understanding definitions is fundamental to the field of cryptography.

There are multiple desiderata to consider when giving a security definition for a primitive including: (i.) Is it philosophically sound? Does it meaningfully model the uses and goals of a primitive in the real world? (ii.) Is it sufficiently strong? Can we prove that this security notion will imply security of higher-level protocols constructed from the primitive? (iii.) Is it sufficiently weak? Can we prove that schemes which "should be" secure satisfy the definition?[1]

In this work, we consider a set of definitions recently introduced by Jaeger and Tyagi [JT20] for the security of encryption schemes and pseudorandom functions in the "adaptive compromise" setting. They gave several examples of schemes achieving their definitions as well as higher-level protocols which can be proven secure based on sub-primitives achieving their definitions, thereby evidencing that their definitions achieve desiderata (ii.) and (iii.). We provide counter-evidence. There are natural goals and constructions for which their definitions fail (ii.) and (iii.).[2]

As an example, it does not seem to be possible to prove that single-user restrictions of their definitions implies the full multi-user versions. Across a wide variety of definitions, the notion of multi-user security that is considered "correct" follows from single-user security by a straightforward hybrid argument. Thus, whether this holds for a definition might be considered a sort of litmus test. A definition for which this is not possible should be examined carefully to understand why. Having done so, we propose new variants of Jaeger and Tyagi's definitions and show that they resolve these shortcomings, while preserving the positive qualities of the original definitions.

1.1 Adaptive Compromise and SIM-AC security

Before discussing our contributions, let us first briefly recall the adaptive compromise setting broadly and the specific SIM-AC definitions of Jaeger and Tyagi (<u>sim</u>ulation security under <u>a</u>daptive <u>c</u>ompromise). Roughly speaking, the adaptive compromise setting captures times when there are multiple users of a system, each of whom have their own secrets. An attacker then interacts with these users and based on these interactions may adaptively decide to steal some of the secrets. In applications of these definitions, this description may be somewhat metaphorical. For example, in the searchable encryption scheme of CJJJKR [CJJ+14] the "users" are keywords, each of which are assigned a secret key. The "stealing" of keys occurs because to perform a search for a particular keyword, the protocol shares the keyword's secret key. The adaptive compromise setting is widely studied in cryptography and is associated with a variety of terms including (but not limited to) adaptive corruption/compromise/security [JT20, Pan07], non-committing encryption[3] [CFGN96, Nie02, CDG+18, CLNS17], and selective-opening attacks [BDWY12, BHK12, BHY09, HPW15, HP16, HRW16]. We refer the reader to a systemization of knowledge by Brunetta,

[1]More nuanced versions of (ii.) and (iii.) ask not just whether these proofs are possible, but also how easy they are to write. Definitions which are difficult to work with can result in proof errors or cryptographers only loosely sketching their proofs (potentially hiding errors).

[2]The examples for which (iii.) fails are "intermediate-level" proofs where both the assumption and desired result use their definitions. Arguably then, it is only with respect to (ii.) that these definitions have issues.

[3]This notion of "non-committing" encryption (exploiting ideal model programming) should not be confused as being the opposite of "committing" encryption notions which require that ciphertexts act as commitments to messages. Schemes can simultaneously be non-committing (for simulators that program the ideal model) and committing (for attackers that can only query the ideal model).

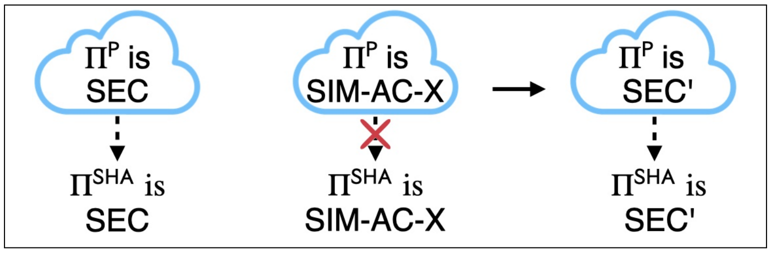

Figure 1: **Left:** Typically one proves a scheme Π achieves a security notion with random oracle P, then heuristically assumes it is SEC secure with a particular hash function (e.g. SHA-384). **Right:** A scheme Π *cannot* be SIM-AC-X secure with any standard model hash function [JT20, Nie02]. Instead, one uses SIM-AC-X security of Π^{P} as an intermediate step to showing that Π^{P} achieves some security notion SEC$'$. Then one heuristically assumes Π^{SHA} is SEC$'$ secure.

Heum, and Stam [BHS24] for an in-depth synthesis of the literature on adaptive compromise for public key encryption.

Jaeger and Tyagi's work was motivated by various papers that ran into adaptive compromise issues for symmetric encryption or PRFs and had addressed the issues by fixing particular uses of random oracles acting like PRFs. They observed that these works all technically required the same detail-intensive random oracle analysis (which was usually omitted or incorrect). To address this, they introduced their SIM-AC definitions which allow one to abstract away this detail-intensive analysis as something that need only be done once at the lowest levels of analysis. They showed that these notions were achieved by standard efficient schemes in appropriate ideal models, and sufficed for proving the security of their motivating higher-level applications. Broadly their definitions were online-simulator based definitions in which the attacker tries to distinguish between a real world where they interact with the honest algorithms of the scheme and an ideal world where the simulator provides responses for every oracle query (including ideal primitive queries). Security requires that for every adversary there is a simulator whose responses it cannot distinguish from the real world.

Ideal model interpretation. Notably these definitions were defined explicitly for use only with ideal primitives because techniques of Nielsen [Nie02] show that such definitions cannot be achieved in the standard model. Arguably this causes issues with desiderata (i.). Consider a scheme Π^{H} which expects access to a hash function H. In practice, the might be deployed with the hash function SHA-384 (giving Π^{SHA}) under the hope that it achieves some security notion SEC. Towards justifying this, the scheme may be analyzed when the hash function is replaced with a random oracle P (giving Π^{P}). If Π^{P} is shown to be SEC secure, this may be taken as heuristic evidence that Π^{SHA} will be SEC secure. However, this clearly cannot be the case for SEC=SIM-AC-X from the aforementioned result that SIM-AC notions cannot be achieved in the standard model.

From our perspective, the "correct" interpretation is that the SIM-AC definitions are intentionally chosen to be overly strong so that (in ideal models) they imply any other security property SEC$'$ one desires. Suppose SEC$'$ is plausibly achievable in the standard model and one proves that SIM-AC-X security implies SEC$'$ security. Then a proof that Π^{P} is SIM-AC-X secure can be viewed as part of a longer ideal model proof that it achieves SEC$'$. Then the proof acts as heuristic evidence that Π^{SHA} is a standard model scheme achieving SEC$'$. We represent this pictorially in Fig. 1.

A similar viewpoint can be taken to proving that a particular hash function construction is indifferentiable from a random oracle. It is trivial to show that no standard model hash function can achieve this. However, analyzing indifferentiability in ideal models still serves as a convenient intermediate notion for heuristically justifying the use of the hash function in some contexts.

A note on terminology. As noted earlier, there are a variety of terms used for the general setting we consider. We use "adaptive compromise" to match the SIM-AC definitions we are directly building. The terminology "adaptive corruption", "adaptive security", or "selective-opening security" could just as easily have been used. We choose to explicitly avoid the "non-committing encryption" terminology, to avoid confusion with definitions of "committing encryption" that require a ciphertext "commit" to the underlying message. It is possible for a scheme to simultaneously be "non-committing" to a simulator which can program the ideal model, but "committing" to an adversary that does not program the ideal model.

1.2 Our results

Shortcomings of SIM-AC. After introducing notations and other preliminaries, we start in Section 3 by recalling the original SIM-AC definitions of Jaeger and Tyagi. In their definitions, an attacker interacts with either a real world (where oracles are instantiated honestly) or an ideal world (where oracles are all simulated by a simulator given only some leakage about the queries being made). The definitions are multi-user and allow the attack to ask that a particular users secrets be revealed at any time. Then, in the ideal world, the simulator is given all of the suppressed information about prior queries and must produce a consistent key, lest it be discovered. In the ideal world, the simulator completely controls the responses of the ideal primitive.

We evidence some shortcomings of these definitions, in that they are seemingly unable to prove some very natural results.[4] One example, which came up in their own work, is that their definitions cannot be used for proofs wherein the ideal object is used multiple times within a protocol (whether by multiple different sub-primitives or repeated use of the same sub-primitive). For example, in the searchable encryption construction of CJJJKR [CJJ+14] the same random oracle was shared across encryption and a PRF, but for the analysis done by Jaeger and Tyagi they were forced to use different primitives for the two uses. One can generically solve this problem via oracle cloning [BDG20], but we find this unsatisfactory. A good definitional framework should allow us to capture when uses of ideal primitives don't require domain separation techniques. Furthermore, while domain separation is relatively fast and efficient for random oracles, we are generally interested in the use of a variety of ideal primitives and it is much less clear how to do oracle cloning efficiently with something like an ideal cipher.

Similar and even more subtle issues arise in some "standard" results that one would expect to hold with a "good" definition. One would expect that it should be possible to prove secure the cascade construction of a PRF [BCK96b, GGM86] which iteratively applies a smaller PRF, as well as to prove that for most security notions single-user security implies multi-user security. The cascade construction underlies several other construction PRFs including AMAC, HMAC, and NMAC [BBT16, BCK96a, Bel06]. These (and other) issues all stem from a common cause. In SIM-AC, the simulator completely controls and replaces the ideal primitive. As such the definition is not robust to proofs which require multiple different applications of security with respect to the same ideal primitive.

New definition, SIM*-AC. Motivated by these shortcomings, in Section 4 we propose new variants of these definitions, which we term SIM*-AC. Our new definitions match the prior SIM-

[4] We use "seemingly" here and similar phrasing elsewhere because, while we have deeply considered these problems and do not see how SIM-AC could be used to prove these results, we do not have any explicit counterexamples showing it is impossible.

AC definitions, but make three crucial modifications. The first is that rather having complete control of the ideal primitive, we give the simulator access to an oracle for querying the primitive and which additionally provides the special power of being able to give an input-output pair which the primitive will program itself to be consistent with, if possible. This modification means that applications of SIM*-AC in a proof will leave the ideal primitive around for use in further proof steps. However, these future steps can run into issues where the simulator is supposed to have programmed the ideal primitive, but a reduction attacker who wants to run the simulator internally has no way of forcing other parties to use a programmed ideal primitive. This issue is resolved by our second modification which *gives the adversary the ability to program the ideal primitive*. The final modification is aimed at proofs which require a polynomial number of hybrids and, as such, the reduction adversary needs to depend on the simulator so that it can properly simulate internal hybrids. We simply reverse the order of quantification so that a universal simulator is quantified before a specific attacker.

After the introduction of the new definitions we show by example that the modifications suffice to write the proofs we identified as seemingly not possible with the original SIM-AC definitions. Namely, we prove that for all of our SIM*-AC definitions (with one exception) single-user security implies multi-user security[5] and that the cascade construction of a large-domain PRF from a small-domain PRF is secure. Both proofs are hybrid arguments which conceptually resemble such proofs for most standard indistinguishability-based security notions. For going from single-user to multi-user security the hybrid is over how many of the users will be honestly run versus emulated by a copy of the single-user simulator. For the cascade construction (which is a generalization of the GGM construction of a PRF from a PRG), we think of there being an underlying tree structure imposed on the internal values of the computation. The proof performs a hybrid over how many layers of the tree are honestly run versus emulated by a multi-user simulator for the underlying PRF. Using multi-user security allows us to hybrid one layer at a time, rather than having hybrid over each node individually.

Asymmetric encryption. The SIM-AC definitions focus on symmetric primitives (encryption and PRFs) because this is what was required by their applications. However, adaptive compromise has been studied in detail for public-key encryption, so it is natural to ask how a SIM*-AC notion for public key encryption would work. We do so in Section 5, providing a definition that captures the compromise of receiver secret decryption keys and sender randomness. The resulting definition roughly matches the SIM-FULL definition of Camensich, Lehmann, Neven, and Samelin [CLNS17].[6] In their work, they showed that SIM-FULL was stronger than various prior adaptive compromise definitions [CLNS16, HPW15] and equivalent to a new universal composability definition they introduce.

Casting this definition in SIM*-AC language provides benefits. Where CLNS constructed one particular secure encryption scheme from one-way trapdoor permutations, the broader context of SIM*-AC style definitions allows us to follow the example of Jaeger and Tyagi by giving modular analysis. In particular, we introduce SIM*-AC definitions for key-encapsulation mechanisms (KEM), then show the KEM/DEM approach [CS03] allows one to combine a KEM with a symmetric encryption scheme to construct public-key encryption. We consider one Fujisaki-Okamoto-style transformation [FO13] (as modularized by Hofheinz, Hövelmanns, and Kiltz [HHK17]) to show that it can lift a KEM satisfying a one-wayness security notion to a KEM satisfying our full SIM*-AC-CCA notion. Thereby we have a more general collection of different options how to construct a public-key encryption scheme secure against adaptive compromise. We can instantiate this with

[5]The exception is key-private security which is meaningless with only a single user.

[6]Their definition is basically a SIM-AC-CCA (not SIM*-AC) definition with labels and using a random oracle.

well-studied and standardized schemes, improving efficiency because our analysis allows the use of block-cipher based symmetric encryption for the DEM.

An interesting comparison point for our KEM/DEM analysis is the work of Heuer and Poettering [HP16] who also looked at the KEM/DEM construction. They proved a weaker offline-simulation notion of security for public key encryption by making a particular concrete assumption about the DEM being constructed from a blockcipher and having a particular simulatable form.

Heuer, Jager, Kiltz, and Schäge [HJKS15] also showed adaptive compromise security of a Fujisaki-Okamoto-style transformation. Their result proves a weaker "offline-simulation" style of security for a public-key encryption scheme obtained by starting with a one-way secure public-key encryption scheme, applying a Fujisaki-Okamoto-style transformation, then applying the KEM/DEM transform with a specific DEM based on one-time pad.

New definition, old results. Jaeger and Tyagi showed a number positive results in their original work. These include that random oracles and ideal ciphers make SIM-AC-PRF secure function families, that various constructions of symmetric encryption achieve SIM-AC security when their underlying function families are SIM-AC-PRF secure, and that higher-level protocols can be proven secure assuming the SIM-AC security of their constituent elements. It would be rather disappointing if our switch to SIM*-AC security required us to re-prove all of these results from scratch.

In Section 6, we dedicate the end of our paper to showing that these results hold with SIM*-AC security. We roughly divide these pre-existing results into three categories: low-level results (constructing basic SIM-AC primitives directly from ideal primitives), intermediate-level results (using one notion of SIM-AC to achieve another), and high-level results (proving secure some non-SIM-AC protocol). For each we discuss how the existing result can be seen, possibly with minor modification to the proof, to hold for SIM*-AC security. In some cases we can get minor improvements along the way, such as allowing the proof to handle when a single ideal primitive is shared between multiple schemes.

2 Preliminaries

Pseudocode notation. We define security notions using pseudocode-based games. The pseudocode "Require bool" is shorthand for "If $\neg\mathsf{bool}$ then return $\bot$". Here $\neg$ denotes negation. If S is a set, then $x \leftarrow_{\$} S$ sets x equal to a uniformly random element of S. The notation $x_{(\cdot)} \leftarrow_{\$} S$ means that each x_u will be sampled according to $x_u \leftarrow_{\$} S$ the first time it is accessed.

The notation $y \leftarrow_{\$} A(x_1, x_2, \cdots : \sigma)$ denotes the (randomized) execution of A with state σ. Deterministic execution uses $\leftarrow$. The state σ is passed by reference, so changes that A makes to σ are maintained after A's execution. All other inputs are passed by value. For given $x_1, x_2, \ldots$ and σ we let $[A(x_1, x_2, \cdots : \sigma)]$ denote the set of possible outputs of A given these inputs.

The symbol $\bot$ is used to indicate rejection or uninitialized variables. The symbol $\diamond$ is used as a return value by functions that do not need to return anything. Unless specified otherwise, these values are assumed not to be contained in sets. Algorithms and oracles will typically assume their input is from a particular domain (e.g. the message space of an encryption scheme). We implicitly assume adversaries never provide them with input not in these domains.

A list T of length $n \in \mathbb{N}$ specifies an ordered sequence of elements $T[1]$, $T[2]$, $\ldots$, $T[n]$. The operation $T.\mathsf{add}(x)$ appends x to this list by setting $T[n+1] \leftarrow x$, so T is now of length $n+1$. We let $|T|$ denote the length of T. In pseudocode lists are assumed to be initialized empty (i.e. have length 0). An empty list or table is denoted by $[\cdot]$. We sometimes use set notation with a list. For example, $x \in T$ is true if $x = T[i]$ for any $1 \leq i \leq |T|$. The loop "For $x \in T$" is defined to be looping "For $i = 1, \ldots, |T|$" and defining $x \leftarrow T[i]$ in each iteration.

If T is a list of tuples (x, y) then we index into T like a table where $T\langle x\rangle$ is the y value of the last tuple in the list with first component x (or is $\perp$ if no such tuple exists). By $T.\mathsf{add}(x, y)$ we mean $T.\mathsf{add}((x, y))$.

We use an asymptotic formalism with security parameter λ. A function f is negligible if for all polynomials p there exists a $\lambda_p \in \mathbb{N}$ such that $f(\lambda) \leq 1/p(\lambda)$ for all $\lambda \geq \lambda_p$. We say it is super-polynomial if $1/f$ is negligible and super-logarithmic if 2^f is super-polynomial.

Suppose $\mathrm{G}^{\mathsf{sec}}_x$ is a game that samples a uniformly random bit b, runs an adversary which guesses bit b', and then returns the boolean $(b = b')$. Then for $d \in \{0, 1\}$, we let $\mathrm{G}^{\mathsf{sec}}_{x,d}$ be the game with b hardcoded to have value d and which outputs the boolean $(b' = 1)$. Standard conditional probability calculations give that $2\Pr[\mathrm{G}^{\mathsf{sec}}_x] - 1 = \Pr[\mathrm{G}^{\mathsf{sec}}_{x,1}] - \Pr[\mathrm{G}^{\mathsf{sec}}_{x,0}]$.

Ideal primitives. Most of the definitions we consider are dependent on ideal primitives such as random oracles or ideal ciphers, so we require a careful formalization of them. An ideal primitive P specifies (for each $\lambda \in \mathbb{N}$) a distibution $\mathcal{P}_\lambda$ over functions $f : \mathcal{K}_\lambda \times \mathcal{D}_\lambda \to \mathcal{R}_\lambda$. When needed to avoid ambiguity we write $\mathsf{P}.\mathcal{P}_\lambda$, $\mathsf{P}.\mathcal{K}_\lambda$, $\mathsf{P}.\mathcal{D}_\lambda$, and $\mathsf{P}.\mathcal{R}_\lambda$. In the P ideal model, $f \leftarrow\!\!\$\, \mathcal{P}_\lambda$ is sampled at the beginning of any security game and algorithms are given oracle access to f.

It is often important that oracle access to an ideal primitive can be efficiently simulated despite the fact that each $f \in \mathcal{P}_\lambda$ is typically exponential in size. This is referred to as lazy sampling, which we notate using an algorithm $\mathsf{P.Ls}$. We will think of f as being (partially) specified by a table σ_{P} indexed by $\mathcal{K}_\lambda \times \mathcal{D}_\lambda$. Then the evaluation algorithm has syntax $y \leftarrow\!\!\$\, \mathsf{P.Ls}(1^\lambda, k, x : \sigma_{\mathsf{P}})$. If $\sigma_{\mathsf{P}}[k, x] = \perp$, it samples $\sigma_{\mathsf{P}}[k, x]$ according to the appropriate distribution conditioned on the current value of σ_{P}.[7] Then it outputs $\sigma_{\mathsf{P}}[k, x]$. We sometimes use A^{P} as shorthand for giving algorithm A oracle access to $\mathsf{P.Ls}(1^\lambda, \cdot, \cdot : \sigma_{\mathsf{P}})$. We write $\sigma_{\mathsf{P}} \leftarrow\!\!\$\, \mathsf{P.Init}$ for initializing this state (i.e., setting $\sigma_{\mathsf{P}} \leftarrow [\cdot, \cdot]$).

The standard model is captured by the primitive P_{sm} for which $\mathcal{P}_\lambda$ always returns the function f defined exactly by $f(\varepsilon, \varepsilon) = \varepsilon$. A random oracle $\mathsf{P}_{\mathsf{rom}}$ is captured by $\mathcal{P}_\lambda$'s output being uniform over the set of all functions $f : \mathcal{K}_\lambda \times \mathcal{D}_\lambda \to \mathcal{R}_\lambda$. An ideal injection $\mathsf{P}_{\mathsf{inj}}$ is captured by letting $\mathcal{D}_\lambda$ consist of tuples $(\circ, x)$ for $\circ \in \{+, -\}$. Then $\mathcal{P}_\lambda$ returns a uniform f for which $f(k, (+, \cdot))$ is an injection with inverse $f(k, (-, \cdot))$ (we define inverse functions to output $\diamond$ on input a value not in the image of the original function). An ideal cipher $\mathsf{P}_{\mathsf{icm}}$ is an ideal injection for which $f(k, (+, \cdot))$ is a bijection on the finite set $\mathcal{R}_\lambda$. Standard techniques allow Ls to be efficiently evaluated for such functions.

Cryptographic schemes may be constructed from multiple underlying cryptographic schemes, each expecting its own ideal primitive. Let P' and P'' be ideal primitives. We define $\mathsf{P} = \mathsf{P}' \times \mathsf{P}''$ via the following algorithms.

```
$\underline{\mathsf{P.Init}(1^\lambda)}$
$\sigma_{\mathsf{P}}' \leftarrow\!\!\$\, \mathsf{P}'.\mathsf{Init}(1^\lambda)$
$\sigma_{\mathsf{P}}'' \leftarrow\!\!\$\, \mathsf{P}''.\mathsf{Init}(1^\lambda)$
Return $(\sigma_{\mathsf{P}}', \sigma_{\mathsf{P}}'')$
```

```
$\underline{\mathsf{P.Ls}(1^\lambda, k, x : \sigma_{\mathsf{P}})}$
$(\sigma_{\mathsf{P}}', \sigma_{\mathsf{P}}'') \leftarrow \sigma_{\mathsf{P}}$
$(d, k) \leftarrow k$
If $d = 1$ then $y \leftarrow\!\!\$\, \mathsf{P}'.\mathsf{Ls}(1^\lambda, k, x : \sigma_{\mathsf{P}}')$
If $d = 2$ then $y \leftarrow\!\!\$\, \mathsf{P}''.\mathsf{Ls}(1^\lambda, k, x : \sigma_{\mathsf{P}}'')$
$\sigma_{\mathsf{P}} \leftarrow (\sigma_{\mathsf{P}}', \sigma_{\mathsf{P}}'')$
Return $y$
```

In other words, $\mathsf{P}.\mathcal{P}_\lambda$ samples $f' \leftarrow\!\!\$\, \mathsf{P}'.\mathcal{P}_\lambda$ and $f'' \leftarrow\!\!\$\, \mathsf{P}''.\mathcal{P}_\lambda$, then defines f by $f((1, k), x) = f'(k, x)$ and $f((2, k), x) = f''(k, x)$.

[7] Concretely, this is the distribution induced by sampling $f \leftarrow\!\!\$\, \mathcal{P}_\lambda$ subject to $f(k', x') = \sigma[k', x']$ wherever the latter is not $\perp$ and assigning $\sigma_{\mathsf{P}}[k, x] \leftarrow f(k, x)$.

Programming ideal primitives. For our new security notions we need to make explicit a notion of "programming" an ideal model. By this we mean allowing some third party to define the output of ideal model on inputs that have not previously been queried. Let σ_{P} be a table indexed by $\mathcal{K}_\lambda \times \mathcal{D}_\lambda$ and let $(k, x, y) \in \mathcal{K}_\lambda \times \mathcal{D}_\lambda \times \mathcal{R}_\lambda$. We say that σ_{P} is compatible with (k, x, y), denoted $\sigma_{\mathsf{P}}\heartsuit(k, x, y)$, if there exists $f \in \mathcal{P}_\lambda$ such that (i) $\sigma_{\mathsf{P}}[k', x'] = f(k', x')$ wherever $\sigma_{\mathsf{P}}[k', x'] \neq \perp$ and (ii) $f(k, x) = y$. Then we allow programming of an ideal model P using the algorithm $\mathsf{P.Prog}$ defined as follows.

```
$\underline{\mathsf{P.Prog}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})}$
If $\sigma_{\mathsf{P}}\heartsuit(k, x, y)$ then $\sigma_{\mathsf{P}}[k, x] \leftarrow y$
Return $\diamond$
```

This ensures that P cannot be redefined on an input where it was already defined and that an ideal injection cannot be made to have inconsistent inverses.

Our careful formalizing of ideal primitives in terms of functions, particularly in requiring that $\mathsf{P.Prog}$ maintain consistency, is important for avoiding subtle issues in later proofs. This formalization ensures that a deterministic algorithm with oracle access to P always gives consistent outputs even if P is programmed between executions. Correctness of a scheme with access to P (e.g. that decryption inverts encryption) is maintained even if P is programmed between executions of different algorithms. Without these properties it would be difficult to avoid erroneous proofs that implicitly assumed them during typically "straightforward" proof steps.

This is not without cost. The requirement for consistency in programming has the potential to introduce subtle errors elsewhere in proofs by implicitly assuming an attempt to program an oracle worked, when in fact it failed because of inconsistency. Additionally, the act of honestly querying the ideal primitive can be detected by a programming adversary who attempts to program at that point and then checks if they succeed in this programming. We believe this cost to be worthwhile because in the analyses we have considered, the places that could cause such proof errors would anyway need to be analyzed carefully to avoid other errors if we were using a more permissive notion of programming.[8]

For generality, we allow the use of non-programmable ideal primitives in games that allow programming. This is captured by defining $\mathsf{P.Prog}$ to immediately return $\diamond$. When we quantify over an arbitrary ideal primitive, we allow it to be programmable or non-programmable (or the combination of multiple ideal primitives – some programmable, some not). When we discuss a specific ideal primitive, we mean the programmable version unless specified otherwise.

Syntax for cryptographic primitives. We assume familiarity with (randomized) symmetric encryption, asymmetric encryption, function families (e.g. PRFs), and key encapsulation mechanisms. We use the following syntax.

Symmetric encryption	Asymmetric encryption
$k \leftarrow\$ \mathsf{SE.Kg}(1^\lambda)$	$(ek, dk) \leftarrow\$ \mathsf{PKE.Kg}(1^\lambda)$
$c \leftarrow\$ \mathsf{SE.Enc}^{\mathsf{P}}(1^\lambda, k, m)$	$c \leftarrow\$ \mathsf{PKE.Enc}^{\mathsf{P}}(1^\lambda, ek, m)$
$m \leftarrow \mathsf{SE.Dec}^{\mathsf{P}}(1^\lambda, k, c)$	$m \leftarrow \mathsf{PKE.Dec}^{\mathsf{P}}(1^\lambda, dk, c)$

Function Family	Key Encapsulation Mechanism
$k \leftarrow\$ \mathsf{F.Kg}^{\mathsf{P}}(1^\lambda)$	$(ek, dk) \leftarrow\$ \mathsf{KEM.Kg}(1^\lambda)$
$y \leftarrow \mathsf{F.Ev}^{\mathsf{P}}(1^\lambda, k, x)$	$(c, k) \leftarrow\$ \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek)$
$x \leftarrow \mathsf{F.Inv}^{\mathsf{P}}(1^\lambda, k, y)$	$k \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk, c)$

[8]Subsequent work by Cheng and Jaeger [CJ24] formalizes "consistency" and "undetectability" of ideal primitives, showing that these two properties are fundamentally at odds and identifying a proof which is seemingly only possible with an undetectable ideal primitive. More work is needed to understand the tradeoffs between these properties.

A family of functions F only has inverse algorithm $\mathsf{F.Inv}$ if it is a blockcipher. For simplicity, we assume perfect correctness which holds for all $f \in \mathsf{P}.\mathcal{P}_\lambda$. We will make careful note of where proofs make use of this correctness. To use notions of imperfect correctness in these proofs, one must choose an imperfect correctness notion that is "robust" to the ideal primitive being programmable.

We additionally will sometimes assume a notion we call *query consistency* which requires that if c is produced by encryption/encapsulation/evaluation, then decrypting/decapsulating/inverting c with the correct key only makes ideal primitive queries that were also made by encryption/encapsulation/evaluation. This ensures that any querying of the ideal primitive while decrypting/decapsulating an honest ciphertext cannot be detected by a programming adversary.

3 SIM-AC Definitions and Their Shortcomings

We start by recalling the definitions that Jaeger and Tyagi [JT20] introduced for the simulation security of symmetric encryption or pseudorandom functions under adaptive compromise. Jaeger and Tyagi showed that these definition were achieved by very natural encryption/PRF constructions in the random oracle or ideal cipher model and that they moreover sufficed for proving the security of higher-level constructions (e.g. searchable encryption schemes, asymmetric password-authenticated key exchange, and self-revocable encrypted cloud storage). In this section, we will identify ways in which these definitions fall short. Namely, that there are other natural encryption/PRF constructions and high-level construction which cannot be proven secure using these definitions.[9]

3.1 SIM-AC Definitions

All of the SIM-AC definitions have a common structure; they measure the ability of an adversary to distinguish between a "real" and a "simulated" world. In the real world, the adversary interacts with multiple "users" that honestly execute the algorithms of scheme. The adversary has access to an exposure oracle which it can query to be given the secret keys of any users it chooses. Finally, the adversary has oracle access to the ideal primitive algorithm $\mathsf{P.Ls}$. In the ideal world, the output of *all* of these oracles is provided instead by a simulator S. For the definition to be meaningful, the behavior of the simulator when responding to queries for "unexposed" users is restricted in some manner. (For example, the simulator may be required to return a uniformly random string or may only be given partial information about what the query was.)

Pseudorandom function security. We start with the notion of SIM-AC-PRF security for a function family F. It is captured by the game $\mathrm{G}^{\mathsf{sim\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}$ shown in Fig. 2. The variable X is used to track which users have been exposed, so X_u is true when the user has been exposed. The game hardcodes that random values are returned for evaluation queries to unexposed users in the simulated world. Inputs and outputs to evaluation are stored in the table T_u which is given to S when u is exposed. Note that here (and in other SIM-AC definitions) the superscript P given to oracles of F indicates that it had direct oracle access to $\mathsf{P.Ls}$. If instead it had access to $\textsc{Prim}$ we would have to modify the game when $b = 0$ so that honest algorithms of F are not run. Otherwise S would see the oracle queries made by F, which is clearly not intended.

We define $\mathsf{Adv}^{\mathsf{sim\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}(\lambda) = 2\Pr[\mathrm{G}^{\mathsf{sim\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}(\lambda)] - 1$ and say that F is SIM-AC-PRF secure with P if for all PPT $\mathcal{A}_{\mathsf{prf}}$ there exists a PPT S such that $\mathsf{Adv}^{\mathsf{sim\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}(\cdot)$ is negligible. Intuitively, this definition captures that the outputs of F_k look random to an adversary until they expose k.

[9]Technically, we do not show that these proofs are impossible. We show why the "natural" proofs fail and informally argue why it seems difficult to find other proofs.

```
Game $\mathrm{G}^{\mathsf{sim\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}(\lambda)$
    $k_{(\cdot)} \leftarrow\!\!\$\ \mathsf{F.Kg}(1^\lambda)$
    $\sigma_{\mathsf{P}} \leftarrow\!\!\$\ \mathsf{P.Init}(1^\lambda)$
    $\sigma \leftarrow\!\!\$\ \mathsf{S.Init}(1^\lambda)$
    $b \leftarrow\!\!\$\ \{0,1\}$
    $b' \leftarrow\!\!\$\ \mathcal{A}_{\mathsf{prf}}^{\textsc{Ev},\textsc{Exp},\textsc{Prim}}(1^\lambda)$
    Return $(b = b')$

$\textsc{Prim}(k, x)$
    $y_1 \leftarrow\!\!\$\ \mathsf{P.Ls}(1^\lambda, k, x : \sigma_{\mathsf{P}})$
    $y_0 \leftarrow\!\!\$\ \mathsf{S.Ls}(1^\lambda, k, x : \sigma)$
    Return $y_b$

$\textsc{Ev}(u, x)$
    If $T_u[x] \neq \perp$ then return $T_u[x]$
    $y_1 \leftarrow \mathsf{F.Ev}^{\mathsf{P}}(1^\lambda, k_u, x)$
    If $X_u$ then $y_0 \leftarrow\!\!\$\ \mathsf{S.Ev}(1^\lambda, u, x : \sigma)$
    Else $y_0 \leftarrow\!\!\$\ \mathsf{F.Out}(\lambda)$
    $T_u[x] \leftarrow y_b$
    Return $y_b$

$\textsc{Exp}(u)$
    $k'_1 \leftarrow k_u$
    $k'_0 \leftarrow\!\!\$\ \mathsf{S.Exp}(1^\lambda, u, T_u : \sigma)$
    $X_u \leftarrow \mathsf{true}$
    Return $k'_b$
```

```
Game $\mathrm{G}^{\mathsf{sim\text{-}ac\text{-}cca}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda)$
    $k_{(\cdot)} \leftarrow\!\!\$\ \mathsf{SE.Kg}(1^\lambda)$
    $\sigma_{\mathsf{P}} \leftarrow\!\!\$\ \mathsf{P.Init}(1^\lambda)$
    $\sigma \leftarrow\!\!\$\ \mathsf{S.Init}(1^\lambda)$
    $b \leftarrow\!\!\$\ \{0,1\}$
    $b' \leftarrow\!\!\$\ \mathcal{A}_{\mathsf{cca}}^{\textsc{Enc},\textsc{Dec},\textsc{Exp},\textsc{Prim}}(1^\lambda)$
    Return $(b = b')$

$\textsc{Prim}(k, x)$
    $y_1 \leftarrow\!\!\$\ \mathsf{P.Ls}(1^\lambda, k, x : \sigma_{\mathsf{P}})$
    $y_0 \leftarrow\!\!\$\ \mathsf{S.Ls}(1^\lambda, k, x : \sigma)$
    Return $y_b$

$\textsc{Enc}(u, m)$
    If $\neg X_u$ then $\ell \leftarrow |m|$ else $\ell \leftarrow m$
    $c_1 \leftarrow\!\!\$\ \mathsf{SE.Enc}^{\mathsf{P}}(1^\lambda, k_u, m)$
    $c_0 \leftarrow\!\!\$\ \mathsf{S.Enc}(1^\lambda, u, \ell : \sigma)$
    $M_u.\mathsf{add}(c_b, m)$; Return $c_b$

$\textsc{Dec}(u, c)$
    If $M_u\langle c\rangle \neq \perp$ then return $M_u\langle c\rangle$
    $m_1 \leftarrow \mathsf{SE.Dec}^{\mathsf{P}}(1^\lambda, k_u, c)$
    $m_0 \leftarrow\!\!\$\ \mathsf{S.Dec}(1^\lambda, u, c : \sigma)$
    Return $m_b$

$\textsc{Exp}(u)$
    $k'_1 \leftarrow k_u$; $k'_0 \leftarrow\!\!\$\ \mathsf{S.Exp}(1^\lambda, u, M_u : \sigma)$
    $X_u \leftarrow \mathsf{true}$; Return $k'_b$
```

Figure 2: Games defining SIM-AC-PRF security of F and SIM-AC-CCA security of SE

Encryption definitions. Next we recall the SIM-AC security notions for a symmetric encryption scheme SE. Consider the game $\mathrm{G}^{\mathsf{sim\text{-}ac\text{-}cca}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda)$ shown in Fig. 2. During encryption queries for unexposed users, the simulator is only told the length of the message m. The list M_u stores the messages queried to user u and ciphertexts returned. It is given to the simulator when that user is exposed. If the attacker forwards challenge ciphertexts from encryption to decryption, this list is used to respond appropriately.

We define $\mathsf{Adv}^{\mathsf{sim\text{-}ac\text{-}cca}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda) = 2\Pr[\mathrm{G}^{\mathsf{sim\text{-}ac\text{-}cca}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda)] - 1$ and say SE is SIM-AC-CCA secure with P if for all PPT $\mathcal{A}_{\mathsf{cca}}$ there exists a PPT S such that $\mathsf{Adv}^{\mathsf{sim\text{-}ac\text{-}cca}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\cdot)$ is negligible. Intuitively, this definition captures that an adversary learns nothing (other than the length) about a message m encrypted with a key k until they expose k. For chosen-plaintext security we restrict attention to attackers that never query decryption. We then write the superscript $\mathsf{sim\text{-}ac\text{-}cpa}$.

Stronger notions of security are captured by requiring that S be chosen from some restricted set. Key-private security (SIM-AC-KP) requires that the CPA simulator respond to encryption queries for un-exposed users using an algorithm $\mathsf{S.Enc}_1(1^\lambda, \ell : \sigma)$ which *is not* given u as input. Indistinguishable from random security (SIM-AC-\$) requires that the CPA simulator respond to encryption queries for un-exposed users by sampling c uniformly from a set $\mathsf{S.Out}(\lambda, \ell)$. Authenticated encryption security (SIM-AC-AE) requires that the CCA simulator respond to encryption queries as in SIM-AC-\$ security and to decryption queries for un-exposed users with $\perp$.

Simplifying assumptions. Jaeger and Tyagi observed the following simplifying assumptions (copied almost verbatim from [JT20]) for their SIM-AC definitions.

- If an oracle is deterministic (and stateless) in the real world, we can assume that the adversary never repeats a query to this oracle or that the simulator always provides the same output to repeated queries.
- We can assume the adversary never makes a query to a user it has already exposed or that for such queries the simulator just runs the code of the real world (replacing calls to P with calls to $\mathsf{S.Ls}$).
- We can assume the adversary always queries with $u \in [u_\lambda] = \{1, 2, \ldots, u_\lambda\}$ for some polynomial $u_{(\cdot)}$ or that the simulator is agnostic to the particular strings used to reference users.

Looking ahead, we will be able to make the analogous assumptions for the new definitions introduced in this paper. These assumptions are convenient for proving that a scheme satisfies a given SIM-AC definition of security. The fact that these assumptions are not hardcoded into the security game is convenient when proving the security of a higher-level construction assuming that constituent schemes satisfy some SIM-AC security notion.

3.2 Shortcomings of SIM-AC

Now that we have introduced SIM-AC security notions we can discuss ways that they fall short of being able to establish the results we would like.

Multiple schemes with the same P. Suppose a higher-level protocol is constructed from multiple underlying schemes satisfying SIM-AC security notions. We generally will not be able to prove the security of the protocol if the underlying schemes make use of the same P.[10] Performing a SIM-AC reduction with the first scheme will replace the entirety of P with some $\mathsf{S.Ls}$. With P being gone, the security of the second scheme with respect to P is of no use.

As a toy example, we might consider function families F_0 and F_1. Even assuming they are both SIM-AC-PRF secure with P, it seems impossible to prove F is SIM-AC-PRF secure where $\mathsf{F.Ev}^{\mathsf{P}}(1^\lambda, (k_0, k_1), (b, x)) = \mathsf{F}_b.\mathsf{Ev}^{\mathsf{P}}(1^\lambda, k_b, x)$. Several of Jaeger and Tyagi's proofs were restricted by this and had to assume underlying schemes used distinct ideal primitives.

Multiple uses of the same scheme. Suppose a higher-level protocol is constructed from an underlying scheme satisfying a SIM-AC security notion and that this scheme is used in several distinct ways in the protocol.

If it's not possible to write a careful reduction that covers all of the uses of the scheme at once, then we run into a similar issue as the above. The first application of the scheme's SIM-AC security will replace its ideal primitive with a simulator, preventing us from applying its security again.

As a toy example, we might consider a function family F. Even assuming F is SIM-AC-PRF secure with P, it seems impossible to prove that F' is SIM-AC-PRF secure where F' is defined by $\mathsf{F}'.\mathsf{Ev}^{\mathsf{P}}(1^\lambda, k, (x_0, x_1)) = \mathsf{F.Ev}^{\mathsf{P}}(1^\lambda, \mathsf{F.Ev}^{\mathsf{P}}(1^\lambda, k, x_0), x_1)$.

One of Jaeger and Tyagi's proofs (for their Theorem D.1) almost ran into issue with this. However, they seemingly got "lucky" in that for that particular proof they were able to use just plain PRF security for the first use of the underlying function family.

Single-user security implies multi-user security. With most "standard" security notions (e.g. PRF, IND-CPA, IND-CCA) single-user security implies multi-user security. These results are

[10]Note this is the more general result, as we could let $\mathsf{P} = \mathsf{P}_1 \times \mathsf{P}_2 \times \ldots$ and have the i-th scheme using P only actually query P_i.

proven by a “hybrid proof” wherein the single-user attacker picks a user u at random. It externally simulates u with its own oracle, internally simulates all “prior” users as in the $b = 0$ world, and internally simulates all “later” users as in the $b = 1$ world.

We run into issue if we try to write an analogous proof for SIM-AC definitions. Note that simulating the $b = 0$ world for some users requires the attacker to run the given single-user simulator. This creates a circular dependency as in SIM-AC the simulator is allowed to depend on the adversary.

Even if we changed the order of quantification, we would still run into issues. Each instance of the single-user simulator expects to already have complete control of the ideal primitive. This makes it unclear what ideal primitive oracle the single-user adversary should provide the multi-user adversary it runs internally. Because of these issues, Jaeger and Tyagi directly consider multi-user SIM-AC definitions and do not discuss single-user variants thereof.

It may seem strange to consider “adaptive compromise” in a single-user setting. Do expose queries make sense where there is only one user to be exposed? It is useful to compare by first observing that multi-user SIM-AC notions would be unchanged if we required that the attacker expose all users before halting. Crucially, these definitions use “online” simulators that are forced to commit to simulated ciphertexts (without knowledge of the encrypted message) for users that will later be exposed (at which time the simulator is told the messages). They are thereby capturing “temporal” properties about security holding before exposures.

4 SIM*-AC Security

We saw in the previous section some ways in which SIM-AC security definitions cannot be used for proving results which intuitively “should” be possible to prove with a “good” security defintion. In this section, we will introduce a related class of security definitions which we notate by SIM*-AC. These new definition will strengthen the power of the attacker (by allowing them to program the ideal model) and weaken the power of the simulator (by requiring they be explicit about how they program the ideal model). This allows proving the results that were a challenge for the prior definitions, while still maintaining the value of the prior definitions. In particular, the results previously shown by Jaeger and Tyagi with SIM-AC can be shown to hold with SIM*-AC, while requiring minimal modifications to the proofs. We discuss the details of this in Section 6.

4.1 SIM*-AC Definitions for PRFs and Symmetric Encryption

Pseudorandom function security. We start with PRF security for a function family F. Our new definition is captured by the game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}$ shown in Fig. 3. It differs from $\mathrm{G}^{\mathsf{sim\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}$ as described above; namely, $\mathcal{A}_{\mathsf{prf}}$ is given oracle $\mathrm{PP{\small RIM}}$ which uses P in both the real and simulated world.[11] In the simulated world, S is also given $\mathrm{PP{\small RIM}}$ to query and program P. Note that the scheme algorithm $\mathsf{F.Ev}$ is still given access only to $\mathsf{P.Ls}$ and not to $\mathsf{P.Prog}$. The algorithms of F and S that can query the primitive are only run for the appropriate value of b, otherwise they would modify the state of P in undesired ways.

We define $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}(\lambda) = 2\Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}(\lambda)] - 1$ and say that F is SIM*-AC-PRF secure with P if there exists a PPT S such that for all PPT $\mathcal{A}_{\mathsf{prf}}$, the advantage function $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}(\cdot)$ is negligible. Note here that we quantified the simulator before the adversary, unlike in SIM-AC-PRF security where the simulator is allowed to depend on the adversary. This strengthens the

[11] Here we are using a notational convention that an algorithm given more inputs than it expects will ignore any extra inputs, so $\mathsf{P.Ls}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})$ is equivalent to $\mathsf{P.Ls}(1^\lambda, k, x : \sigma_{\mathsf{P}})$.

```
Game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}(\lambda)$
$k_{(\cdot)} \leftarrow\!\!\$\ \mathsf{F.Kg}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow\!\!\$\ \mathsf{P.Init}(1^\lambda)$
$\sigma \leftarrow\!\!\$\ \mathsf{S.Init}(1^\lambda)$
$b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\textsc{Ev},\textsc{Exp},\textsc{PPrim}}_{\mathsf{prf}}(1^\lambda)$
Return $(b = b')$

$\textsc{PPrim}(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$y \leftarrow\!\!\$\ \mathsf{P.Op}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})$
Return $y$

$\textsc{Ev}(u, x)$
If $T_u[x] \neq \bot$ then return $T_u[x]$
If $b = 1$ then $y \leftarrow\!\!\$\ \mathsf{F.Ev}^{\mathsf{P}}(1^\lambda, k_u, x)$
If $b = 0$ then
    If $X_u$ then $y \leftarrow \mathsf{S.Ev}^{\textsc{PPrim}}(1^\lambda, u, x : \sigma)$
    Else $y \leftarrow\!\!\$\ \mathsf{F.Out}(\lambda)$
$T_u[x] \leftarrow y$
Return $y$

$\textsc{Exp}(u)$
If $b = 1$ then $k' \leftarrow k_u$
If $b = 0$ then $k' \leftarrow\!\!\$\ \mathsf{S.Exp}^{\textsc{PPrim}}(1^\lambda, u, T_u : \sigma)$
$X_u \leftarrow \mathsf{true}$; Return $k'$
```

```
Game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda)$
$k_{(\cdot)} \leftarrow\!\!\$\ \mathsf{SE.Kg}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow\!\!\$\ \mathsf{P.Init}(1^\lambda)$
$\sigma \leftarrow\!\!\$\ \mathsf{S.Init}(1^\lambda)$
$b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\textsc{Enc},\textsc{Dec},\textsc{Exp},\textsc{PPrim}}_{\mathsf{cca}}(1^\lambda)$
Return $(b = b')$

$\textsc{PPrim}(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$y \leftarrow\!\!\$\ \mathsf{P.Op}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})$
Return $y$

$\textsc{Enc}(u, m)$
If $\neg X_u$ then $\ell \leftarrow |m|$ else $\ell \leftarrow m$
If $b = 1$ then $c \leftarrow\!\!\$\ \mathsf{SE.Enc}^{\mathsf{P}}(1^\lambda, k_u, m)$
If $b = 0$ then $c \leftarrow\!\!\$\ \mathsf{S.Enc}^{\textsc{PPrim}}(1^\lambda, u, \ell : \sigma)$
$M_u.\mathsf{add}(c, m)$; Return $c$

$\textsc{Dec}(u, c)$
If $M_u\langle c\rangle \neq \bot$ then return $M_u\langle c\rangle$
If $b = 1$ then $m \leftarrow \mathsf{SE.Dec}^{\mathsf{P}}(1^\lambda, k_u, c)$
If $b = 0$ then $m \leftarrow\!\!\$\ \mathsf{S.Dec}^{\textsc{PPrim}}(1^\lambda, u, c : \sigma)$
Return $m$

$\textsc{Exp}(u)$
If $b = 1$ then $k' \leftarrow k_u$
If $b = 0$ then $k' \leftarrow\!\!\$\ \mathsf{S.Exp}^{\textsc{PPrim}}(1^\lambda, u, M_u : \sigma)$
$X_u \leftarrow \mathsf{true}$; Return $k'$
```

Figure 3: Games defining SIM*-AC-PRF security of F and SIM*-AC-CCA security of SE. We use highlighting to indicate where the definitions differ from SIM-AC versions.

definition and is necessary for some of our positive results, but for some of our results the weaker quantification will suffice. We say F is wSIM*-AC-PRF secure with P if for all PPT $\mathcal{A}_{\mathsf{prf}}$ there exists a PPT S such that $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}(\cdot)$ is negligible.

Encryption definitions. The SIM*-AC-CCA security of an encryption scheme SE is similarly captured by the game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cca}}$ defined in Fig. 3 which modifies the SIM-AC game to have the attacker and simulator both use $\textsc{PPrim}$. We define $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda) = 2\Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda)] - 1$ and say SE is SIM*-AC-CCA secure with P if there exists a PPT S such that for all PPT $\mathcal{A}_{\mathsf{cca}}$, the advantage function $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\cdot)$ is negligible. wSIM*-AC-CCA is captured by quantifying the simulator after the adversary.

Chosen-plaintext security is captured by restricting attention to attackers that do not query decryption. We then write $\mathsf{sim^*\text{-}ac\text{-}cpa}$ in superscripts. SIM*-AC-X and wSIM*-AC-X security for $\mathrm{X} \in \{\mathrm{KPA}, \$, \mathrm{AE}\}$ security are defined by restricting the behavior of the simulator appropriately.

4.2 Discussing the Definitions

In this section, we discuss the ideas that motivated how we defined SIM*-AC security. In the next sections we show how the definitions can prove results that seem out of reach for SIM-AC security.

Motivating the new definition. The starting place for our new definitions partially goes back to the original explicit proposal of random oracles by Bellare and Rogaway [BR93]. Therein, their definition of zero knowledge in the random oracle model requires that the (offline) simulator's final outputs includes the list of points at which it would like the random oracle to have given values. At all other points, the oracle is sampled at random. Wee [Wee09] built on this, considering different levels of how the simulator controls the random oracle and showing that zero-knowledge proofs are closed under sequential composition when the random oracle is explicitly programmable (or non-programmable). Sequential composition fails in the "fully programmable" model as applying the simulator for the first round of execution replaces the random oracle completely, at which point we cannot use it to reason about further rounds.

There is a second subtle detail allowing Wee's sequential composition proof to go though with polynomially many rounds. It is important that (part of) the adversary was quantified *after* the simulator. The proof followed a hybrid argument wherein rounds of zero knowledge are switched from real to simulated, one at a time. To apply security for a particular round, the attacker must simulate the other (real and simulated) rounds. For a constant number of rounds, we could fix the attacker for the first round, be given its simulator, use the simulator in the attacker for the second round, be given its simulator, and so on. When the number of rounds is polynomial, we cannot fix an attacker for each round. Instead a single attacker must work for all rounds, which requires knowing the simulator ahead of time so it can properly emulate simulated rounds.

To resolve the issues identified with composition and hybrid arguments for SIM-AC we restrict the simulator to explicitly program the ideal primitive and require a universal simulator that works for all attackers. However, this still is not enough! The zero knowledge composition discussed above is importantly "sequential" in an "offline simulation" setting. The simulator runs once in isolation, then provides its output to the attacker which runs in isolation. The attacker has complete control over all code executing with it, so can perfectly emulate the programmed random oracle. In an "online simulation" setting like SIM-AC, the attacker runs in parallel with the honest scheme algorithms or the simulator. Our proofs would run into issues when attackers internally run copies of the simulator which want to program the random oracle, but the attacker is then unable to force the honest scheme algorithms or simulator it does not control to use this modified random oracle. We resolve this issue by expanding the power of the adversary and giving it the capacity to program the ideal primitive.[12] We use the prefix SIM*-AC for all definitions we write in this style.

Summarizing, in our SIM*-AC definitions simulators and adversaries can access an oracle $\mathrm{PP}\textsc{rim}$ which allows them to evaluate *or explicitly program* the ideal primitive. Schemes are still restricted to not program the ideal primitive. This is a restriction on the simulator and strengthening of the attacker. Because of the programmability of P we must write the code so that S is only run in the ideal world and SE is only run in the real world.

Comparisons to prior definitions. Through this sequence of ideas we have reached the same general structure of random oracle modeling proposed by Camenisch, Drijvers, Gagliardoni, Lehmann, and Neven [CDG+18]. Their work is in the universal composability (UC) setting where they consider several models for global random oracles. In one, simulators and adversaries can explicitly program the random oracle. They show it allows security proofs that very efficient and natural random oracle-based constructions of several primitives satisfy the desired security. Our work generalizes this any ideal primitive (not just random oracles) and considers its application outside the universal composability framework. That UC and SIM-AC work well with a similar programability notion is, in hindsight, natural as they both consider *online* simulation.

[12]Wee would have run into similar issues had their hybrid tried to switch rounds to simulated from first to last, rather than the last to first approach they took.

Our SIM*-AC definitions are not strictly better for cryptographers than the SIM-AC definitions of Jaeger and Tyagi [JT20]. One benefit of their work was the ease with which existing results could be ported to the SIM-AC setting (e.g. replacing IND-CPA in a proof with SIM-AC-CPA). This holds to some extent with the new SIM*-AC definitions as well, but proofs do occasionally run into additional difficulties because of fragilities caused by the programming of the oracle. Overall we believe that this cost is worth the benefits provided by our new definitions being able to show natural and desirable results that are seemingly out of reach of plain SIM-AC.

High-level remarks. There is value in incorporating this explicit programming capacity for adversaries even into non-simulation definitions. Consider the construction of some high-level system making use of multiple underlying schemes that use the same ideal primitive, some for SIM*-AC security and some for non-simulation security notions. (See, e.g., the searchable encryption proof in [JT20] that involved the standard notion of PRF security in addition to SIM-AC-PRF/KPA security.) If the proof requires use of the non-simulation security notion *after* a SIM*-AC notion has already been applied, this will only be possible if the attacker can program the ideal primitive in the non-simulation notion.

Allowing the adversary to program the ideal primitive is *strange*. It does not seem to capture anything about reality, despite the fact that we allow the adversary to do this programming even in the "real world". However, this ability will be crucial to how we can use this new definition to prove the results that we were unable to with the original SIM-AC definitions. We can view this in the same paradigm we discussed for SIM-AC-style definitions in general; there is value in studying very strong definitions which exploit ideal primitives beyond how they can reasonably be thought to capture something about reality because these notions can then serve as intermediate steps for proving (in the ideal model) that the scheme satisfies other more "reasonable" security notions.

4.3 Single-user Security Implies Multi-user Security

As with SIM-AC security, we can capture single-user SIM*-AC security by requiring that all of the attacker's oracle queries use the same value of u. The following theorem captures that single-user SIM*-AC-CPA security implies multi-user security. The result would also hold with SIM*-AC-X security for any $\mathrm{X} \in \{\mathrm{PRF}, \mathrm{CCA}, \$, \mathrm{AE}\}$, via the same proof technique. If *does not* hold for $\mathrm{X} = \mathrm{KP}$. We will discuss why in more detail after the proof.

Theorem 4.1 *Single-user SIM*-AC-CPA security implies multi-user SIM*-AC-CPA security.*

This proof follows using the ideas from a fairly standard single-user to multi-user proof via a hybrid argument. Given a single-user simulator S_1 and multi-user adversary $\mathcal{A}$, we define single-user $\mathcal{A}_1$ to pick a random t and respond to queries with $u < t$ by encrypting honestly, with $u = t$ using its own encryption oracle, and with $u > t$ using a copy of S_1 specific for that user. The multi-user simulator we construct runs multiple independent copies of the single-user simulator – one for each user. Note that this proof critically requires all three of the changes we used to derive SIM*-AC from SIM-AC: (i) the simulator needs to be quantified before the adversary so that $\mathcal{A}_1$ can run S_1, (ii) the simulator must not have full control of the ideal primitives output so there is no ambiguity in which "copy" of the simulator run by $\mathcal{A}_1$ should get to respond to primitive queries, and (iii) the adversary must be able to program the ideal primitive so that $\mathcal{A}_1$ is able to correctly control the primitive when running copies of S_1.

Proof: Let SE be single-user SIM*-AC-CPA secure with P and S_1 be the simulator that is guaranteed to exist. We show that SE is SIM*-AC-CPA secure with P via the following simulator which runs independent copies of S_1 for each user.

```
Hybrid $\mathrm{H}_i(\lambda)$, $0 \le i \le u_\lambda$
For $u \in [u_\lambda]$ do
    $k_u \leftarrow\$ \mathsf{SE.Kg}(1^\lambda)$
    $\sigma_u \leftarrow\$ \mathsf{S}_1.\mathsf{Init}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow\$ \mathsf{P.Init}(1^\lambda)$
$b' \leftarrow\$ \mathcal{A}^{\textsc{Enc},\textsc{Exp},\textsc{PPrim}}(1^\lambda)$
Return $(b' = 1)$

$\textsc{Enc}(u, m)$
If $u \le i$ then $d \leftarrow 0$
Else $d \leftarrow 1$
$c \leftarrow \textsc{Enc}_d(u, m)$
Return $c$

$\textsc{Exp}(u)$
If $u \le i$ then $d \leftarrow 0$
Else $d \leftarrow 1$
$k \leftarrow \textsc{Exp}_d(u)$
Return $k$
```

```
$\textsc{Enc}_d(u, m)$
If $\neg X_u$ then $\ell \leftarrow |m|$ else $\ell \leftarrow m$
If $d = 1$ then $c \leftarrow\$ \mathsf{SE.Enc}^{\mathsf{P}}(1^\lambda, k_u, m)$
Else $c \leftarrow\$ \mathsf{S}_1.\mathsf{Enc}^{\textsc{PPrim}}(1^\lambda, u, \ell : \sigma_u)$
$M_u.\mathsf{add}(c, m)$
Return $c$

$\textsc{Exp}_d(u)$
If $d = 1$ then $k \leftarrow k_u$
Else $k \leftarrow\$ \mathsf{S}_1.\mathsf{Exp}^{\textsc{PPrim}}(1^\lambda, u, M_u : \sigma_u)$
$X_u \leftarrow \mathsf{true}$
Return $k$
```

```
Adversary $\mathcal{A}_1^{\textsc{Enc},\textsc{Exp},\textsc{PPrim}}(\lambda)$
For $u \in [u_\lambda]$ do
    $k_u \leftarrow\$ \mathsf{SE.Kg}(1^\lambda)$
    $\sigma_u \leftarrow\$ \mathsf{S}_1.\mathsf{Init}(1^\lambda)$
$t \leftarrow\$ \{1, \ldots, u_\lambda\}$
$b' \leftarrow\$ \mathcal{A}^{\textsc{EncSim},\textsc{ExpSim},\textsc{PPrim}}(1^\lambda)$
Return $b'$

$\textsc{Enc}_d(u, m)$, $\textsc{Exp}_d(u)$
//Unchanged from above

$\textsc{EncSim}(u, m)$
If $u < t$ then $c \leftarrow \textsc{Enc}_0(u, m)$
Else if $u = t$ then $c \leftarrow \textsc{Enc}(u, m)$
Else $c \leftarrow \textsc{Enc}_1(u, m)$
Return $c$

$\textsc{ExpSim}(u)$
If $u < t$ then $k \leftarrow \textsc{Exp}_0(u)$
Else if $u = t$ then $k \leftarrow \textsc{Exp}(u)$
Else $k \leftarrow \textsc{Exp}_1(u)$
Return $k$
```

Figure 4: Hybrids and adversary showing single-user security implies multi-user

```
$\mathsf{S.Init}(1^\lambda)$
$\sigma_{(\cdot)} \leftarrow\$ \mathsf{S}_1.\mathsf{Init}(1^\lambda)$
Return $\sigma_{(\cdot)}$

$\mathsf{S.Enc}^{\textsc{PPrim}}(1^\lambda, u, \ell : \sigma_{(\cdot)})$
$c \leftarrow\$ \mathsf{S}_1.\mathsf{Enc}^{\textsc{PPrim}}(1^\lambda, u, \ell : \sigma_u)$
Return $c$

$\mathsf{S.Exp}^{\textsc{PPrim}}(1^\lambda, u, M_u : \sigma_{(\cdot)})$
$k \leftarrow\$ \mathsf{S}_1.\mathsf{Exp}^{\textsc{PPrim}}(1^\lambda, u, M_u : \sigma_u)$
Return $k$
```

Let $\mathcal{A}$ be a SIM*-AC-CPA adversary. It will be notationally convenient to assume that it only queries users with identifiers $u \in [u_\lambda] = \{1, \ldots, u_\lambda\}$ where $u_{(\cdot)}$ is a polynomial. This assumption is without loss of generality.

Now, consider the hybrid games H_i for $i = 0, \ldots, u_\lambda$ defined in Fig. 4. For $u \le i$, the game uses $\textsc{Enc}_0$ and $\textsc{Exp}_0$ to respond to encryption and exposure queries as in the $b = 0$ simulated world of $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cpa}}$ using S. Otherwise, it uses $\textsc{Enc}_1$ and $\textsc{Exp}_1$ to respond as in the $b = 1$ real world. Each hybrid game returns true whenever $\mathcal{A}$ outputs 1. When $i = u_\lambda$, it always holds that $u \le i$ so this game is identical to the $b = 0$ simulated world (except that the output boolean is flipped). In the other extreme, when $i = 0$, it never holds that $u \le i$ so this game is identical to the $b = 1$ real world. Then (by standard conditional probability calculation) we have

$$\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cpa}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}}(\lambda) = \Pr[\mathrm{H}_0] - \Pr[\mathrm{H}_{u_\lambda}] = \sum_{i=1}^{u_\lambda} \Pr[\mathrm{H}_{i-1}] - \Pr[\mathrm{H}_i].$$

We construct a single-user adversary $\mathcal{A}_1$ that obtains advantage $1/u_\lambda$ times the above. It samples an index $t \in \{1, \ldots, u_\lambda\}$ at random. Then it runs $\mathcal{A}$, simulating their oracle queries. When $u < t$, it responds as in the simulated world of $\mathrm{G}^{\mathsf{ep\text{-}sim\text{-}ac\text{-}cpa}}$ using S_1. When $u = t$ it forwards the query to its own oracle. Otherwise, it responds to $\textsc{Enc}$ and $\textsc{Exp}$ queries as in the real world. Let b denote the bit in the game $\mathcal{A}_1$ is being run in and t be the random value picked by $\mathcal{A}_1$. Then in the view of $\mathcal{A}$, the oracles for the first $t-b$ users are simulated and the rest are real – this is identical to its view in the hybrid game H_{t-b}.

Then the following calculations complete the proof.

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cpa}}_{\mathsf{SE},\mathsf{S}_1,\mathsf{P},\mathcal{A}_1}(\lambda) &= \mathbf{E}_t[\Pr[\mathrm{H}_{t-1}]] - \mathbf{E}_t[\Pr[\mathrm{H}_{t-0}]] \\
&= (1/u_\lambda)\sum_{t=1}^{u_\lambda}\Pr[\mathrm{H}_{t-1}] - (1/u_\lambda)\sum_{t=1}^{u_\lambda}\Pr[\mathrm{H}_t] \\
&= (1/u_\lambda)\sum_{i=1}^{u_\lambda}\Pr[\mathrm{H}_{i-1}] - \Pr[\mathrm{H}_i] \\
&= (1/u_\lambda)\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cpa}}_{\mathsf{SE},\mathsf{S},\mathsf{P},\mathcal{A}}(\lambda).
\end{aligned}$$

Here $\mathbf{E}_t$ denotes expectation over $t \leftarrow_\$ \{1,\ldots,u_\lambda\}$. ▮

We can note in the above proof that for $\mathcal{A}_1$ to be able to correctly run $\textsc{Enc}_0$ and $\textsc{Exp}_0$ it needed to run S_1. This means that we needed the stronger quantification where the adversary can depend on the simulator and that the adversary needed to have the ability to program the random oracle.

Key-private security. Among the various SIM*-AC security notions we consider here, the only variant for which single-user security does not imply multi-user security is SIM*-AC-KPA security. Here, the simulator may not make use of its input u when replying to encryption queries for unexposed users (beyond checking if they are exposed). Note that in the hybrid argument above, the multi-user simulator S uses the user identifier u to decide which state σ_u to use. Hence this is incompatible with SIM*-AC-KPA security. Taking a step back, we can notice that this issue with the proof is unsurprising and inherent. The issue is that that single-user SIM*-AC-KPA does not meaningfully capture any notion of key-privacy because the restriction on the simulator's behavior is trivially achievable when the attacker will only every query a single user. This is nicely captured by the following result.

Theorem 4.2 *Single-user SIM*-AC-KPA security is equivalent to SIM*-AC-CPA security, which is weaker than SIM*-AC-KPA security.*

Proof: Note that single-user SIM*-AC-KPA security implies single-user SIM*-AC-CPA security trivially. Then, by Theorem 4.1 this implies SIM*-AC-CPA security. In the other direction, we can create a single-user SIM*-AC-KPA simulator from a SIM*-AC-CPA simulator by always running the latter on, say, $u = 1$. Hence the first claim of the theorem holds.

We can see that SIM*-AC-CPA security is weaker than SIM*-AC-KPA security by constructing a contrived scheme. Given some scheme SE, we define a new scheme which adds a random bit d to its keys and then appends d to every ciphertext produced. It is straightforward to show this new scheme is SIM*-AC-CPA secure if SE was, but that is is not SIM*-AC-KPA secure. ▮

4.4 Cascade PRF Construction

If $\mathsf{F} : \mathsf{F.K} \times \mathsf{F.Inp} \to \mathsf{F.K}$ is a function family and n is a polynomial, then the n-cascade construction $\mathsf{F}^n : \mathsf{F.K} \times \mathsf{F.Inp}^n \to \mathsf{F.K}$ is defined by the evaluation algorithm $\mathsf{F^n.Ev}(1^\lambda, k_0, \vec{x})$ which computes

$k_i \leftarrow \mathsf{F.Ev}(1^\lambda, k_{i-1}, \vec{x}_i)$ for $i = 1, \ldots, n(\lambda)$ and then outputs $k_{n(\lambda)}$. Here $\vec{x}_i$ denotes the i-th entry of vector $\vec{x}$. This is a "domain extension" technique for building a PRF with a large domain from one with a small domain. It was originally defined and analyzed in [BCK96b].[13] F^n generalizes the GGM construction of a PRF from a PRG [GGM86]. It underlies several other constructions of PRFs including AMAC, HMAC, and NMAC [BBT16, BCK96a, Bel06].

Theorem 4.3 *If* F *is SIM*-AC-PRF secure with* P*, then* F^n *is as well.*

The proof of this result is given in Appendix A. Intuitively, we can think of the possible keys generated by F^n existing in a tree structure. Our proof does a hybrid argument over the layers of the tree, starting from the root, where we one at a time switch the layers to being simulated. The simulator for a given layer treats all of the keys at its layer as being multiple F "users". This proof requires the "strong" quantification, the simulator to not completely replacing the ideal primitive, and the adversary having the ability to program the ideal primitive so that it can internally run the simulator for layers that have been switched already.

Jaeger and Tyagi [JT20, ePrint, p.22-23] said, "It is often useful to construct a PRF H with large input domains from a PRF F with smaller input domains [. . .] one can often [use our techniques] to lift a PRF security proof for H to a SIM-AC-PRF security proof for H whenever F is SIM-AC-PRF secure." The cascade construction is one choice of H for which this is *not* possible with SIM-AC, but becomes possible with SIM*-AC.

5 Asymmetric Encryption

In this section, we provide our treatment for the security of asymmetric cryptographic primitives against adaptive compromise. We start by providing our security definitions for public-key encryption (PKEs) and key-encapsulation mechanisms (KEMs). Then we discuss how our definitions compare to prior definitions, in particular those of Camensich, Lehmann, Neven, and Samelin [CLNS17]. We show that the KEM/DEM approach to constructing a PKE scheme works with these definitions and that standard ways of constructing CPA/CCA secure KEMs from one-way secure primitives and a random oracle are secure.

5.1 Definitions

Public-key encryption. The SIM*-AC-CCA security of a public-key encryption scheme PKE is captured by the game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cca}}$ shown in Fig. 5. It differs from the SIM*-AC-CCA definition for symmetric encryption (Fig. 2) in that it introduces an encryption key oracle (Ek) that the adversary can call to learn the public encryption key for a user and it has oracles for two different kinds of exposure. The receiver exposure oracle (RExp) is like the exposure oracles from prior games, returning a user's secret decryption key. The sender exposure oracle (SExp) allows the attacker to ask for the randomness underlying the ciphertexts that were returned by encryption. We use $\mathsf{PKE.Rand}$ to denote the set from which this randomness is sampled.

We define $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{PKE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda) = 2\Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{PKE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda)] - 1$ and say PKE is SIM*-AC-CCA secure with P if there exists a PPT S such that for all PPT $\mathcal{A}_{\mathsf{cca}}$, the advantage function $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{PKE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\cdot)$ is negligible. wSIM*-AC-CCA is captured by quantifying the simulator after the adversary. We capture xSIM*-AC-CPA by ignoring the decryption oracle. Security considering only compromise

[13]Technically, they considered a more general construction where the number of iterations was not a priori fixed and so the adversary was restricted to make only prefix-free queries. Our proof would extend to this setting as well.

```
Game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{PKE},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda)$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow_{\$} \mathsf{PKE.Kg}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow_{\$} \mathsf{P.Init}(1^\lambda)$
$\sigma \leftarrow_{\$} \mathsf{S.Init}(1^\lambda)$
$b \leftarrow_{\$} \{0,1\}$
$b' \leftarrow_{\$} \mathcal{A}_{\mathsf{cca}}^{\textsc{Ek},\textsc{Enc},\textsc{Dec},\textsc{SExp},\textsc{RExp},\textsc{PPrim}}(1^\lambda)$
Return $(b = b')$

Game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{KEM},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cca}}}(\lambda)$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow_{\$} \mathsf{KEM.Kg}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow_{\$} \mathsf{P.Init}(1^\lambda)$
$\sigma \leftarrow_{\$} \mathsf{S.Init}(1^\lambda)$
$b \leftarrow_{\$} \{0,1\}$
$b' \leftarrow_{\$} \mathcal{A}_{\mathsf{cca}}^{\textsc{Ek},\textsc{Encaps},\textsc{Decaps},\textsc{SExp},\textsc{RExp},\textsc{PPrim}}(1^\lambda)$
Return $(b = b')$

$\textsc{Ek}(u)$
If $b = 1$ then $ek' \leftarrow ek_u$
If $b = 0$ then $ek' \leftarrow_{\$} \mathsf{S.Ek}^{\textsc{PPrim}}(1^\lambda, u : \sigma)$
Return $ek'$

$\textsc{PPrim}(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$y \leftarrow_{\$} \mathsf{P.Op}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})$
Return $y$

$\textsc{Enc}(u, m)$
If $\neg X_u$ then $\ell \leftarrow |m|$ else $\ell \leftarrow m$
$r \leftarrow_{\$} \mathsf{PKE.Rand}(\lambda)$
If $b = 1$ then $c \leftarrow \mathsf{PKE.Enc}^{\mathsf{P}}(1^\lambda, ek_u, m; r)$
If $b = 0$ then $c \leftarrow_{\$} \mathsf{S.Enc}^{\textsc{PPrim}}(1^\lambda, u, \ell : \sigma)$
$M_u.\mathsf{add}(c, m)$; $R_u.\mathsf{add}(r)$
Return $c$

$\textsc{Dec}(u, c)$
If $M_u\langle c\rangle \neq \bot$ then return $M_u\langle c\rangle$
If $b = 1$ then $m \leftarrow \mathsf{PKE.Dec}^{\mathsf{P}}(1^\lambda, dk_u, c)$
If $b = 0$ then $m \leftarrow_{\$} \mathsf{S.Dec}^{\textsc{PPrim}}(1^\lambda, u, c : \sigma)$
Return $m$

$\textsc{SExp}(u, i)$
If $b = 1$ then $r \leftarrow R_u[i]$
If $b = 0$ then $r \leftarrow_{\$} \mathsf{S.SExp}^{\textsc{PPrim}}(1^\lambda, u, i, M_u[i] : \sigma)$
Return $r$

$\textsc{RExp}(u)$
If $b = 1$ then $dk' \leftarrow dk_u$
If $b = 0$ then $dk' \leftarrow_{\$} \mathsf{S.RExp}^{\textsc{PPrim}}(1^\lambda, u, M_u : \sigma)$
$X_u \leftarrow \mathsf{true}$
Return $dk'$

$\textsc{Encaps}(u)$
$r \leftarrow_{\$} \mathsf{KEM.Rand}(\lambda)$
If $b = 1$ then $(c, k) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek_u; r)$
If $b = 0$ then
    $(c, k) \leftarrow_{\$} \mathsf{S.Encaps}^{\textsc{PPrim}}(1^\lambda, u : \sigma)$
    If $\neg X_u$ then $k \leftarrow_{\$} \mathsf{KEM.K}(\lambda)$
$M_u.\mathsf{add}(c, k)$; $R_u.\mathsf{add}(r)$
Return $(c, k)$

$\textsc{Decaps}(u, c)$
If $M_u\langle c\rangle \neq \bot$ then return $M_u\langle c\rangle$
If $b = 1$ then $k \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c)$
If $b = 0$ then $k \leftarrow_{\$} \mathsf{S.Decaps}^{\textsc{PPrim}}(1^\lambda, u, c : \sigma)$
Return $k$
```

Figure 5: Games defining the SIM*-AC-CCA security of PKE and KEM

of the receiver/sender can be captured by ignoring the appropriate oracle. Then we write SIM*-rAC or SIM*-sAC.

Key encapsulation mechanism. We also give definitions for key encapsulation mechanisms (KEM). Our SIM*-AC definitions are highly analogous to the corresponding public-key encryption definition. They are formally specified by the game $\mathrm{G}^{\mathsf{sim\text{-}ac\text{-}cca}}$ shown in Fig. 5. Therein, the Enc and Dec oracles have been replaced with Encaps and Decaps oracles. The encapsulation oracle returns a ciphertext along with the corresponding encapsulated key. In the ideal world, the simulator provides the ciphertext and the encapsulated key is chosen at random from the key space $\mathsf{KEM.K}$ by the game for unexposed users.

We define $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{KEM},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{cpa}}}(\lambda)$ and the notions xSIM*-yAC-X for $\mathrm{x} \in \{\varepsilon, \mathrm{w}\}$, $\mathrm{y} \in \{\varepsilon, \mathrm{r}, \mathrm{s}\}$, and $\mathrm{X} \in \{\mathrm{CCA}, \mathrm{CPA}\}$ as for PKE.

5.2 Comparisons to Prior Definitions

Brunetta, Heum, and Stam [BHS24] systematized a large number of definitions of security for the adaptive compromise of asymmetric encryption. The strongest definition they consider (NCE-CCA) is roughly equivalent to a SIM-AC variant of our definition and is thus implied by SIM*-AC-CCA.

FULL-SIM Security of PKE. Our SIM*-AC-CCA definition for PKE is similar to the FULL-SIM security definition introduced by Camensich, Lehmann, Neven, and Samelin (CLNS) [CLNS17]. We quickly summarize the differences. There are two dimensions in which their definition is stronger than ours. First, their definition considers PKE with labels, while we have decided not consider labels. Labels can easily be added. Likely, the best way to incorporate labels in constructions would be to use a symmetric encryption scheme that accepts associated data as part of the KEM/DEM transform (discussed momentarily). Second, in FULL-SIM the randomness used by key generation is revealed rather than the decryption key. SIM-AC* can be used to reason over this case by simply modify the scheme to use said randomness as its decryption key (and recompute the actual decryption key during decryption).

Our definition strengthens theirs in several dimension. Theirs is more closely analogous to SIM-AC than SIM*-AC as the simulator is given complete control of the random oracle, the adversary is not able to modify it, and the "weak" quantification is used. Resultantly, their single-user definition is seemingly unable to prove that a corresponding multi-user definition holds.[14] We are not restrictive in the type of ideal primitive considered where they specifically assume a random oracle is used.

CLNS provide a specific FULL-SIM construction in which they basically used a trapdoor permutation generator as a KEM and then hand-crafted a symmetric encryption scheme using a random oracle.[15] We will momentarily show that the task of building SIM*-AC secure PKE can be broken down into constructing KEMs and symmetric encryption. This is modular, allowing numerous instantiation and allowing the symmetric encryption to be instantiated by well-studied and standardized schemes based on blockciphers rather than using less efficient hash functions throughout.

CLNS showed that FULL-SIM implied a variety of prior definitions considering compromise scenarios for PKE. Appropriate analogs of these implications will carry over to our definition as well. Further, CLNS considered a UC secure notion and proved it to be essentially equivalent to FULL-SIM. Camenisch, Drijvers, Gagliardoni, Lehmann, and Nevin [CDG+18] considered this in the UC programmable random oracle model, proving the same construction secure. Likely SIM*-AC-CCA is equivalent to this notion and so our result will give modular, standard, efficient instantiations of UC secure public key encryption secure under adaptive compromise.

NCKE Security of KEMs. Our KEM SIM*-AC definitions are similar to the NCKE definitions of Jager, Kiltz, Riepel, and Schäge [JKRS21]. Their definition is roughly equivalent to a SIM-AC variant which only considers receiver corruptions and assumes there is a separate random oracle for each user. The definition can be viewed as placing some additional restrictions on the simulator, including that it must commit ahead of time to the secret keys for each user and cannot store any state beyond specific information stored for it by the game in unordered sets.[16] Simulators for the

[14] CLNS claim (their Proposition 2) that FULL-SIM security implies a weaker multi-user definition called RSIM-SO, but the proof is incomplete and seems to be erroneous. SIM*-AC-CPA security should suffice for this implication.

[15] Speaking loosely, they basically use a random input to the trapdoor permutation as a "symmetric key" with which they perform counter mode encryption, using the random oracle as a pseudorandom function and then perform a MAC over all of the relevant variables, again using the random oracle as a pseudorandom function.

[16] Technically, as written, the KEM encapsulation algorithm is run even in the "simulated" world of their definition. This can modify the state of the random oracle but is clearly unintended, as later use of the definition does not account for this. In general, care is needed with stateful ideal primitives as subtleties abound.

schemes we consider can follow these restrictions.

Having separate random oracles for every user allowed using two separate reductions to NCKE in their high-level proofs (for key exchange protocols built from KEMs). It seems likely NCKE could be replaced by SIM*-AC in these proofs, removing the need for separate random oracles.

5.3 KEM/DEM Hybrid Encryption.

A common technique for building public key encryption is KEM/DEM hybrid encryption in which a key encapsulation mechanism produces a key which is then used to encrypt the message with a symmetric encryption scheme (i.e. "data encapsulation mechanism"). This was originally proven secure by Cramer and Shoup [CS03].

Let KEM be a key encapsulation mechanism and SE be a symmetric encryption scheme (i.e. data encapsulation mechanism) where $\mathsf{SE.Kg}$ samples uniformly from $\mathsf{KEM.K}$. We denote the KEM/DEM scheme as $\mathsf{KD}[\mathsf{KEM}, \mathsf{SE}]$ and provide the algorithms $\mathsf{KD.Enc}$ and $\mathsf{KD.Dec}$ below, where we assume KEM and SE expect access to ideal primitive P. Then KD expects access to P. It key generation algorithm is defined by $\mathsf{KD}[\mathsf{KEM}, \mathsf{SE}].\mathsf{Kg} = \mathsf{KEM.Kg}$.

```
$\underline{\mathsf{KD}[\mathsf{KEM},\mathsf{SE}].\mathsf{Enc}^{\mathsf{P}}(1^\lambda, ek, m)}$
$(c_{\mathsf{KEM}}, k) \leftarrow\!\!\$\ \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek)$
$c_{\mathsf{SE}} \leftarrow\!\!\$\ \mathsf{SE.Enc}^{\mathsf{P}}(1^\lambda, k, m)$
$c \leftarrow (c_{\mathsf{KEM}}, c_{\mathsf{SE}})$
Return $c$

$\underline{\mathsf{KD}[\mathsf{KEM},\mathsf{SE}].\mathsf{Dec}^{\mathsf{P}}(1^\lambda, dk, c)}$
$(c_{\mathsf{KEM}}, c_{\mathsf{SE}}) \leftarrow c$
$k \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk, c_{\mathsf{KEM}})$
$m \leftarrow \mathsf{SE.Dec}^{\mathsf{P}}(1^\lambda, k, c_{\mathsf{SE}})$
Return $m$
```

It is assumed that $\mathsf{SE.Dec}$ immediately halts and returns $\perp$ if $k = \perp$. Next, we show that given the appropriate adaptive compromise security for the underlying KEM scheme and encryption scheme, the composed KEM/DEM scheme is also secure against adaptive compromise.

Exposure of encryption randomness is not captured by our definitions for symmetric encryption. Rather than introduce a new security definition, in these cases we restrict attention to *coin extractable* schemes for which there exists a coin extraction algorithm $\mathsf{SE.CExt}$ which always satisfies $\mathsf{SE.CExt}^{\mathsf{P}}(1^\lambda, k, \mathsf{SE.Enc}^{\mathsf{P}}(1^\lambda, k, m; r)) = r$. Typical symmetric encryption schemes satisfy this. For technical reasons, we require that $\mathsf{SE.CExt}$ is query consistent by which we mean that it does not make any ideal primitive queries that were not made by the execution of $\mathsf{SE.Enc}$ that produced its input.

Theorem 5.1 *Let $x \in \{\varepsilon, w\}$, $y \in \{\varepsilon, r, s\}$, and $X \in \{CPA, CCA\}$. If* KEM *is xSIM*-yAC-X secure with* P *and* SE *is xSIM*-AC-X secure with* P *(and coin extractable if $y \in \{\varepsilon, s\}$), then* $\mathsf{KD}[\mathsf{KEM}, \mathsf{SE}]$ *is xSIM*-yAC-X secure with* P.

In fact, for the DEM we need only "single-challenge" security wherein the attacker makes at most one encryption query per user. This allows the use of deterministic DEMs. The proof of this theorem is given in Appendix B. The general flow of the proof is what one would expect, first we replace honest use of the KEM with simulated use that outputs uniformly random keys. We think of the i-th key generated for user u as correspond to a DEM user (u, i) and replace the DEM with simulation.

5.4 Hashed KEM

We consider a simple, standard way to construct a CPA secure KEM from a one-way secure KEM and a random oracle. Conceptually, this construction follows from the CPA secure PKE scheme considered in [BR93]. Let KEM be a key encapsulation mechanism. Then the hashed KEM scheme which outputs the hash of a key generated by KEM is denoted as $\mathsf{HKEM}[\mathsf{KEM}]$. Its algorithms are defined as follows. Its key generation algorithm is defined by $\mathsf{HKEM}[\mathsf{KEM}].\mathsf{Kg} = \mathsf{KEM}.\mathsf{Kg}$.

```
$\underline{\mathsf{HKEM}[\mathsf{KEM}].\mathsf{Encaps}^{\mathsf{P}\times\mathsf{P}_{\mathsf{rom}}}(1^\lambda, ek)}$
$(c, k_{\mathsf{KEM}}) \leftarrow\!\!\$\ \mathsf{KEM}.\mathsf{Encaps}^{\mathsf{P}}(1^\lambda, ek)$
$k \leftarrow \mathsf{P}_{\mathsf{rom}}(k_{\mathsf{KEM}}, \varepsilon)$; Return $(c, k)$
```

```
$\underline{\mathsf{HKEM}[\mathsf{KEM}].\mathsf{Decaps}^{\mathsf{P}\times\mathsf{P}_{\mathsf{rom}}}(1^\lambda, dk, c)}$
$k_{\mathsf{KEM}} \leftarrow \mathsf{KEM}.\mathsf{Decaps}^{\mathsf{P}}(1^\lambda, dk, c)$
$k \leftarrow \mathsf{P}_{\mathsf{rom}}(k_{\mathsf{KEM}}, \varepsilon)$; Return $k$
```

If the KEM expects access to P, then HKEM expects access to $\mathsf{P} \times \mathsf{P}_{\mathsf{rom}}$. Note that the random oracle must be "new" and cannot be queried by KEM. This is necessary as the KEM could otherwise query the random oracle on the key it will output and include that as part of the ciphertext. Oracle cloning [BDG20] can create multiple random oracles from a single random oracle.

Intuitively, CPA security is achieved if the attacker cannot predict k_{KEM} and query it to the random oracle, i.e., as long as the KEM is one-way secure.

Theorem 5.2 *If* KEM *is OW* secure with* P*, then* $\mathsf{HKEM}[\mathsf{KEM}]$ *is SIM*-AC-CPA secure with respect to* $\mathsf{P} \times \mathsf{P}_{\mathsf{rom}}$*.*

The full proof (and the formal definition of OW*) are given in Appendix C. The proof works as one would expect. The simulator produces ciphertexts by using KEM honestly. On exposures, it returns the keys/randomness it used and attempts to reprogram the random oracle to map keys encapsulated by KEM to the keys that were randomly sampled by the encapsulation oracle.

5.5 A Fujisaki-Okamoto-Style Transform

Finally, we consider a way to construct a CCA secure KEM from a one-way secure KEM. In particular, we look at part of one version of the Fujisaki-Okamoto transformation [FO13]. We work from the modular treatment of Hofheinz, Hövelmanns, and Kiltz [HHK17] (HHK), in particular showing that the transformation which they refer to as $\mathsf{U}^{\not\perp}$ achieves SIM*-AC-CCA security. This should extend to the other variants as well, but we have focused on one for simplicity. Slightly corrected versions of HHK's proofs can be found in [Höv21, Sec. 2.1-2.2].

Let KEM be a key encapsulation mechanism and F be a function family. Then we consider the scheme $\mathsf{U}^{\not\perp}[\mathsf{KEM}, \mathsf{F}]$ defined as follows. The key generation algorithm $\mathsf{U}^{\not\perp}[\mathsf{KEM}, \mathsf{F}].\mathsf{Kg}$ generates keys $(ek, dk) \leftarrow\!\!\$\ \mathsf{KEM}.\mathsf{Kg}(1^\lambda)$ and $fk \leftarrow\!\!\$\ \mathsf{F}.\mathsf{Kg}(1^\lambda)$, then outputs $(ek, (dk, fk))$.

```
$\underline{\mathsf{U}^{\not\perp}[\mathsf{KEM}, \mathsf{F}].\mathsf{Encaps}^{\mathsf{P}\times\mathsf{P}_{\mathsf{rom}}}(1^\lambda, ek)}$
$(c, k_{\mathsf{KEM}}) \leftarrow\!\!\$\ \mathsf{KEM}.\mathsf{Encaps}^{\mathsf{P}}(1^\lambda, ek)$
$k \leftarrow \mathsf{P}_{\mathsf{rom}}(k_{\mathsf{KEM}}, c)$; Return $(c, k)$
```

```
$\underline{\mathsf{U}^{\not\perp}[\mathsf{KEM}, \mathsf{F}].\mathsf{Decaps}^{\mathsf{P}\times\mathsf{P}_{\mathsf{rom}}}(1^\lambda, (dk, fk), c)}$
$k_{\mathsf{KEM}} \leftarrow \mathsf{KEM}.\mathsf{Decaps}^{\mathsf{P}}(1^\lambda, dk, c)$
If $k_{\mathsf{KEM}} \neq \perp$ then $k \leftarrow \mathsf{P}_{\mathsf{rom}}(k_{\mathsf{KEM}}, c)$
Else $k \leftarrow \mathsf{F}.\mathsf{Ev}^{\mathsf{P}\times\mathsf{P}_{\mathsf{rom}}}(1^\lambda, fk, c)$
Return $k$
```

Here $\mathsf{U}^{\not\perp}[\mathsf{KEM}, \mathsf{F}].\mathsf{K}(\lambda) = \mathsf{F}.\mathsf{Out}(\lambda) = \mathsf{P}_{\mathsf{rom}}.\mathcal{R}_\lambda$. Note that if KEM expects access to P, then $\mathsf{U}^{\not\perp}[\mathsf{KEM}, \mathsf{F}]$ expects access to $\mathsf{P} \times \mathsf{P}_{\mathsf{rom}}$. We allow F to have access to $\mathsf{P} \times \mathsf{P}_{\mathsf{rom}}$. It is important that KEM not have access to the random oracle used by the transform (otherwise it could, for example, ensure that it always produces output for which the first bit of $\mathsf{P}_{\mathsf{rom}}(k_{\mathsf{KEM}}, c)$ is 0 and thus distinguishable from random).

However, our results show there is no issue with F having access to the same random oracle used by $\mathsf{U}^{\not\perp}$. Indeed, HHK actually used the specific construction $\mathsf{F}.\mathsf{Ev}(1^\lambda, fk, c) = \mathsf{P}_{\mathsf{rom}}(fk, c)$. Considering an arbitrary F is more general. We emphasize the proof with this generality only works because we are using our new SIM*-AC security definitions. Moreover, given that caveat, this supports

Jaeger and Tyagi's motivation for introducing SIM-AC definitions because this modularity allows our proof to avoid the details of the random oracle analysis required to prove that $\mathsf{P}_{\mathsf{rom}}(k_{\mathsf{KEM}}, c)$ is secure.

Additionally HHK use a public-key encryption scheme applied to a random message in place of KEM. Again, $\mathsf{U}^{\not\perp}[\mathsf{KEM}, \mathsf{F}]$ is a generalization of this as the security they assume of the encryption scheme implies that the KEM obtained by encrypting a randome message satisfies the security we require.

HHK showed that the construction is IND-CCA secure as long as the underlying scheme achieves a variant of one-way security which provides to the attacker a plaintext checking oracle which decrypts a given ciphertext and returns a boolean indicating whether the result is the same as a given message. We show the same for our security definition.

Theorem 5.3 *If* KEM *is OW*-PCA secure with* P *and* F *is SIM*-AC-PRF secure with* $\mathsf{P} \times \mathsf{P}_{\mathsf{rom}}$*, then* $\mathsf{U}^{\not\perp}[\mathsf{KEM}, \mathsf{F}]$ *is SIM*-AC-CCA secure with* $\mathsf{P} \times \mathsf{P}_{\mathsf{rom}}$*.*

The full proof (and the formal definition of OW*) are given in Appendix C. HHK gave a transform T which transforms a OW secure PKE scheme into a OW-PCA secure PKE scheme. Interpreting this as a KEM in the natural manner gives a OW-PCA secure KEM.

6 Recovering Prior Results

Finally, we conclude by showing that the positive results Jaeger and Tyagi [JT20] established regarding various notions of SIM-AC security also hold with respect to our analogous SIM*-AC notions. For this, we divide the results of Jaeger and Tyagi into three general categories. This first category covers results where (non-SIM-AC) security of some "high-level" construction is shown assuming its constituent elements satisfy SIM-AC security. The second category covers results where SIM-AC security of some "intermediate-level" construction is shown assuming its constituent elements satisfy SIM-AC security. The final category covers results where SIM-AC security of some "low-level" primitive is shown by direct ideal model analysis.

6.1 High-level Proofs

The first category is the easiest in which to replace SIM-AC with SIM*-AC. In particular Jaeger and Tyagi showed: (1) SIM-AC-CPA secure encryption suffices for a version of the OPAQUE password-authenticated key exchange protocol of Jarecki, et al. [JKX18] (because the latter was proven secure assuming "equivocable encryption" which is a weaker notion than SIM-AC-CPA security), (2) SIM-AC-PRF secure PRFs and SIM-AC-KP secure encryption suffice for a searchable symmetric encryption scheme of Cash, et al. [CJJ+14], and (3) SIM-AC-CPA secure encryption suffices for the self-revocable cloud storage scheme of Tyagi, et al. [TMRM18]. We can recover these results with wSIM*-AC in place of SIM-AC by noting that our new notion is strictly stronger.

Lemma 6.1 *For* $\mathrm{X} \in \{\mathrm{PRF}, \mathrm{CPA}, \mathrm{KPA}, \$, \mathrm{CCA}, \mathrm{AE}\}$*, wSIM*-AC-X security implies SIM-AC-X security. The converse does not hold.*

This result follows from the fact that wSIM*-AC security strengthens adversaries (by allowing them to program the ideal primitive) and weakens simulators (by restricting them to explicitly program the ideal primitive rather than having complete control of it). For the converse, note that a SIM*-AC adversary can, e.g., break the one-way function or collision-resistance security of a random oracle by programming it appropriately. Hence, one can modify a SIM-AC secure scheme to be trivially insecure (e.g. reveal its secret key) when a collision in the random oracle is known. SIM-AC security will be maintained, but the modified scheme will not be SIM*-AC secure.

In each of the searchable symmetric encryption and BurnBox proof, Jaeger and Tyagi had to assume that the constituent elements each used separate ideal primitives. Using SIM*-AC definitions we could reproduce these results without the assumption of separate ideal primitives using the proof modifications we discussion for intermediate-level proofs.

6.2 Intermediate-level Proofs

In the second category, Jaeger and Tyagi gave security results for several encryption schemes. There is no general way to prove that these result carry over from SIM-AC to SIM*-AC security notions.[17] However, by examining the details of the proofs used for each of these result we can see that we are in luck. In each, the ideal primitive was used as a black-box. Constructed SIM-AC reduction adversaries provided the given SIM-AC adversaries with direct access to their own Prim oracle. The $\mathsf{S.Ls}$ algorithm of any constructed SIM-AC simulators S just ran the corresponding algorithms of the given SIM-AC simulators.

As such, modifying these proofs for SIM*-AC (or wSIM*-AC) requires only syntactic change to treat the ideal primitive as a black-box. Reduction adversaries provide their given adversaries with direct access to PPrim. Rather than having a $\mathsf{S.Ls}$ algorithm, SIM*-AC simulators will provide their given underlying simulators with direct access to PPrim. Otherwise the analysis follows as given.

In fact, in places where multiple SIM-AC primitive had to use separate ideal primitives, this black-box use of the primitives allow them to share the same primitive for SIM*-AC security without any extra effort.

Moreover, the only way in which constructed simulators depended on adversaries was through dependance on given simulators for the constituent algorithms (which were allowed to depend on the adversary per SIM-AC security). As such, there is no issue when using the order of quantification required for SIM-AC rather than wSIM*-AC security. Hence the following results hold.

Lemma 6.2 *Let $x \in \{\varepsilon, w\}$. Then the following hold.*

- *If SE is xSIM*-AC-CPA and INT-CTXT* secure with P, then SE is xSIM*-AC-CCA secure with P.*
- *If SE is xSIM*-AC-CPA secure with P and F is UF-CMA* secure with P, then $(\mathsf{SE}, \mathsf{F})$ encrypt-then-mac is xSIM*-AC-CCA secure with P.*
- *If $\mathsf{SE}[\cdot]$ is IND-AC-EXT secure and F is xSIM*-AC-PRF secure with P, then $\mathsf{SE}[\mathsf{F}]$ is xSIM*-AC-\$ secure with P.*

This last result covers modes of operation such as counter (CTR), cipher-block chaining (CBC), cipher feedback (CFB), and output feedback (OFB) mode.

The asterisks added to INT-CTXT and UF-CMA indicate that we need these security notions to hold *even for adversaries who are able to program the ideal primitive.* We note, for example, that UF-CMA* security is implied by SIM*-AC-PRF security. We similarly expect that schemes which are known to achieve INT-CTXT security when constructed from a PRF secure function family can be shown by essentially the same proof to achieve INT-CTXT* security when using a SIM*-AC-PRF secure function family.

[17]This follows from the counter-example described above where we construct a scheme which is trivially insecure if a collision in the random oracle is known.

6.3 Low-level Proofs

For the third category, Jaeger and Tyagi used information theoretic analysis to show that random oracles are SIM-AC-PRF secure, ideal ciphers are SIM-AC-PRF secure, and the ideal encryption model [TMRM18] is SIM-AC-AE secure.[18] To re-establish these results one technically has to re-write the proofs. We will sketch how to modify the SIM-AC proofs for the first two of these (formal proofs are given in the appendices).

Lemma 6.3 *Random oracles are SIM*-AC-PRF secure (assuming $|\mathcal{K}_\lambda|$ is super-polynomial) and ideal ciphers are SIM*-AC-PRF secure (assuming $|\mathcal{K}_\lambda|$ and $|\mathcal{D}_\lambda|$ are super-polynomial).*

The simulators given for both work by honestly simulating the ideal primitive except whenever a new users is exposed they sample the key at random and then program the primitive to be consistent with the random values returned by earlier evaluation queries. This can done with the more restricted SIM*-AC syntax for simulators. These simulators do not depend on the adversary.

If $\mathsf{F} : \mathcal{K}_\lambda \times \mathcal{D}_\lambda \to \mathcal{R}_\lambda$, the analysis of Jaeger and Tyagi showed that

$$\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S}_{\mathsf{prf}},\mathsf{P}_{\mathsf{rom}},\mathcal{A}}(\lambda) \leq \frac{u_\lambda^2}{|\mathcal{K}_\lambda|} + \frac{u_\lambda p_\lambda}{|\mathcal{K}_\lambda|} \text{ and}$$

$$\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S}_{\mathsf{prf}},\mathsf{P}_{\mathsf{icm}},\mathcal{A}}(\lambda) \leq \frac{u_\lambda^2}{|\mathcal{K}_\lambda|} + \frac{u_\lambda p_\lambda}{|\mathcal{K}_\lambda|} + \frac{q_\lambda^2}{2|\mathcal{D}_\lambda|}.$$

Here u_λ is the number of distinct users $\mathcal{A}$ interacts with, p_λ is the number of ideal primitive queries it makes, and q_λ is the number of evaluation queries it makes. Each summand represents a bound of the probability that a bad event occurs which could let an adversary distinguish the real and simulated worlds. The first corresponds to distinct users choosing the same random key. The second corresponds to the attacker making an ideal primitive query with an unexposed user's key. The third corresponds to random outputs of Ev colliding.

For SIM*-AC-PRF/PRP security of these constructions the only additional bad event we will have to analyze is that of the attacker happening to make an ideal model *programming* query using an unexposed user's key. If p'_λ denotes the number of programming queries the attacker makes, this just adds an additional term of $u_\lambda p'_\lambda/|\mathcal{K}_\lambda|$ to either bound. Alternatively, we could leave the bound unchanged and redefine p_λ to include programming queries as well.

In Appendix D, we give (a generalization of) the proof for random oracles. In Appendix E, we give the proof for ideal ciphers. We first introduce a notion of SIM*-AC-PRP security and prove that it is achieved by an ideal cipher (this proof is also a special case of the general proof given in Appendix D). Then, naturally, SIM*-AC-PRP and SIM*-AC-PRF security are equivalent up to a birthday bound.

Acknowledgments

We thank Nirvan Tyagi for their collaboration on [JT20], early drafts of which considered asymmetric encryption. Some definitions and results in Section 5 grew out of discussions about those drafts.

[18]The last of these is a slight "cheat" as the ideal encryption model does not satisfy their (or our) definitions of what an ideal primitive is.

References

[BBT16] Mihir Bellare, Daniel J. Bernstein, and Stefano Tessaro. Hash-function based PRFs: AMAC and its multi-user security. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 566–595. Springer, Heidelberg, May 2016. doi:10.1007/978-3-662-49890-3_22. 4, 18

[BCK96a] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 1–15. Springer, Heidelberg, August 1996. doi:10.1007/3-540-68697-5_1. 4, 18

[BCK96b] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *37th FOCS*, pages 514–523. IEEE Computer Society Press, October 1996. doi:10.1109/SFCS.1996.548510. 4, 18

[BDG20] Mihir Bellare, Hannah Davis, and Felix Günther. Separate your domains: NIST PQC KEMs, oracle cloning and read-only indifferentiability. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 3–32. Springer, Heidelberg, May 2020. doi:10.1007/978-3-030-45724-2_1. 4, 22

[BDWY12] Mihir Bellare, Rafael Dowsley, Brent Waters, and Scott Yilek. Standard security does not imply security against selective-opening. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 645–662. Springer, Heidelberg, April 2012. doi:10.1007/978-3-642-29011-4_38. 2

[Bel06] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 602–619. Springer, Heidelberg, August 2006. doi:10.1007/11818175_36. 4, 18

[BHK12] Florian Böhl, Dennis Hofheinz, and Daniel Kraschewski. On definitions of selective opening security. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 522–539. Springer, Heidelberg, May 2012. doi:10.1007/978-3-642-30057-8_31. 2

[BHS24] Carlo Brunetta, Hans Heum, and Martijn Stam. SoK: Public key encryption with openings. In Qiang Tang and Vanessa Teague, editors, *PKC 2024, Part II*, volume 14604 of *LNCS*, pages 35–68. Springer, Heidelberg, April 2024. doi:10.1007/978-3-031-57728-4_2. 3, 20

[BHY09] Mihir Bellare, Dennis Hofheinz, and Scott Yilek. Possibility and impossibility results for encryption and commitment secure under selective opening. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 1–35. Springer, Heidelberg, April 2009. doi:10.1007/978-3-642-01001-9_1. 2

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. doi:10.1145/168588.168596. 14, 22

[CDG$^+$18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 280–312. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78381-9_11. 2, 14, 20

[CFGN96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *28th ACM STOC*, pages 639–648. ACM Press, May 1996. doi:10.1145/237814.238015. 2

[CJ24] Kaishuo Cheng and Joseph Jaeger. Adaptive security for graph-based games in ideal models (unpublished manuscript). 2024. 8

[CJJ+14] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*. The Internet Society, February 2014. 2, 4, 23

[CLNS16] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. Virtual smart cards: How to sign with a password and a server. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16*, volume 9841 of *LNCS*, pages 353–371. Springer, Heidelberg, August / September 2016. doi:10.1007/978-3-319-44618-9_19. 5

[CLNS17] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. UC-secure non-interactive public-key encryption. In Boris Köpf and Steve Chong, editors, *CSF 2017 Computer Security Foundations Symposium*, pages 217–233. IEEE Computer Society Press, 2017. doi:10.1109/CSF.2017.14. 2, 5, 18, 20

[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. doi:10.1137/S0097539702403773. 5, 21

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, January 2013. doi:10.1007/s00145-011-9114-1. 5, 22

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986. doi:10.1145/6490.6503. 4, 18

[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, Heidelberg, November 2017. doi:10.1007/978-3-319-70500-2_12. 5, 22, 42

[HJKS15] Felix Heuer, Tibor Jager, Eike Kiltz, and Sven Schäge. On the selective opening security of practical public-key encryption schemes. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 27–51. Springer, Heidelberg, March / April 2015. doi:10.1007/978-3-662-46447-2_2. 6

[Höv21] Kathrin Hövelmanns. *Generic constructions of quantum-resistant cryptosystems*. PhD thesis, Dissertation, Bochum, Ruhr-Universität Bochum, 2020, 2021. doi:10.13154/294-7758. 22, 42

[HP16] Felix Heuer and Bertram Poettering. Selective opening security from simulatable data encapsulation. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 248–277. Springer, Heidelberg, December 2016. doi:10.1007/978-3-662-53890-6_9. 2, 6

[HPW15] Carmit Hazay, Arpita Patra, and Bogdan Warinschi. Selective opening security for receivers. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 443–469. Springer, Heidelberg, November / December 2015. doi:10.1007/978-3-662-48797-6_19. 2, 5

[HRW16] Dennis Hofheinz, Vanishree Rao, and Daniel Wichs. Standard security does not imply indistinguishability under selective opening. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 121–145. Springer, Heidelberg, October / November 2016. doi:10.1007/978-3-662-53644-5_5. 2

[JKRS21] Tibor Jager, Eike Kiltz, Doreen Riepel, and Sven Schäge. Tightly-secure authenticated key exchange, revisited. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 117–146. Springer, Heidelberg, October 2021. doi:10.1007/978-3-030-77870-5_5. 20

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78372-7_15. 23

[JT20] Joseph Jaeger and Nirvan Tyagi. Handling adaptive compromise for practical encryption schemes. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2020. doi:10.1007/978-3-030-56784-2_1. 2, 3, 9, 11, 15, 18, 23, 25, 59

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 111–126. Springer, Heidelberg, August 2002. doi:10.1007/3-540-45708-9_8. 2, 3

[Pan07] Saurabh Panjwani. Tackling adaptive corruptions in multicast encryption protocols. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 21–40. Springer, Heidelberg, February 2007. doi:10.1007/978-3-540-70936-7_2. 2

[TMRM18] Nirvan Tyagi, Muhammad Haris Mughees, Thomas Ristenpart, and Ian Miers. BurnBox: Self-revocable encryption in a world of compelled access. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 445–461. USENIX Association, August 2018. 23, 25

[Wee09] Hoeteck Wee. Zero knowledge in the random oracle model, revisited. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 417–434. Springer, Heidelberg, December 2009. doi:10.1007/978-3-642-10366-7_25. 14

A Cascade Proof (Theorem 4.3)

Proof: We will, without loss of generality, restrict attention to attackers that never makes queries to Ev or Exp for a user that has already been exposed and that never makes queries to Exp for a user without having first made Ev queries to that user. It will be convenient to introduce some notation. If $\vec{x} = (\vec{x}_1, \ldots, \vec{x}_m)$ is a vector, then $\vec{x}_{\leq i} = (\vec{x}_1, \ldots, \vec{x}_i)$ is its length i prefix and $\vec{x} \,\|\, x$ is the vector $(\vec{x}_1, \ldots, \vec{x}_m, x)$. It will be convenient to think of the following notation for F^n.

```
$\underline{\mathsf{F^n.Ev}(1^\lambda, k_{u,()}, \vec{x})}$
For $i = 0, \ldots, n(\lambda) - 1$ do
    $k_{u,\vec{x}_{\leq i+1}} \leftarrow \mathsf{F.Ev}(1^\lambda, k_{u,\vec{x}_{\leq i}}, \vec{x}_{i+1})$
Return $k_{u,\vec{x}}$
```

Here all the (sub-)keys used by F^n are indexed by a user and the prefix of the input which has been processed prior to the use of that key. Note then that $k_{u,()}$ is the user's starting key (which would normally be called k_u) because () is the empty vector. For each $\vec{z}$ of length at most n, we will think of $k_{u,\vec{z}}$ as a key belonging to a user $(u, \vec{z})$.

We can think of there being an underlying forest of rooted trees of keys where each tree corresponds to some u. A node of a tree is labelled by $(u, \vec{z})$ and (if $|\vec{z}| < n$) has an edge connecting to $(u, \vec{z} \,\|\, x)$ for each $x \in \mathsf{F.Inp}$. We refer to the former node at the parent of the latter, which is its child. This edge represents that fact that (in the real world) $(u, \vec{z} \,\|\, x)$'s key is derived using $(u, \vec{z})$'s key. An attacker's Ev oracle allows it to request the keys of leaf nodes. Its Exp oracle allows it to request the key of the root of a tree. In the real world, it could then compute forward along the paths to each of the queried leaf nodes in this tree to see that it does indeed compute the same keys it saw earlier. In the ideal world, leaf node keys are picked at random (because u is unexposed). We need to construct a simulator which (given the keys of the leaf nodes the attacker has previously requested) is able to produce a consistent key for the root node.

Let $\mathsf{S_F}$ be the simulator that is guaranteed to exist by the security of F. We will define a simulator S for F^n, that internally runs $n(\lambda)$ copies of $\mathsf{S_F}$ (using states $\sigma_0, \ldots, \sigma_{n(\lambda)-1}$). The $n-1$ copy of this simulator (using $\sigma_{n(\lambda)-1}$) will be given the keys of the queried leaf nodes of the tree (which have depth n) and use them to produce the depth $n-1$ keys. Specifically, consider the subtree consisting of all the paths from the root to the queried leaves. For each depth $n-1$ node $(u, \vec{x}_{\le n-1})$ in this subtree, S will give the $n-1$ copy of $\mathsf{S_F}$ the keys of all the child leaf nodes $(u, \vec{x}_{\le n-1} \,\|\, x)$ so that it can produce a key for this node. Once we've computed the keys for every depth $n-1$ node in the subtree, we continue propagating toward the root using the $n-2$ copy of the simulator to produce keys for the nodes at depth $n-2$ from the keys of their children nodes at depth $n-1$. We continue similarly until we've computed the root node key.

Formal pseudocode for our simulator is as follows.

```
$\underline{\mathsf{S.Init}(1^\lambda)}$
$\sigma_{(\cdot)} \leftarrow\!\!\$\ \mathsf{S_F.Init}(1^\lambda)$
Return $\sigma_{(\cdot)}$

$\underline{\mathsf{S.Exp}^{\text{PPRIM}}(1^\lambda, u, T_u : \sigma_{(\cdot)})}$
For $\vec{x} \in \{\, \vec{x} \,:\, T_u[\vec{x}] \ne \bot \,\}$ do
    $k_{u,\vec{x}} \leftarrow T_u[\vec{x}]$
For $i = n(\lambda)-1, n(\lambda)-2, \ldots, 0$ do
    For $\vec{z} \in \{\, \vec{x}_{\le i} \,:\, T_u[\vec{x}] \ne \bot \,\}$ do
        $T_{u,\vec{z}}[\cdot] \leftarrow k_{u,\vec{z}\,\|\,(\cdot)}$
        $k_{u,\vec{z}} \leftarrow\!\!\$\ \mathsf{S_F.Exp}^{\text{PPRIM}}(1^\lambda, (u,\vec{z}), T_{u,\vec{z}} : \sigma_i)$
Return $k_{u,()}$
```

Note that we do not specify $\mathsf{S.Ev}$ because by our assumption the adversary never queries Ev with a user that has previously been exposed and so $\mathsf{S.Ev}$ is never used. The first loop of $\mathsf{S.Exp}$ extracts the keys $k_{u,\vec{x}}$ for each leaf node from the table T_u. Then the second for loop iterates through the subtree from leaves to root, with iteration i being used to simulate the keys of the depth i nodes. The inner for loop iterates over each node $(u, \vec{z})$ at depth i by looking at all $\vec{z}$ that are of the form $\vec{x}_{\le i}$ for some $\vec{x}$ such that $T_u[\vec{x}] \ne \bot$. It uses the i copy of the simulator to simulate the key of this node. The table $T_{u,\vec{z}}$ given as input to this copy of the simulator is defined so that $T_{u,\vec{z}}[z] = k_{u,\vec{z}\,\|\,z}$ for each $z \in \mathsf{F.Inp}$. From earlier iterations, this is non-$\bot$ iff $\vec{z} \,\|\, z$ is the prefix of some $\vec{x}$ for which $T_u[\vec{x}] \ne \bot$.

Now let $\mathcal{A}_{\mathsf{prf}}$ be an adversary attacking F^n. We will reduce its advantage against S to that of a related attacker $\mathcal{A}_{\mathsf{F}}$ against $\mathsf{S_F}$ simulating for F. This reduction will following a typical hybrid argument form, where we hybrid over layers of the tree at different depths.

In particular, we define the hybrids H_l for $0 \le l \le n$ as shown in Fig. 6. In hybrid H_l, all keys at depth l are sampled at random by the game. Decedent nodes (depth greater than l) are computed from these using F and ancestors nodes (depth less than l) are simulated from these using $\mathsf{S_F}$. So then H_0 perfectly matches the real world of $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prf}}$ and H_n perfectly matches the ideal world using simulator S. Noting that these hybrids return whether $b' = 1$, standard calculations give $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F,S,P},\mathcal{A}_{\mathsf{prf}}}(\lambda) = \Pr[\mathrm{H}_0(\lambda)] - \Pr[\mathrm{H}_{n(\lambda)}(\lambda)] = \sum_{l=1}^{n(\lambda)} \Pr[\mathrm{H}_{l-1}(\lambda)] - \Pr[\mathrm{H}_l(\lambda)]$.

Now the attacker $\mathcal{A}_{\mathsf{F}}$, is shown in Fig. 6. It samples t between 1 and n, then simulates the view of $\mathcal{A}$. Its goal in this simulation is if the secret bit in the game it is playing is b, then the view it provides to $\mathcal{A}$ perfectly matches that from hybrid H_{t-b}.

It achieves this as follows. In simulating the evaluation oracle it uses its own Ev oracle to obtain the depth t key, then uses F to compute deeper keys. In simulating the exposure oracle it uses its

```
Hybrids $\mathrm{H}_l(\lambda)$, $0 \le l \le n(\lambda)$
For $\vec{z} \in \mathsf{F.Inp}(\lambda)^l$ do
    $k_{(\cdot),\vec{z}} \leftarrow\$ \mathsf{F.K}(\lambda)$
$\sigma_{\mathsf{P}} \leftarrow\$ \mathsf{P.Init}(1^\lambda)$
For $i = 0, \ldots, l-1$ do
    $\sigma_i \leftarrow\$ \mathsf{S_F.Init}(1^\lambda)$
$b' \leftarrow\$ \mathcal{A}_{\mathsf{prf}}^{\mathrm{Ev},\mathrm{Exp},\mathrm{PPrim}}(1^\lambda)$
Return $(b' = 1)$

$\mathrm{PPrim}(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$y \leftarrow\$ \mathsf{P.Op}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})$
Return $y$

$\mathrm{Ev}(u, \vec{x})$
For $i = l, l+1, \ldots, n(\lambda)-1$ do
    $k_{u,\vec{x}_{\le i+1}} \leftarrow \mathsf{F.Ev}(1^\lambda, k_{u,\vec{x}_{\le i}}, \vec{x}_{i+1})$
$T_u[\vec{x}] \leftarrow k_{u,\vec{x}}$
Return $k_{u,\vec{x}}$

$\mathrm{Exp}(u)$
For $i = l-1, l-2, \ldots, 0$ do
    For $\vec{z} \in \{\, \vec{x}_{\le i} \,:\, T_u[\vec{x}] \ne \bot \,\}$ do
        $T_{u,\vec{z}}[\cdot] \leftarrow k_{u,\vec{z}\,\|\,(\cdot)}$
        $k_{u,\vec{z}} \leftarrow\$ \mathsf{S_F.Exp}^{\mathrm{PPrim}}(1^\lambda, (u,\vec{z}), T_{u,\vec{z}} : \sigma_i)$
Return $k_{u,()}$
```

```
Adversary $\mathcal{A}_{\mathsf{F}}^{\mathrm{Ev},\mathrm{Exp},\mathrm{PPrim}}(\lambda)$
$t \leftarrow\$ \{1, \ldots, n(\lambda)\}$
For $i = 0, \ldots, t-2$ do
    $\sigma_i \leftarrow\$ \mathsf{S_F.Init}(1^\lambda)$
$b' \leftarrow\$ \mathcal{A}^{\mathrm{EvSim},\mathrm{ExpSim},\mathrm{PPrim}}(1^\lambda)$
Return $b'$

$\mathrm{EvSim}(u, \vec{x})$
$k_{u,\vec{x}_{\le t}} \leftarrow \mathrm{Ev}((u, \vec{x}_{\le t-1}), \vec{x}_t)$
For $i = t, \ldots, n(\lambda)-1$ do
    $k_{u,\vec{x}_{\le i+1}} \leftarrow \mathsf{F.Ev}(1^\lambda, k_{u,\vec{x}_{\le i}}, \vec{x}_{i+1})$
$T_u[\vec{x}] \leftarrow k_{u,\vec{x}}$
Return $k_{u,\vec{x}}$

$\mathrm{ExpSim}(u)$
For $\vec{z} \in \{\, \vec{x}_{\le t-1} \,:\, T_u[\vec{x}] \ne \bot \,\}$ do
    $k_{u,\vec{z}} \leftarrow \mathrm{Exp}((u, \vec{z}))$
For $i = t-2, \ldots, 0$ do
    For $\vec{z} \in \{\, \vec{x}_{\le i} \,:\, T_u[\vec{x}] \ne \bot \,\}$ do
        $T_{u,\vec{z}}[\cdot] \leftarrow k_{u,\vec{z}\,\|\,(\cdot)}$
        $k_{u,\vec{z}} \leftarrow\$ \mathsf{S_F.Exp}^{\mathrm{PPrim}}(1^\lambda, (u,\vec{z}), T_{u,\vec{z}} : \sigma_i)$
Return $k_{u,()}$
```

Figure 6: **Left**: Hybrid games used for proving security of F^n. In H_0's oracle Exp, the for loop is skipped as i starts with a value less than zero. **Right**: Adversary against F for proof that F^n is secure

own Exp to obtain the depth $t-1$ keys, then uses its internal copies of $\mathsf{S_F}$ to compute shallower keys. When $b = 1$ the depth $t-1$ keys are chosen at random and the depth t keys are computed from F. When $b = 0$ the depth t keys are chosen at random and the depth $t-1$ keys are chosen by $\mathsf{S_F}$.

Hence $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S_F},\mathsf{P},\mathcal{A}_{\mathsf{F}}}(\lambda) = \mathbf{E}_t[\Pr[\mathrm{H}_{t-1}(\lambda)]] - \mathbf{E}_t[\Pr[\mathrm{H}_t(\lambda)]] = (1/n(\lambda)) \cdot \mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{prf}}}(\lambda)$, where $\mathbf{E}_t$ denotes expectation over $t \leftarrow\$ \{1, \ldots, n(\lambda)\}$, completing the proof.▌

B KEM/DEM Proof (Theorem 5.1)

We consider SIM*-AC-CCA security. Other notions follow by ignoring the appropriate oracle(s). Let KEM, SE, and adversary $\mathcal{A}_{\mathsf{cca}}$ be given. We will assume that $\mathcal{A}_{\mathsf{cca}}$ never queries encryption for an exposed user and never queries challenge ciphertexts to decryption. We will prove the security of $\mathsf{KD}[\mathsf{KEM}, \mathsf{SE}]$ using a sequence of four hybrid games (G_0 through G_3).

KEM reduction. Consider the first two hybrid games, shown in Fig. 7, and in particular game G_0 which contains the $\boxed{\text{boxed}}$ but not the highlighted code. This game was created by inserting the code of KD into the real world of the SIM*-AC-CCA game, then making some minor code

```
Games $\boxed{\mathrm{G}_0(\lambda)}$, $\mathrm{G}_1(\lambda)$
  $\boxed{(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow\$ \mathsf{KEM.Kg}(1^\lambda)}$
  $\sigma_{\mathsf{P}} \leftarrow\$ \mathsf{P.Init}(1^\lambda)$
  $\sigma_{\mathsf{K}} \leftarrow\$ \mathsf{S_K.Init}(1^\lambda)$
  $b' \leftarrow\$ \mathcal{A}_{\mathsf{cca}}^{\mathrm{Ek},\mathrm{Enc},\mathrm{Dec},\mathrm{SExp},\mathrm{RExp},\mathrm{PPrim}}(1^\lambda)$
  Return $(b' = 1)$

$\mathrm{Ek}(u)$
  $\boxed{ek' \leftarrow ek_u}$
  $ek' \leftarrow\$ \mathsf{S_K.Ek}^{\mathrm{PPrim}}(1^\lambda, u : \sigma_{\mathsf{K}})$
  Return $ek'$

$\mathrm{RExp}(u)$
  $\boxed{dk' \leftarrow dk_u}$
  $dk' \leftarrow\$ \mathsf{S_K.RExp}^{\mathrm{PPrim}}(1^\lambda, u, K_u : \sigma_{\mathsf{K}})$
  Return $dk'$

$\mathrm{SExp}(u, i)$
  $\boxed{r_{\mathsf{K}} \leftarrow R_u[i]}$
  $r_{\mathsf{K}} \leftarrow\$ \mathsf{S_K.SExp}^{\mathrm{PPrim}}(1^\lambda, u, i, K_u[i] : \sigma_{\mathsf{K}})$
  $(\cdot, k) \leftarrow K_u[i]$; $c_{\mathsf{D}} \leftarrow C_u[i]$
  $r_{\mathsf{D}} \leftarrow \mathsf{SE.CExt}^{\mathsf{P}}(1^\lambda, k, c_{\mathsf{D}})$
  Return $r_{\mathsf{K}} \,\|\, r_{\mathsf{D}}$

$\mathrm{Enc}(u, m)$
  $\boxed{r \leftarrow\$ \mathsf{KEM.Rand}(\lambda)}$
  $\boxed{(c_{\mathsf{K}}, k) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek_u; r)}$
  $(c_{\mathsf{K}}, \cdot) \leftarrow\$ \mathsf{S_K.Encaps}^{\mathrm{PPrim}}(1^\lambda, u : \sigma_{\mathsf{K}})$
  $k \leftarrow\$ \mathsf{KEM.K}(\lambda)$
  $c_{\mathsf{D}} \leftarrow\$ \mathsf{SE.Enc}^{\mathsf{P}}(1^\lambda, k, m)$
  $R_u.\mathsf{add}(r)$; $K_u.\mathsf{add}(c_{\mathsf{K}}, k)$; $C_u.\mathsf{add}(c_{\mathsf{D}})$
  Return $(c_{\mathsf{K}}, c_{\mathsf{D}})$

$\mathrm{Dec}(u, (c_{\mathsf{K}}, c_{\mathsf{D}}))$
  If $K_u\langle c_{\mathsf{K}}\rangle \neq \perp$ then $k \leftarrow K_u\langle c_{\mathsf{K}}\rangle$
  $\boxed{\text{Else } k \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c_{\mathsf{K}})}$
  Else $k \leftarrow\$ \mathsf{S_K.Decaps}^{\mathrm{PPrim}}(1^\lambda, u, c_{\mathsf{K}} : \sigma_{\mathsf{K}})$
  If $k = \perp$ then return $\perp$
  $m \leftarrow \mathsf{SE.Dec}^{\mathsf{P}}(1^\lambda, k, c_{\mathsf{D}})$
  Return $m$
```

```
Adversary $\mathcal{A}_{\mathsf{K}}^{\mathrm{Ek},\mathrm{Encaps},\mathrm{Decaps},\mathrm{SExp},\mathrm{RExp},\mathrm{PPrim}}$
  $b' \leftarrow\$ \mathcal{A}_{\mathsf{cca}}^{\mathrm{Ek},\mathrm{EncSim},\mathrm{DecSim},\mathrm{SExpSim},\mathrm{RExp},\mathrm{PPrim}}(1^\lambda)$
  Return $b'$

$\mathrm{SExpSim}(u, i)$
  $\boxed{r_{\mathsf{K}} \leftarrow \mathrm{SExp}(u, i)}$
  $(\cdot, k) \leftarrow K_u[i]$; $c_{\mathsf{D}} \leftarrow C_u[i]$
  $r_{\mathsf{D}} \leftarrow \mathsf{SE.CExt}^{\mathsf{P}}(1^\lambda, k, c_{\mathsf{D}})$
  Return $r_{\mathsf{K}} \,\|\, r_{\mathsf{D}}$

$\mathrm{EncSim}(u, m)$
  $\boxed{(c_{\mathsf{K}}, k) \leftarrow \mathrm{Encaps}(u)}$
  $c_{\mathsf{D}} \leftarrow\$ \mathsf{SE.Enc}^{\mathsf{P}}(1^\lambda, k, m)$
  $K_u.\mathsf{add}(c_{\mathsf{K}}, k)$; $C_u.\mathsf{add}(c_{\mathsf{D}})$
  Return $(c_{\mathsf{K}}, c_{\mathsf{D}})$

$\mathrm{DecSim}(u, (c_{\mathsf{K}}, c_{\mathsf{D}}))$
  $\boxed{k \leftarrow \mathrm{Decaps}(u, c_{\mathsf{K}})}$
  If $k = \perp$ then return $\perp$
  $m \leftarrow \mathsf{SE.Dec}^{\mathsf{P}}(1^\lambda, k, c_{\mathsf{D}})$
  Return $m$
```

Figure 7: First two hybrids and corresponding reduction in security proof for the KEM/DEM construction. Oracle PPRIM is omitted as it is unchanged from the SIM*-AC-CPA/CCA security game. Giving P as an oracle means access to $\mathrm{PPrim}(\mathsf{Ls}, \cdot, \cdot, \cdot)$.

simplifications.[19] This included making explicit the logic that decryption returns $\perp$ if the decapsulation algorithm does. We added a list K_u to the encryption and decryption oracles. It stores lists of challenge KEM ciphertexts and the underlying keys. In decryption, if K_u has a key for the KEM ciphertext, we use that rather than recovering the key through decapsulation. Under the assumption that KEM is perfectly correct and query consistent, this does not change the behavior of the game. The former is used to ensure that decapsulation would give the same key obtained by encapsulation and the later is used to ensure that decapsulation would not make any ideal primitive

[19]Our assumption that the attacker does not make encapsulation queries for exposed users let us remove the list X and simplify logic around it. Our assumption that the attacker never forwards challenge ciphertexts from encryption to decryption let us remove the list M_u and simplify logic around it.

queries that were not previously made (so that the attacker cannot detect that decapsulation is not being run and making queries). Finally, rather than remembering the randomness used by SE, we remembered the ciphertext in table C_u and extracted the randomness inside of the exposure oracle. The assumed query consistency of $\mathsf{SE.CExt}$ ensures this does not make any additional primitive queries and so is undetectable. Hence, $\Pr[\mathrm{G}_0(\lambda)] = \Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{KD,S,P},\mathcal{A}_{\mathsf{cca}},1}(\lambda)]$.

Now consider the hybrid game G_1 which contains the highlighted, but not the $\boxed{\text{boxed}}$ code. The changes from G_0 switch from using honest values generated by the KEM to using simulation by a simulator $\mathsf{S_K}$. Unsurprisingly then, the difference between these games can be bounded by the security of the KEM. This is captured by adversary $\mathcal{A}_{\mathsf{K}}$, shown in the same figure, which uses its own oracles to simulate the view of $\mathcal{A}_{\mathsf{cca}}$ in these hybrids (notice it gives direct access to EK, REXP, and PPRIM). We used dashed boxes to indicate where its code was modified from the hybrid games. When the bit in its underlying game is b, it perfectly simulates G_{1-b} and hence we have that $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{KEM,S_K,P},\mathcal{A}_{\mathsf{K}}}(\lambda) = \Pr[\mathrm{G}_0(\lambda)] - \Pr[\mathrm{G}_1(\lambda)]$.

DEM reduction. Now consider the final two hybrid games, shown in Fig. 8, and in particular game G_2 which contains the $\boxed{\text{boxed}}$ but not the highlighted code. It was created by making syntactic changes to G_1 that do not change the behavior of the game, so $\Pr[\mathrm{G}_2(\lambda)] = \Pr[\mathrm{G}_1(\lambda)]$.

The main changes concern the random keys the were sampled from $\mathsf{KEM.K}$ inside encryption. The key that would be sampled in the i-th query to user u is now referred to as $k_{(u,i)}$. The code to sample it is at the beginning of the game (using $\mathsf{SE.Kg}$ rather than $\mathsf{KEM.K}$, which we assumed to be the same thing). Per user counters i_u track how many encryption queries have been made. Rather than storing keys in $K_u\langle c_{\mathsf{K}}\rangle$ we store the appropriate counter value in $I_u\langle c_{\mathsf{K}}\rangle$. If $I_u\langle c_{\mathsf{K}}\rangle = i$, then we know that $K_u\langle c_{\mathsf{K}}\rangle$ would have stored $k_{(u,i)}$. We've changed the code inside of decapsulation to match this and we copied the code that occurred after the if statement to instead occur inside of both branches (and then simplified slightly).

There is one place, however, where we still need K_u, which is to give it as input to $\mathsf{S_K}$ in REXP. So, inside of REXP we use a for loop to iterate over all queries to this user so far and recreate K_u. We set K_u back to an empty table afterwards so a fresh K_u is used each time. (This resetting is omitted in our coming simulator, as it is implicit from that fact it does not store the table as part of its state.) A smaller change we made is to store both c_{K} and c_{D} in C_u. These are used in SEXP for recreating values $K_u[i]$ for $\mathsf{S_K}$ and extracting randomness from c_{D} in REXP.

For the next transition, we will think of the key $k_{(u,i)}$ as belonging to a DEM user known as (u, i). Consider the hybrid game G_3 which contains the highlighted, but not the $\boxed{\text{boxed}}$ code. The changes from G_2 switch so that honest values generated by the DEM with $k_{(u,i)}$ are instead simulated by a simulator $\mathsf{S_D}$ told to simulate user (u, i). Unsurprisingly then, the difference between these games can be bounded by the security of the DEM. This is captured by adversary $\mathcal{A}_{\mathsf{D}}$, shown in the same figure, which uses its own oracles to simulate the view of $\mathcal{A}_{\mathsf{cca}}$ in these hybrids (notice it gives direct access to PPRIM). We used dashed boxes to indicate where its code was modified from these hybrid games. When the bit in its underlying game is b, it perfectly simulates G_{3-b} and hence we have that $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{SE,S_D,P},\mathcal{A}_{\mathsf{D}}}(\lambda) = \Pr[\mathrm{G}_2(\lambda)] - \Pr[\mathrm{G}_3(\lambda)]$.

Defining the simulator. Finally we define our simulator S. Naturally, this simulator is defined so that $\Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{KD,S,P},\mathcal{A}_{\mathsf{cca}},0}(\lambda)] = \Pr[\mathrm{G}_3(\lambda)]$. It is defined as follows. Mostly the code is copied directly from G_3. The one interesting change arises in encryption because it does not know the message and so cannot fill $M_{(u,i)}$ appropriately. The simulator will create this later on exposures when it is given M_u. Therein it iterates over the entries of M_u in order, which correspond to the encryption queries made to u. It uses the i-th entry to fill $M_{(u,i)}$.

```
Games $\boxed{\mathrm{G}_2(\lambda)}$,$\mathrm{G}_3(\lambda)$
$i_{(\cdot)} \leftarrow 0$
$\boxed{k_{(\cdot,\cdot)} \leftarrow_{\$} \mathsf{SE.Kg}(\lambda)}$
$\sigma_{\mathsf{P}} \leftarrow_{\$} \mathsf{P.Init}(1^\lambda)$
$\sigma_{\mathsf{K}} \leftarrow_{\$} \mathsf{S_K.Init}(1^\lambda)$
$\sigma_{\mathsf{D}} \leftarrow_{\$} \mathsf{S_D.Init}(1^\lambda)$
$b' \leftarrow_{\$} \mathcal{A}_{\mathsf{cca}}^{\mathrm{Ek,Enc,Dec,SExp,RExp,PPrim}}(1^\lambda)$
Return $(b' = 1)$

$\mathrm{Ek}(u)$
$ek' \leftarrow_{\$} \mathsf{S_K.Ek}^{\mathrm{PPrim}}(1^\lambda, u : \sigma_{\mathsf{K}})$
Return $ek'$

$\mathrm{RExp}(u)$
For $(c_{\mathsf{K}}, i) \in I_u$ do
    $\boxed{k \leftarrow k_{(u,i)}}$
    $k \leftarrow_{\$} \mathsf{S_D.Exp}^{\mathrm{PPrim}}(1^\lambda, (u,i), M_{(u,i)} : \sigma_{\mathsf{D}})$
    $K_u.\mathsf{add}(c_{\mathsf{K}}, k)$
$dk' \leftarrow_{\$} \mathsf{S_K.RExp}^{\mathrm{PPrim}}(1^\lambda, u, K_u : \sigma_{\mathsf{K}})$
$K_u \leftarrow [\cdot]$; Return $dk'$

$\mathrm{SExp}(u, i)$
$(c_{\mathsf{K}}, c_{\mathsf{D}}) \leftarrow C_u[i]$
$\boxed{k \leftarrow k_{(u,i)}}$
$k \leftarrow_{\$} \mathsf{S_D.Exp}^{\mathrm{PPrim}}(1^\lambda, (u,i), M_{(u,i)} : \sigma_{\mathsf{D}})$
$r_{\mathsf{K}} \leftarrow_{\$} \mathsf{S_K.SExp}^{\mathrm{PPrim}}(1^\lambda, u, i, (c_{\mathsf{K}}, k) : \sigma_{\mathsf{K}})$
$r_{\mathsf{D}} \leftarrow \mathsf{SE.CExt}^{\mathsf{P}}(1^\lambda, k, c_{\mathsf{D}})$
Return $r_{\mathsf{K}} \| r_{\mathsf{D}}$
```

```
$\mathrm{Enc}(u, m)$
$(c_{\mathsf{K}}, \cdot) \leftarrow_{\$} \mathsf{S_K.Encaps}^{\mathrm{PPrim}}(1^\lambda, u : \sigma_{\mathsf{K}})$
$i_u \leftarrow i_u + 1$; $I_u.\mathsf{add}(c_{\mathsf{K}}, i_u)$
$\boxed{c_{\mathsf{D}} \leftarrow_{\$} \mathsf{SE.Enc}^{\mathsf{P}}(1^\lambda, k_{(u,i_u)}, m)}$
$c_{\mathsf{D}} \leftarrow_{\$} \mathsf{S_D.Enc}^{\mathrm{PPrim}}(1^\lambda, (u, i_u), |m| : \sigma_{\mathsf{D}})$
$M_{(u,i_u)}.\mathsf{add}(c_{\mathsf{D}}, m)$
$C_u.\mathsf{add}(c_{\mathsf{K}}, c_{\mathsf{D}})$
Return $(c_{\mathsf{K}}, c_{\mathsf{D}})$

$\mathrm{Dec}(u, (c_{\mathsf{K}}, c_{\mathsf{D}}))$
If $I_u\langle c_{\mathsf{K}} \rangle \neq \bot$ then
    $i \leftarrow I_u\langle c_{\mathsf{K}} \rangle$
    $\boxed{m \leftarrow \mathsf{SE.Dec}^{\mathsf{P}}(1^\lambda, k_{(u,i)}, c_{\mathsf{D}})}$
    $m \leftarrow_{\$} \mathsf{S_D.Dec}^{\mathrm{PPrim}}(1^\lambda, (u,i), c_{\mathsf{D}} : \sigma_{\mathsf{D}})$
Else
    $k \leftarrow_{\$} \mathsf{S_K.Decaps}^{\mathrm{PPrim}}(1^\lambda, u, c_{\mathsf{K}} : \sigma_{\mathsf{K}})$
    If $k = \bot$ then return $\bot$
    $m \leftarrow \mathsf{SE.Dec}^{\mathsf{P}}(1^\lambda, k, c_{\mathsf{D}})$
Return $m$
```

```
Adversary $\mathcal{A}_{\mathsf{D}}^{\mathrm{Enc,Dec,Exp,PPrim}}(1^\lambda)$
$i_{(\cdot)} \leftarrow 0$
$\sigma_{\mathsf{K}} \leftarrow_{\$} \mathsf{S_K.Init}(1^\lambda)$
$b' \leftarrow_{\$} \mathcal{A}_{\mathsf{cca}}^{\mathrm{EkSim,EncSim,DecSim,SExpSim,RExpSim,PPrim}}(1^\lambda)$
Return $(b' = 1)$

$\mathrm{EkSim}(u)$
$ek' \leftarrow_{\$} \mathsf{S_K.Ek}^{\mathrm{PPrim}}(1^\lambda, u : \sigma_{\mathsf{K}})$
Return $ek'$

$\mathrm{RExp}(u)$
For $(c_{\mathsf{K}}, i) \in I_u$ do
    $k \leftarrow \mathrm{Exp}((u,i))$
    $K_u.\mathsf{add}(c_{\mathsf{K}}, k)$
$dk' \leftarrow_{\$} \mathsf{S_K.RExp}^{\mathrm{PPrim}}(1^\lambda, u, K_u : \sigma_{\mathsf{K}})$
$K_u \leftarrow [\cdot]$; Return $dk'$

$\mathrm{SExp}(u, i)$
$(c_{\mathsf{K}}, c_{\mathsf{D}}) \leftarrow C_u[i]$
$k \leftarrow \mathrm{Exp}((u,i))$
$r_{\mathsf{K}} \leftarrow_{\$} \mathsf{S_K.SExp}^{\mathrm{PPrim}}(1^\lambda, u, i, (c_{\mathsf{K}}, k) : \sigma_{\mathsf{K}})$
$r_{\mathsf{D}} \leftarrow \mathsf{SE.CExt}^{\mathsf{P}}(1^\lambda, k, c_{\mathsf{D}})$
Return $r_{\mathsf{K}} \| r_{\mathsf{D}}$
```

```
$\mathrm{EncSim}(u, m)$
$(c_{\mathsf{K}}, \cdot) \leftarrow_{\$} \mathsf{S_K.Encaps}^{\mathrm{PPrim}}(1^\lambda, u : \sigma_{\mathsf{K}})$
$i_u \leftarrow i_u + 1$; $I_u.\mathsf{add}(c_{\mathsf{K}}, i_u)$
$c_{\mathsf{D}} \leftarrow_{\$} \mathrm{Enc}((u, i_u), m)$
$C_u.\mathsf{add}(c_{\mathsf{K}}, c_{\mathsf{D}})$
Return $(c_{\mathsf{K}}, c_{\mathsf{D}})$

$\mathrm{DecSim}(u, (c_{\mathsf{K}}, c_{\mathsf{D}}))$
If $I_u\langle c_{\mathsf{K}} \rangle \neq \bot$ then
    $i \leftarrow I_u\langle c_{\mathsf{K}} \rangle$
    $m \leftarrow \mathrm{Dec}((u,i), c_{\mathsf{D}})$
Else
    $k \leftarrow_{\$} \mathsf{S_K.Decaps}^{\mathrm{PPrim}}(1^\lambda, u, c_{\mathsf{K}} : \sigma_{\mathsf{K}})$
    If $k = \bot$ then return $\bot$
    $m \leftarrow \mathsf{SE.Dec}^{\mathsf{P}}(1^\lambda, k, c_{\mathsf{D}})$
Return $m$
```

Figure 8: Last two hybrids and corresponding reduction in security proof for the KEM/DEM construction. Oracle PPrim is omitted as it is unchanged from the SIM*-AC-CPA/CCA security game. Giving P as an oracle means access to $\mathrm{PPrim}(\mathsf{Ls}, \cdot, \cdot, \varepsilon)$.

```
$\underline{\mathsf{S.Init}(1^\lambda)}$
$i_{(\cdot)} \leftarrow 0$
$\sigma_\mathsf{K} \leftarrow\!\!\$\ \mathsf{S_K.Init}(1^\lambda)$
$\sigma_\mathsf{D} \leftarrow\!\!\$\ \mathsf{S_D.Init}(1^\lambda)$
Return $(i_{(\cdot)}, I_{(\cdot)}, \sigma_\mathsf{K}, \sigma_\mathsf{D})$

$\underline{\mathsf{S.Ek}^{\mathrm{PPrim}}(1^\lambda, u : (i_{(\cdot)}, I_{(\cdot)}, \sigma_\mathsf{K}, \sigma_\mathsf{D}))}$
$ek' \leftarrow\!\!\$\ \mathsf{S_K.Ek}^{\mathrm{PPrim}}(1^\lambda, u : \sigma_\mathsf{K})$
Return $ek'$

$\underline{\mathsf{S.RExp}^{\mathrm{PPrim}}(1^\lambda, u, M_u : (i_{(\cdot)}, I_{(\cdot)}, \sigma_\mathsf{K}, \sigma_\mathsf{D}))}$
For $i = 1, \ldots, |M_u|$ do
    $((c_\mathsf{K}, c_\mathsf{D}), m) \leftarrow M_u[i]$
    $M_{(u,i)}.\mathsf{add}(c_\mathsf{D}, m)$
    $k \leftarrow\!\!\$\ \mathsf{S_D.Exp}^{\mathrm{PPrim}}(1^\lambda, (u,i), M_{(u,i)} : \sigma_\mathsf{D})$
    $K_u.\mathsf{add}(c_\mathsf{K}, k)$
$dk \leftarrow\!\!\$\ \mathsf{S_K.RExp}^{\mathrm{PPrim}}(1^\lambda, u, K_u : \sigma_\mathsf{K})$
Return $dk$

$\underline{\mathsf{S.SExp}^{\mathrm{PPrim}}(1^\lambda, u, i, M_u[i] : (i_{(\cdot)}, I_{(\cdot)}, \sigma_\mathsf{K}, \sigma_\mathsf{D}))}$
$((c_\mathsf{K}, c_\mathsf{D}), m) \leftarrow M_u[i]$; $M_{(u,i)}.\mathsf{add}(c_\mathsf{D}, m)$
$k \leftarrow\!\!\$\ \mathsf{S_D.Exp}^{\mathrm{PPrim}}(1^\lambda, (u,i), M_{(u,i)} : \sigma_\mathsf{D})$
$r_\mathsf{K} \leftarrow\!\!\$\ \mathsf{S_K.SExp}^{\mathrm{PPrim}}(1^\lambda, u, i, (c_\mathsf{K}, k) : \sigma_\mathsf{K})$
$r_\mathsf{D} \leftarrow \mathsf{SE.CExt}^{\mathsf{P}}(1^\lambda, k, c_\mathsf{D})$
Return $r_\mathsf{K} \| r_\mathsf{D}$
```

```
$\underline{\mathsf{S.Enc}^{\mathrm{PPrim}}(1^\lambda, u, \ell : (i_{(\cdot)}, I_{(\cdot)}, \sigma_\mathsf{K}, \sigma_\mathsf{D}))}$
$(c_\mathsf{K}, \cdot) \leftarrow\!\!\$\ \mathsf{S_K.Encaps}^{\mathrm{PPrim}}(1^\lambda, u : \sigma_\mathsf{K})$
$i_u \leftarrow i_u + 1$; $I_u.\mathsf{add}(c_\mathsf{K}, i_u)$
$c_\mathsf{D} \leftarrow\!\!\$\ \mathsf{S_D.Enc}^{\mathrm{PPrim}}(1^\lambda, (u, i_u), \ell : \sigma_\mathsf{D})$
Return $(c_\mathsf{K}, c_\mathsf{D})$

$\underline{\mathsf{S.Dec}^{\mathrm{PPrim}}(1^\lambda, u, (c_\mathsf{K}, c_\mathsf{D}) : (i_{(\cdot)}, I_{(\cdot)}, \sigma_\mathsf{K}, \sigma_\mathsf{D}))}$
If $I_u\langle c_\mathsf{K}\rangle \neq \bot$ then
    $i \leftarrow I_u\langle c_\mathsf{K}\rangle$
    $m \leftarrow\!\!\$\ \mathsf{S_D.Dec}^{\mathrm{PPrim}}(1^\lambda, (u,i), c_\mathsf{D} : \sigma_\mathsf{D})$
Else
    $k \leftarrow\!\!\$\ \mathsf{S_K.Decaps}^{\mathrm{PPrim}}(1^\lambda, u, c_\mathsf{K} : \sigma_\mathsf{K})$
    If $k = \bot$ then return $\bot$
    $m \leftarrow \mathsf{SE.Dec}^{\mathsf{P}}(1^\lambda, k, c_\mathsf{D})$
Return $m$
```

Putting together the equalities we have shown gives that $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cpa}}_{\mathsf{KD,S,P},\mathcal{A}_{\mathsf{cca}}}(\lambda) = \mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{KEM,S_K,P},\mathcal{A}_\mathsf{K}}(\lambda) + \mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{SE,S_D,P},\mathcal{A}_\mathsf{D}}(\lambda)$. Note that $\mathcal{A}_\mathsf{D}$ and S depends on $\mathsf{S_K}$, and that S depends on $\mathsf{S_D}$. This limited chain of dependencies (as well as the efficiency of all algorithms we provided), then gives the result for both the "weak" and "strong" quantifications.

C Hashed KEM and Fujisaki-Okamoto Transform Proofs

In this section, we provide the deferred proofs of security for Hashed KEM (Theorem 5.2, proof in C.3) and the Fujisaki-Okamoto Transform (Theorem 5.3, proof in C.4). Before starting those we give a useful lemma for formalizing such proofs (C.1) and actually formally define the one-wayness definition we use (C.2).

C.1 Deferred Programming Lemma

We introduce a lemma that we found useful for analysis of some "low-level" SIM*-AC proofs more precise. This lemma captures how difficult it is to distinguish between honestly querying a random oracle versus sampling a purported output from the query at random and then later attempting to program the random oracle to be consistent with output.

As an example, consider running one of the two following sequences of code.

- $\sigma_\mathsf{P} \leftarrow [\cdot,\cdot]$; $(k, x, \sigma_\mathcal{G}) \leftarrow\!\!\$\ \mathcal{G}_1^{\mathsf{P_{rom}}}$; $y \leftarrow \mathsf{P_{rom}.Ls}(1^\lambda, k, x : \sigma_\mathsf{P})$; $\mathcal{G}_2^{\mathsf{P_{rom}}}(y : \sigma_\mathcal{G})$; $\mathcal{G}_3^{\mathsf{P_{rom}}}(\sigma_\mathcal{G})$
- $\sigma_\mathsf{P} \leftarrow [\cdot,\cdot]$; $(k, x, \sigma_\mathcal{G}) \leftarrow\!\!\$\ \mathcal{G}_1^{\mathsf{P_{rom}}}$; $y \leftarrow\!\!\$\ \mathsf{P}.\mathcal{R}_\lambda$; $\mathcal{G}_2^{\mathsf{P_{rom}}}(y : \sigma_\mathcal{G})$; $\mathsf{P_{rom}.Prog}(1^\lambda, k, x, y : \sigma_\mathsf{P})$; $\mathcal{G}_3^{\mathsf{P_{rom}}}(\sigma_\mathcal{G})$

The view of $\mathcal{G}$ can only differ between these sequence if $\mathcal{G}_1$ or $\mathcal{G}_2$ queries $\mathsf{P_{rom}}$ on (k, x). We will generalize this to consider multiple instances of deferred programming.

```
Game $\mathrm{G}^{\textsf{def-prog}}_{\mathsf{P_{rom}},\mathcal{G}}(\lambda)$
    $\sigma_{\mathsf{P}} \leftarrow_{\$} \mathsf{P_{rom}}.\mathsf{Init}(1^\lambda)$
    $b \leftarrow_{\$} \{0,1\}$
    $b' \leftarrow_{\$} \mathcal{G}^{\text{QUERY},\text{DEFPROG},\text{PPRIM}}(1^\lambda)$
    $\mathsf{bad} \leftarrow \mathsf{bad}_0 \vee \mathsf{bad}_1 \vee \mathsf{bad}_2$
    Return $(b = b')$

PPRIM$(\mathsf{Op}, k, x, y)$
    Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
    If $Y\langle k, x\rangle \neq \bot$ then $\mathsf{bad}_2 \leftarrow \mathsf{true}$
    $y \leftarrow_{\$} \mathsf{P_{rom}}.\mathsf{Op}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})$
    Return $y$

QUERY$(k, x)$
    If $Y\langle k, x\rangle \neq \bot$ then $\mathsf{bad}_0 \leftarrow \mathsf{true}$
    If $\sigma_{\mathsf{P}}[k, x] \neq \bot$ then $\mathsf{bad}_1 \leftarrow \mathsf{true}$
    If $b = 1$ then $y \leftarrow_{\$} \mathsf{P_{rom}}.\mathsf{Ls}(1^\lambda, k, x : \sigma_{\mathsf{P}})$
    If $b = 0$ then $y \leftarrow_{\$} \mathsf{P_{rom}}.\mathcal{R}_\lambda$
    $Y.\mathsf{add}((k, x), y)$
    Return $y$

DEFPROG$(t)$
    Require $Y[t] \neq \bot$
    $((k, x), y) \leftarrow Y[t]$
    $Y[t] \leftarrow \bot$
    If $b = 0$ then $\mathsf{P_{rom}}.\mathsf{Prog}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})$
    Return $\diamond$
```

Figure 9: Game for analyzing deferred programming of a random oracle

Consider the game $\mathrm{G}^{\textsf{def-prog}}_{\mathsf{P_{rom}},\mathcal{G}}$ shown in Fig. 9. In it, the adversary $\mathcal{G}$ interacts with $\mathsf{P_{rom}}$ through three oracles. The PPRIM oracle is the same as the similarly named oracle we have seen before. The deferred programming happens between the QUERY and DEFPROG oracles. In the real world ($b = 1$), the oracle QUERY(k, x) evaluates the random oracle on (k, x) and returns the result y. The DEFPROG oracle does nothing. In the ideal world ($b = 0$), the result y is picked uniformly at random. This value is not shared with $\mathsf{P_{rom}}$, so a PPRIM$(\mathsf{Ls}, k, x, \varepsilon)$ query would not return y. On input t, the oracle DEFPROG tries to patch this up by programming $\mathsf{P_{rom}}$ with the (k, x, y) values from the t-th query to QUERY.

We define the advantage of $\mathcal{G}$ by $\mathsf{Adv}^{\textsf{def-prog}}_{\mathsf{P_{rom}},\mathcal{G}}(\lambda) = 2\Pr[\mathrm{G}^{\textsf{def-prog}}_{\mathsf{P_{rom}},\mathcal{G}}(\lambda)] - 1$. It is easy to define a $\mathcal{G}$ which has advantage 1 (e.g., make the same query to QUERY twice in a row). When we use deferred programming, we will restrict our attention to restricted classes of $\mathcal{G}$ (typically of the form that $\mathcal{G}$ is emulating some other game to an adversary $\mathcal{A}$ that it runs internally) and show that this advantage is small for $\mathcal{G}$ in that class.

The game sets three bad flag when queries are made which will allow the adversary to detect the deferred programming. This is formalized in the following result.

Lemma C.1 (Deferred Programming Lemma) *For all $\mathcal{G}$, it holds that*

$$\mathsf{Adv}^{\textsf{def-prog}}_{\mathsf{P_{rom}},\mathcal{G}}(\lambda) \leq \Pr[\mathrm{G}^{\textsf{def-prog}}_{\mathsf{P_{rom}},\mathcal{G},0}(\lambda) \text{ sets } \mathsf{bad}] = \Pr[\mathrm{G}^{\textsf{def-prog}}_{\mathsf{P_{rom}},\mathcal{G},1}(\lambda) \text{ sets } \mathsf{bad}].$$

To understand why this is the case, let's think through the different bad flags that make up bad. The flags bad_1 and bad_2 together capture the case that a query PPRIM$(\cdot, k, x, \cdot)$ is made for some (k, x) that was queried to QUERY, before the corresponding query to DEFPROG was made. The two flags differ in whether the PPRIM query was made before (bad_1) or after (bad_2) the QUERY query. The flags bad_0 and bad_1 capture the case that two queries to QUERY are make with the same (k, x). The two flags differ in whether the DEFPROG query corresponding to the first QUERY query was made before (bad_1) or after (bad_0) the second QUERY query.

This argument could have been embedded in our later proofs without requiring this separate formalization. We found this modularization convenient so that we can extract analysis of the detectability of deferred programming from the complicated games in which we will need it.

C.2 One-way Security Definitions for KEMs

We define one-wayness of key encapsulation scheme KEM by the game given in Fig. 10. The adversary can query ENCAPS to get challenge ciphertexts under an unknown decapsulation key. The adversary can also learn the decapsulation key by querying REXP or the underlying randomness used by querying SEXP. The adversary wins if they guess a key associated with a challenge ciphertext for an unexposed user. We define the advantage function $\mathsf{Adv}^{\mathsf{ow}^*}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) = \Pr[\mathrm{G}^{\mathsf{ow}^*}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda)]$. We say KEM is OW* secure with P if for all PPT $\mathcal{A}_{\mathsf{ow}}$, the advantage $\mathsf{Adv}^{\mathsf{ow}^*}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\cdot)$ is negligible. Our results using one-wayness work whether P is programmable or non-programmable.

```
Game $\mathrm{G}^{\mathsf{ow}^*}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda)$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow\!\!\$\ \mathsf{KEM.Kg}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow\!\!\$\ \mathsf{P.Init}(1^\lambda)$
$\mathcal{A}_{\mathsf{ow}}^{\textsc{Ek},\textsc{Encaps},\textsc{SExp},\textsc{RExp},\textsc{Guess},\textsc{PPrim}}(1^\lambda)$
Return $\mathsf{win}$

$\textsc{PPrim}(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$y \leftarrow\!\!\$\ \mathsf{P.Op}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})$
Return $y$

$\textsc{Ek}(u)$
Return $ek_u$

$\textsc{SExp}(u, i)$
$X_{u,i} \leftarrow \mathsf{true}$
Return $(R_u[i], T_u[i])$

$\textsc{RExp}(u)$
$X_u \leftarrow \mathsf{true}$
Return $(dk_u, T_u)$

$\textsc{Encaps}(u)$
$r \leftarrow\!\!\$\ \mathsf{KEM.Rand}(\lambda)$
$(c, k) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek_u; r)$
$\mathsf{coll} \leftarrow \mathsf{coll} \vee (\neg X_u \wedge \exists u' : k \in T_{u'})$
$\mathsf{early} \leftarrow \mathsf{early} \vee (\neg X_u \wedge k \in K)$
$T_u.\mathsf{add}(k)$; $R_u.\mathsf{add}(r)$
Return $c$

$\textsc{Guess}(k)$
$K.\mathsf{add}(k)$
$\mathsf{correct} \leftarrow \exists u, i : \neg X_u \wedge \neg X_{u,i} \wedge (k = T_u[i])$
$\mathsf{win} \leftarrow \mathsf{win} \vee \mathsf{correct}$
```

Figure 10: Game defining one-wayness of KEM

Stronger variants. To capture stronger variants of one-wayness, we add additional oracles to the above game. We consider a plaintext-checking oracle PC and a decryption-consistency oracle DC defined by the following pseudocode.

```
$\textsc{Pc}(u, \mathbf{K}, c)$
Require $\bot \notin \mathbf{K}$
$k \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c)$
If $k \in \mathbf{K}$ then return $k$
Return $\bot$

$\textsc{Dc}(u, \mathbf{V}, c)$
Require $u \notin \mathbf{V}$
$k \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c)$
For $v \in \mathbf{V}$ do
    $k' \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_v, c)$
    If $k = k' \neq \bot$ then return $(v, k)$
Return $\bot$
```

The plaintext-checking oracle takes a set of keys $\mathbf{K}$ and a ciphertext c for a user u and tells which, if any, of the given keys is encrypted by that ciphertext. The require statement emphasizes that the oracle cannot be used for simply checking if the ciphertext is valid for u. A more common definition of the oracle only allows checking one key at a time; this version can be emulated by calling such an oracle $|\mathbf{K}|$ times. We denote by $\mathrm{G}^{\mathsf{ow}^*\text{-}\mathsf{pca}}$ the game obtained by adding PC and define $\mathsf{Adv}^{\mathsf{ow}^*\text{-}\mathsf{pca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) = \Pr[\mathrm{G}^{\mathsf{ow}^*\text{-}\mathsf{pca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda)]$. We say KEM is OW*-PCA secure with P if for all PPT $\mathcal{A}_{\mathsf{ow}}$, the advantage $\mathsf{Adv}^{\mathsf{ow}^*\text{-}\mathsf{pca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\cdot)$ is negligible.

The decryption consistency oracle takes a user u, ciphertext c, and set of other users $\mathbf{V}$. It checks if any of these other users decrypt the ciphertext to the same (non-$\bot$) key as u. We denote by $\mathrm{G}^{\mathsf{ow}^*\text{-}\mathsf{pcdca}}$ the game obtained by adding both PC and DC and define $\mathsf{Adv}^{\mathsf{ow}^*\text{-}\mathsf{pcdca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) = \Pr[\mathrm{G}^{\mathsf{ow}^*\text{-}\mathsf{pcdca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda)]$. We say KEM is OW*-PCDCA secure with P if for all PPT $\mathcal{A}_{\mathsf{ow}}$, the advantage $\mathsf{Adv}^{\mathsf{ow}^*\text{-}\mathsf{pcdca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\cdot)$ is negligible.

Single query security. Our one-wayness definitions allow for multiple users, challenges, and guesses. They allow for the exposure of receiver keys and sender randomness. All of these definitional choices were made to simplify the security proofs that use one-wayness. However, we emphasize that a standard argument shows that this is implied by a version with a single user, single challenge, a single guess, and no exposures. Given an adversary $\mathcal{A}_{\mathsf{ow}}$, one can build an adversary $\mathcal{A}_1$ that guesses which user, encapsulation query, and guess query will result in $\mathcal{A}_{\mathsf{ow}}$ winning. The corresponding queries are forwarded to $\mathcal{A}_1$'s oracles while $\mathcal{A}_1$ itself locally simulates all other users and queries. If $\mathcal{A}_{\mathsf{ow}}$ would expose one of the guessed queries, $\mathcal{A}_1$ can halt early.

In the case of OW*-PCDCA, $\mathcal{A}_1$ will need to use the PC oracle to simulate the DC oracle. It can perform the decapsulations in DC for all of the users it internally simulates. Then a PC query is used to compare a key obtained in this manner to one that should be obtained by decapsulating the ciphertext with the user that $\mathcal{A}_1$ is trying to attack. As consequence of this, we can also conclude that OW*-PCDCA security is implied by OW*-PCA security.[20]

Related quantities (key collisions and early guessing). There are two other related quantities we will want to measure about an adversary $\mathcal{A}$ playing the one-wayness games. In the games, coll is set in the event that the same key k is sampled in two different queries to ENCAPS (where the user in the second query is unexposed at the time of the query) and early is set in the event that ENCAPS for an unexposed user samples a key k that has already been queried to GUESS by the attacker. For a given scheme KEM expecting ideal primitive P and security parameter λ, we define the key min-entropy of KEM, denoted $\mathrm{H}_{\infty}^{\mathsf{KEM}}(\lambda)$, to be the largest real value satisfying

$$\Pr[k = k^* : (\cdot, k) \leftarrow\$ \mathsf{KEM.Encaps}^f(1^\lambda, ek)] \leq 2^{-\mathrm{H}_{\infty}^{\mathsf{KEM}}(\lambda)}$$

for all k^*, ek, and $f \in \mathsf{P}.\mathcal{P}_\lambda$. Then we can bound the probability of either event using key min-entropy or one-wayness itself.

Lemma C.2 *Let* X $\in$ *{OW*, OW*-PCA, OW*-PCDCA} and $\mathcal{A}$ be a PPT attacker against the* X *security of* KEM. *If* $\mathrm{H}_{\infty}^{\mathsf{KEM}}$ *is super-logarithmic or* KEM *is* X *secure with* P, *then the probability* $\Pr[\mathrm{G}^{\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}}(\cdot)$ sets coll or early$]$ *is negligible.*

Proof: Let q_{ENCAPS} denote the number of encapsulation queries $\mathcal{A}$ makes and q_{GUESS} denote the number of guess queries it makes (both implicitly at most polynomial in λ). Union bounds give

$$\Pr[\mathrm{G}^{\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}}(\lambda) \text{ sets } \mathsf{coll}] \leq \binom{q_{\text{ENCAPS}}}{2} \cdot 2^{-\mathrm{H}_{\infty}^{\mathsf{KEM}}(\lambda)} \text{ and}$$

$$\Pr[\mathrm{G}^{\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}}(\lambda) \text{ sets } \mathsf{early}] \leq q_{\text{ENCAPS}} q_{\text{GUESS}} \cdot 2^{-\mathrm{H}_{\infty}^{\mathsf{KEM}}(\lambda)}.$$

Let $\mathcal{A}'$ pick a random number t between 1 and q_{ENCAPS}, then run $\mathcal{A}$ up until immediately after the t-th encapsulation query. Then $\mathcal{A}'$ re-queries GUESS on all values of k previously queried to GUESS. Then for each encapsulation query that happened before the t-th, $\mathcal{A}'$ calls SEXP and queries the key it returns to GUESS. The adversary $\mathcal{A}'$ will succeed at the one-wayness game if either event was triggered by the t-th encapsulation query of $\mathcal{A}$. Hence we have that

$$\Pr[\mathrm{G}^{\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}}(\lambda) \text{ sets } \mathsf{coll} \text{ or } \mathsf{early}] \leq q_{\text{ENCAPS}} \cdot \mathsf{Adv}^{\mathsf{ow}^*}_{\mathsf{KEM},\mathsf{P},\mathcal{A}'}(\lambda),$$

completing the proof. ▮

[20]The DC oracle only returns anything meaningful if the attacker can produce a ciphertext that is decrypted identically by two different users. Proving that this is hard for a particular KEM can give a tighter connection between OW*-PCDCA and OW*-PCA security without going through single-user security. Consider, e.g., modifying a given KEM to replace k with $ek_u \,\|\, k$.

Related quantity (ciphertext early guessing). For one proof it will also be useful to argue that an attacker cannot guess a *ciphertext* before it is output by encapsulation. We will show that one-wayness implies this must be hard. Define $\mathrm{G}^{\mathsf{cguess}^*\text{-}\mathsf{x}}$ to be identical to $\mathrm{G}^{\mathsf{ow}^*\text{-}\mathsf{x}}$ except for the following changes:

- Oracle Guess is replaced with CGuess which on input c performs $C.\mathsf{add}(c)$.
- Oracle Encaps also computes $\mathsf{cguess} \leftarrow \mathsf{cguess} \vee (\neg X_u \wedge c \in C)$.
- The game returns cguess rather than win.

We define $\mathsf{Adv}^{\mathsf{cguess}^*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}}(\lambda) = \Pr[\mathrm{G}^{\mathsf{cguess}^*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}}(\lambda)]$ and say KEM is CGUESS-X secure with P if for all PPT $\mathcal{A}$, the advantage $\mathsf{Adv}^{\mathsf{cguess}^*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}}(\cdot)$ is negligible.

We can use key min-entropy or one-wayness to imply CGUESS* security as follows.

Lemma C.3 *If* $\mathrm{H}^{\mathsf{KEM}}_{\infty}$ *is super-logarithmic or* KEM *is OW*-X secure with* P*, then* KEM *is CGUESS*-X secure.*

Proof: For this proof, we make use of perfect correctness of KEM.

Let $\mathcal{A}$ against CGUESS*-X be given. Let q_{Encaps} denote the number of encapsulation queries $\mathcal{A}$ makes and q_{CGuess} denote the number of CGuess queries it makes (both implicitly at most polynomial in λ). If two ciphertexts are equal, then their decapsulations under a fixed decapsulation key will be equal so a union bound gives

$$\mathsf{Adv}^{\mathsf{cguess}^*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}}(\lambda) \leq q_{\text{Encaps}} q_{\text{CGuess}} \cdot 2^{-\mathrm{H}^{\mathsf{KEM}}_{\infty}(\lambda)}.$$

By picking random queries to Encaps and CGuess, we can build an adversary $\mathcal{A}'$ which makes at most one query to each and for which $\mathsf{Adv}^{\mathsf{cguess}^*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}}(\lambda) \leq q_{\text{Encaps}} q_{\text{CGuess}} \cdot \mathsf{Adv}^{\mathsf{cguess}^*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}'}(\lambda)$. Assume, without loss of generality, that the Encaps query is made for an unexposed user and after the CGuess query, then $\mathcal{A}'$ halts.

Next we build a related one-wayness adversary $\mathcal{A}_{\mathsf{ow}}$. It will run $\mathcal{A}'$ until it halts, forwarding all queries. After that $\mathcal{A}_{\mathsf{ow}}$ will query $(r, k) \leftarrow \text{SExp}(u, 1)$, make a second $\text{Encaps}(u)$ query, then query $\text{Guess}(k)$. If both Encaps queries return the ciphertext c^* in C_u, then correctness ensures they had the same key and so $\mathcal{A}'$ would win.

For the analysis, think of P being fully defined before the final query of $\mathcal{A}'$. Let Φ denote a random variable denoting all the randomness used by $\mathcal{A}'$ or the game until before its Encaps query. Let c and c' denote the ciphertexts output by the two Encaps queries. Then

$$\mathsf{Adv}^{\mathsf{cguess}^*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}'}(\lambda) = \mathbf{E}_{\Phi}[\Pr[c = c^* \mid \Phi]] \text{ and}$$
$$\mathsf{Adv}^{\mathsf{ow}*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) \geq \mathbf{E}_{\Phi}[\Pr[c = c' = c^* \mid \Phi]].$$

Here $\mathbf{E}_{\Phi}$ denotes an expectation over a random choice of Φ. Once Φ is fixed, c^* is fixed and the only randomness remaining in the choice of c and c' is the randomness sampled by encapsulation so they are independent when conditioned on Φ. Hence $\Pr[c = c' = c^* \mid \Phi] = \Pr[c = c^* \mid \Phi] \cdot \Pr[c' = c^* \mid \Phi] = \Pr[c = c^* \mid \Phi]^2$. Then because squaring is a convex function, Jensen's Inequality gives $\mathsf{Adv}^{\mathsf{ow}*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) \geq \mathsf{Adv}^{\mathsf{cguess}^*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}'}(\lambda)^2$. Thus,

$$\mathsf{Adv}^{\mathsf{cguess}^*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}}(\lambda) \leq q_{\text{Encaps}} q_{\text{CGuess}} \cdot \sqrt{\mathsf{Adv}^{\mathsf{ow}*\text{-}\mathsf{x}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda)}.$$

completing the proof. ▌

C.3 Hashed KEM Proof (Theorem 5.2)

This section provides a proof of Theorem 5.2 from Section 5. It establishes the SIM*-AC-CPA security of the hashed KEM construction built from a OW* secure KEM when the hash function is modeled as a random oracle.

Proof: Let $\mathcal{A}_{\mathsf{cpa}}$ be an adversary against the SIM-AC-CPA security of HKEM. For notational convenience, we assume without loss of generality that it does not make any oracle queries for users that have already been exposed or make SEXP queries corresponding to ENCAPS queries that have not been made yet. We construct adversary $\mathcal{A}_{\mathsf{ow}}$ (in Fig. 11) and simulator S such that

$$\mathsf{Adv}^{\mathsf{sim\text{-}ac\text{-}cpa}}_{\mathsf{HKEM[KEM]},\mathsf{S},\mathsf{P}\times\mathsf{P_{rom}},\mathcal{A}_{\mathsf{cpa}}}(\lambda) \leq \mathsf{Adv}^{\mathsf{ow}^*}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) + \Pr[\mathrm{G}^{\mathsf{ow}^*}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) \text{ sets } \mathsf{coll} \text{ or } \mathsf{early}]\,.$$

It will be clear from examination that the new algorithms we introduce are PPT. As shown by Lemma C.2, the assumed OW* security of the KEM implies that the latter probability term is negligibile. Thus, the theorem follows by the OW* security of KEM.

Our simulator S is defined as follows. It honestly samples keys for KEM and uses them for encapsulation, remembering the randomness that it used. During exposures, when it learns the key k that is supposed to be the output of some earlier HKEM[KEM] encapsulation, it tries to program the random oracle to match.[21] In its code, we write P to mean oracle access to $\mathrm{PPRIM}(\mathsf{Ls},(1,\cdot),\cdot,\diamond)$.

```
$\underline{\mathsf{S.Init}(1^\lambda)}$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow\!\!\$\ \mathsf{KEM.Kg}(1^\lambda)$
Return $(ek_{(\cdot)}, dk_{(\cdot)}, R_{(\cdot)}, T_{(\cdot)})$

$\underline{\mathsf{S.Ek}^{\mathrm{PPRIM}}(1^\lambda, u : (ek_{(\cdot)}, dk_{(\cdot)}, R_{(\cdot)}, T_{(\cdot)}))}$
Return $ek_u$

$\underline{\mathsf{S.Encaps}^{\mathrm{PPRIM}}(1^\lambda, u : (ek_{(\cdot)}, dk_{(\cdot)}, R_{(\cdot)}, T_{(\cdot)}))}$
$r \leftarrow\!\!\$\ \mathsf{KEM.Rand}(\lambda)$
$(c, k_{\mathsf{KEM}}) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek_u; r)$
$R_u.\mathsf{add}(r)$; $T_u.\mathsf{add}(k_{\mathsf{KEM}})$
Return $(c, \diamond)$

$\underline{\mathsf{S.SExp}^{\mathrm{PPRIM}}(1^\lambda, u, i, M_u[i] : (ek_{(\cdot)}, dk_{(\cdot)}, R_{(\cdot)}, T_{(\cdot)}))}$
$(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow T_u[i]$
$\mathrm{PPRIM}(\mathsf{Prog}, (2, k_{\mathsf{KEM}}), \varepsilon, k)$
Return $R_u[i]$

$\underline{\mathsf{S.RExp}^{\mathrm{PPRIM}}(1^\lambda, u, M_u : (ek_{(\cdot)}, dk_{(\cdot)}, R_{(\cdot)}, T_{(\cdot)}))}$
For $i = 1, \ldots, |M_u|$ do
    $(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow T_u[i]$
    $\mathrm{PPRIM}(\mathsf{Prog}, (2, k_{\mathsf{KEM}}), \varepsilon, k)$
Return $dk_u$
```

To analyze this simulator, consider the game $\mathrm{G}_1(\lambda)$ defined in Fig. 11. It includes the boxed code, but not the highlighted code. It was obtained by plugging the code of S and HKEM[KEM] into $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cpa}}$. Then we simplified by combining places where the same code was run whether $b = 0$ or $b = 1$ (such as the code running KEM.Encaps inside of ENCAPS). We removed code tracking which users and queries have been exposed, using our assumption that $\mathcal{A}_{\mathsf{cpa}}$ does not make queries for exposed users. By is construction, we have $\Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cpa}}_{\mathsf{HKEM[KEM]},\mathsf{S},\mathsf{P}\times\mathsf{P_{rom}},\mathcal{A}_{\mathsf{cpa}}}(\lambda)] = \Pr[\mathrm{G}_1(\lambda)]$.

When $b = 1$, the key k is chosen with $\mathsf{P_{rom}}$ while when $b = 0$ it is chosen at random and then later programmed into $\mathsf{P_{rom}}$ on exposures. Naturally, we want to analyze this with our deferred programming lemma. The game G_2 starts this process by adding the table Y and corresponding bad flags to match the deferred programming game. It includes the highlighted code, but not the boxed code. Note that Y is set up to store the values that were previously in M_u and T_u. We use I to store the mapping from (u, i) values used to index into M and T to the counter value used to map into Y. The only difference occurs in that values are erased from Y and in later exposures

[21] Our simulator remembers the points at which the random oracle will be programmed, k_{KEM}, in a table T_u. It could have instead recovered k_{KEM} by decrypting the ciphertext under consideration. Correctness and query consistency of KEM would ensure this is equivalent.

```
Game $\boxed{\mathrm{G}_1(\lambda)}$, $\mathrm{G}_2(\lambda)$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow\!\!\$\ \mathsf{KEM.Kg}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow\!\!\$\ \mathsf{P.Init}(1^\lambda)$
$\sigma_{\mathsf{P_{rom}}} \leftarrow\!\!\$\ \mathsf{P_{rom}.Init}(1^\lambda)$
$b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow\!\!\$\ \mathcal{A}_{\mathsf{cpa}}^{\textsc{Ek},\textsc{Encaps},\textsc{SExp},\textsc{RExp},\textsc{PPrim}}(1^\lambda)$
$\mathsf{bad} \leftarrow \mathsf{bad}_0 \vee \mathsf{bad}_1 \vee \mathsf{bad}_2$
Return $(b = b')$

$\textsc{Ek}(u)$
Return $ek_u$

$\textsc{PPrim}(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$(d, k_{\mathsf{KEM}}) \leftarrow k$
If $d = 1$ then $y \leftarrow\!\!\$\ \mathsf{P.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P}})$
If $d = 2$ then
  If $Y\langle k_{\mathsf{KEM}}, x\rangle \neq \bot$ then $\mathsf{bad}_2 \leftarrow \mathsf{true}$
  $y \leftarrow\!\!\$\ \mathsf{P_{rom}.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P_{rom}}})$
Return $y$
```

```
$\textsc{Encaps}(u)$
$r \leftarrow\!\!\$\ \mathsf{KEM.Rand}(\lambda)$
$(c, k_{\mathsf{KEM}}) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek_u; r)$
If $Y\langle k_{\mathsf{KEM}}, \varepsilon\rangle \neq \bot$ then $\mathsf{bad}_0 \leftarrow \mathsf{true}$
If $\sigma_{\mathsf{P_{rom}}}[k_{\mathsf{KEM}}, \varepsilon] \neq \bot$ then $\mathsf{bad}_1 \leftarrow \mathsf{true}$
If $b = 1$ then $k \leftarrow\!\!\$\ \mathsf{P_{rom}.Ls}(1^\lambda, k_{\mathsf{KEM}}, \varepsilon : \sigma_{\mathsf{P_{rom}}})$
If $b = 0$ then $k \leftarrow\!\!\$\ \mathsf{P_{rom}}.\mathcal{R}_\lambda$
$Y.\mathsf{add}((k_{\mathsf{KEM}}, \varepsilon), k)$; $t \leftarrow t + 1$; $I.\mathsf{add}((u, |R_u| + 1), t)$
$\boxed{M_u.\mathsf{add}(c, k);\ T_u.\mathsf{add}(k_{\mathsf{KEM}});}$ $R_u.\mathsf{add}(r)$
Return $(c, k)$

$\textsc{SExp}(u, i)$
If $\boxed{\mathsf{true}}$ $Y[I\langle u, i\rangle] \neq \bot$ then
  $\boxed{(c, k) \leftarrow M_u[i];\ k_{\mathsf{KEM}} \leftarrow T_u[i]}$
  $(k_{\mathsf{KEM}}, \cdot, k) \leftarrow Y[I\langle u, i\rangle]$; $Y[I\langle u, i\rangle] \leftarrow \bot$
  If $b = 0$ then $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, \varepsilon, k : \sigma_{\mathsf{P_{rom}}})$
Return $R_u[i]$

$\textsc{RExp}(u)$
For $i = 1, \ldots, |R_u|$ do
  If $\boxed{\mathsf{true}}$ $Y[I\langle u, i\rangle] \neq \bot$ then
    $\boxed{(c, k) \leftarrow M_u[i];\ k_{\mathsf{KEM}} \leftarrow T_u[i]}$
    $(k_{\mathsf{KEM}}, \cdot, k) \leftarrow Y[I\langle u, i\rangle]$; $Y[I\langle u, i\rangle] \leftarrow \bot$
    If $b = 0$ then $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, \varepsilon, k : \sigma_{\mathsf{P_{rom}}})$
Return $dk_u$
```

```
Adversary $\mathcal{G}^{\textsc{Query},\textsc{DefProg},\textsc{PPrim}}(1^\lambda)$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow\!\!\$\ \mathsf{KEM.Kg}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow\!\!\$\ \mathsf{P.Init}(1^\lambda)$
$b' \leftarrow\!\!\$\ \mathcal{A}_{\mathsf{cpa}}^{\textsc{Ek},\textsc{Encaps},\textsc{SExp},\textsc{RExp},\textsc{PPrimSim}}(1^\lambda)$
Return $b'$

$\textsc{Ek}(u)$
Return $ek_u$

$\textsc{PPrimSim}(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$(d, k_{\mathsf{KEM}}) \leftarrow k$
If $d = 1$ then $y \leftarrow\!\!\$\ \mathsf{P.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P}})$
If $d = 2$ then $\boxed{y \leftarrow \textsc{PPrim}(\mathsf{Op}, k_{\mathsf{KEM}}, x, y)}$
Return $y$

$\textsc{Encaps}(u)$
$r \leftarrow\!\!\$\ \mathsf{KEM.Rand}(\lambda)$
$(c, k_{\mathsf{KEM}}) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek_u; r)$
$\boxed{k \leftarrow \textsc{Query}(k_{\mathsf{KEM}}, \varepsilon)}$
$t \leftarrow t + 1$; $I.\mathsf{add}((u, |R_u| + 1), t)$
$R_u.\mathsf{add}(r)$
Return $(c, k)$

$\textsc{SExp}(u, i)$
$\boxed{\textsc{DefProg}(I\langle u, i\rangle)}$; Return $R_u[i]$

$\textsc{RExp}(u)$
For $i = 1, \ldots, |R_u|$ do $\boxed{\textsc{DefProg}(I\langle u, i\rangle)}$
Return $dk_u$
```

```
Adversary $\mathcal{A}_{\mathsf{ow}}^{\textsc{Ek},\textsc{Encaps},\textsc{SExp},\textsc{RExp},\textsc{Guess},\textsc{PPrim}}$
$\sigma_{\mathsf{P_{rom}}} \leftarrow\!\!\$\ \mathsf{P_{rom}.Init}(1^\lambda)$
$b' \leftarrow\!\!\$\ \mathcal{A}_{\mathsf{cpa}}^{\textsc{Ek},\textsc{EncapsSim},\textsc{SExpSim},\textsc{RExpSim},\textsc{PPrimSim}}(1^\lambda)$
Return $\diamond$

$\textsc{PPrimSim}(\mathsf{Op}, k, x, y)$
$(d, k_{\mathsf{KEM}}) \leftarrow k$
If $d = 1$ then $y \leftarrow \textsc{PPrim}(\mathsf{Op}, k_{\mathsf{KEM}}, x, y)$
If $d = 2$ then
  $\textsc{Guess}(k_{\mathsf{KEM}})$
  $y \leftarrow\!\!\$\ \mathsf{P_{rom}.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P_{rom}}})$
Return $y$

$\textsc{EncapsSim}(u)$
$c \leftarrow \textsc{Encaps}(u)$; $k \leftarrow\!\!\$\ \mathsf{P_{rom}}.\mathcal{R}_\lambda$
$M_u.\mathsf{add}(c, k)$; Return $(c, k)$

$\textsc{SExpSim}(u, i)$
$(c, k) \leftarrow M_u[i]$; $(r, k_{\mathsf{KEM}}) \leftarrow \textsc{SExp}(u, i)$
$\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, \varepsilon, k : \sigma_{\mathsf{P_{rom}}})$
Return $r$

$\textsc{RExpSim}(u)$
$(dk_u, T_u) \leftarrow \textsc{RExp}(u)$
For $i = 1, \ldots, |M_u|$ do
  $(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow T_u[i]$
  $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, \varepsilon, k : \sigma_{\mathsf{P_{rom}}})$
Return $dk_u$
```

Figure 11: Games and adversaries used for proof of security for Hashed KEM

the programming is skipped places where Y has been erased. But then anyway the programming would have had no effect, so $\Pr[G_1(\lambda)] = \Pr[G_2(\lambda)]$.

The adversary $\mathcal{G}$ makes explicit the reduction to deferred programming. It simulates the game G_2 using its own oracle (as shown by the boxes) to simulate the deferred programming of the random oracle. This gives us that $\Pr[G_2(\lambda)] = \Pr[G^{\mathsf{def\text{-}prog}}_{\mathsf{P_{rom}},\mathcal{G}}(\lambda)]$. Then applying the Deferred Programming Lemma (Lemma C.1) with the calculations so far we get that $\mathsf{Adv}^{\mathsf{sim\text{-}ac\text{-}cpa}}_{\mathsf{HKEM[KEM],S,P\times P_{rom}},\mathcal{A}_{\mathsf{cpa}}}(\lambda) \leq \Pr[G^{\mathsf{def\text{-}prog}}_{\mathsf{P_{rom}},\mathcal{G},0}(\lambda)$ sets $\mathsf{bad}]$. This bad event perfectly matches the one shown explicitly in G_2.

Now finally, consider the adversary $\mathcal{A}_{\mathsf{ow}}$. It simulates the view of $\mathcal{A}$ in $G^{\mathsf{def\text{-}prog}}_{\mathsf{P_{rom}},\mathcal{G},0}$ (equivalently, in G_2 when $b = 0$). At the time of encapsulation, it does not know k_{KEM} so it uses M_u and T_u as in G_1, rather than Y. By the logic from before, this does not change the correctness of its simulation. Whenever $\mathcal{A}_{\mathsf{cpa}}$ makes a query to the random oracle, $\mathcal{A}_{\mathsf{ow}}$ guesses the key to its own oracle. We will argue that whenever bad would be set in $G_2(\lambda)$, one of win, coll, or early would be set in $G^{\mathsf{ow}^*}_{\mathsf{KEM,P},\mathcal{A}_{\mathsf{ow}}}$. This implies $\Pr[G^{\mathsf{def\text{-}prog}}_{\mathsf{P_{rom}},\mathcal{G},0}(\lambda)$ sets $\mathsf{bad}] \leq \mathsf{Adv}^{\mathsf{ow}^*}_{\mathsf{KEM,P},\mathcal{A}_{\mathsf{ow}}}(\lambda) + \Pr[G^{\mathsf{ow}^*}_{\mathsf{KEM,P},\mathcal{A}_{\mathsf{ow}}}(\lambda)$ sets coll or $\mathsf{early}]$, completing the proof.

Suppose $G_2(\lambda)$ would set bad_0. Then the current k_{KEM} value must have also been sampled in an earlier encapsulation query, so coll would be set in G^{ow^*}. If $G_2(\lambda)$ would set bad_1, then one of two cases holds. If $\sigma_{\mathsf{P_{rom}}}[k_{\mathsf{KEM}}, \varepsilon]$ was set in an earlier exposure query, then the current k_{KEM} value must have also been sampled in an earlier encapsulation query, so coll would be set in G^{ow^*}. If $\sigma_{\mathsf{P_{rom}}}[k_{\mathsf{KEM}}, \varepsilon]$ was set in an earlier PPrim query, then $\mathcal{A}_{\mathsf{ow}}$ would have guessed k_{KEM} so early would be set in G^{ow^*}. For both coll and early we use the assumption that queries are only made for unexposed users so $\neg X_u$ must hold. Finally, if $G_2(\lambda)$ would set bad_2, then the attacker will guess this k_{KEM} which comes from an unexposed encapsulation query and so win would be set in G^{ow^*}.■

C.4 Fujisaki-Okamoto Style Transform Proof (Theorem 5.3)

Now we will prove the SIM*-AC-CCA security of $\mathsf{U}^{\not\perp}[\mathsf{KEM},\mathsf{F}]$ under the assumption that KEM is OW*-PCA secure and F is SIM*-AC-PRF secure. By our observations in Appendix C.2 we can use OW*-PCDCA and CGUESS*-PCDCA security as needed because they are implied by OW*-PCA security.

In our proof we start with a game transition based on SIM*-AC-PRF security. Then we complete the rest of the proof twice, giving both a single-user proof based on OW*-PCA and CGUESS*-PCA security and a direct multi-user proof based on OW*-PCDCA and CGUESS*-PCDCA security. Asymptotically, a SIM*-AC hybrid argument (similar to the one in Theorem 4.1) and the relationship between single- and multi-user one-wayness discussed in Appendix C.2 ensures that the two proofs are asymptotically equivalent. The multi-user variant likely provides better concrete bounds, but does so at the cost of a more complicated proof because the proof has to account for distinct users happening to query the random oracle on the same input.

Proof: Let $\mathcal{A}_{\mathsf{cca}}$ be a SIM*-AC-CCA adversary. To simplify notation, assume that it never makes oracle queries to an exposed u, never queries challenge ciphertexts to Decaps, and never repeats oracle queries to Decaps.

Simulator. We will start by presenting pseudocode for our simulator S which is given below. It honestly samples key for KEM and F, then uses them for honestly running the expected algorithms. The only difference from an honest execution is that it does not query k_{KEM} to the random oracle in encapsulation, but instead when an exposure occurs it reprograms the random oracle to consistently match k_{KEM} to k.[22]

In the code, it provides a P oracle to KEM with the meaning it appropriately forwards queries to the oracle to $\mathrm{PPRIM}(\mathsf{Ls}, (1, \cdot), \cdot, \diamond)$. Its query to $\mathsf{P}_{\mathsf{rom}}$ during decapsulation is shorthand for querying $\mathrm{PPRIM}(\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c, \diamond)$. For each of its algorithms we implicitly parse σ into $(ek_{(\cdot)}, dk_{(\cdot)}, fk_{(\cdot)}, \sigma_{\mathsf{F}}, R_{(\cdot)}, K_{(\cdot)})$.

```
$\underline{\mathsf{S.Init}(1^\lambda)}$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow\!\!\$\ \mathsf{KEM.Kg}(1^\lambda)$
$fk_{(\cdot)} \leftarrow\!\!\$\ \mathsf{F.Kg}(1^\lambda)$
Return $(ek_{(\cdot)}, dk_{(\cdot)}, fk_{(\cdot)}, \sigma_{\mathsf{F}}, R_{(\cdot)}, K_{(\cdot)})$

$\underline{\mathsf{S.Ek}^{\mathrm{PPRIM}}(1^\lambda, u : \sigma)}$
Return $ek_u$

$\underline{\mathsf{S.Encaps}^{\mathrm{PPRIM}}(1^\lambda, u : \sigma)}$
$r \leftarrow\!\!\$\ \mathsf{KEM.Rand}(\lambda)$
$(c, k_{\mathsf{KEM}}) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek_u; r)$
$R_u.\mathsf{add}(r)$; $K_u.\mathsf{add}(k_{\mathsf{KEM}})$
Return $(c, \diamond)$

$\underline{\mathsf{S.Decaps}^{\mathrm{PPRIM}}(1^\lambda, u, c : \sigma)}$
$k_{\mathsf{KEM}} \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then $k \leftarrow \mathsf{P}_{\mathsf{rom}}(k_{\mathsf{KEM}}, c)$
Else $k \leftarrow \mathsf{F.Ev}^{\mathsf{P}\times\mathsf{P}_{\mathsf{rom}}}(1^\lambda, fk_u, c)$
Return $k$

$\underline{\mathsf{S.SExp}^{\mathrm{PPRIM}}(1^\lambda, u, i, M_u[i] : \sigma)}$
$(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow K_u[i]$
$\mathrm{PPRIM}(\mathsf{Prog}, (2, k_{\mathsf{KEM}}), c, k)$
Return $R_u[i]$

$\underline{\mathsf{S.RExp}^{\mathrm{PPRIM}}(1^\lambda, u, M_u : \sigma)}$
For $i = 1, \ldots, |M_u|$ do
    $(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow K_u[i]$
    $\mathrm{PPRIM}(\mathsf{Prog}, (2, k_{\mathsf{KEM}}), c, k)$
Return $(dk_u, fk_u)$
```

Main ideas. We will use the Deferred Programming Lemma (Lemma C.1) to formalize the general idea that the only way to distinguish simulation from the real world is to query an honestly generated k_{KEM} to the random oracle before a corresponding exposure occurs. Intuitively then, we want to show this is implied by one-wayness by showing that a one-wayness adversary can simulate $\mathcal{A}_{\mathsf{cca}}$'s view and then use every key queried to the random oracle as a guess. The main challenge to this is that decapsulation uses a secret key which the one-wayness adversary does not know.

To allow simulation without knowing the decapsulation key, we first switch F's output to be random. Then we will respond to some decapsulation queries with randomness without knowing whether the given ciphertext decrypts at all. This could be detected if the attacker knows the encapsulated key and makes the corresponding query to the random oracle. To avoid this detection we use the plaintext-checking oracle on ciphertext-key pairs that are queried to the random oracle; if they match, we force consistency between decapsulation and the random oracle.

These general ideas have the same conceptual core as the proof for IND-CCA security of (PKE-based) $\mathsf{U}^{\not\perp}$ by HHK [HHK17, Thm. 3.4] and Hövelmanns [Höv21, Thm. 2.1.5]. Beyond the complexities introduced by allowing randomness and key exposures, there are also additionally subtle "bad" events we have to handle because our proof allows multiple users, multiple challenges, and other queries to be made before encapsulation queries (the cited proofs were single-user, single-challenge, and the challenge was sampled at the start of the game). These differences introduce additional challenges in ensuring consistency of the random oracle and detection of ways that a challenge (c, k_{KEM}) might be indirectly queried to the random oracle. For example, we introduced CGUESS* to modularly bound the probability of querying a ciphertext to decapsulation that later

[22]For exposed users it would make this random oracle query. Because the random oracle is consistent, it is important we not the query for unexposed users so we can program it later.

```
Games $\boxed{\mathrm{G}_0(\lambda)}$, $\mathrm{G}_1(\lambda)$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow\!\!\$\ \mathsf{KEM.Kg}(1^\lambda)$
$(\sigma_\mathsf{P}, \sigma_{\mathsf{P_{rom}}}) \leftarrow\!\!\$\ \mathsf{P} \times \mathsf{P_{rom}}.\mathsf{Init}(1^\lambda)$
$\boxed{fk_{(\cdot)} \leftarrow\!\!\$\ \mathsf{F.Kg}(1^\lambda)}$
$\sigma_\mathsf{F} \leftarrow\!\!\$\ \mathsf{S_F}.\mathsf{Init}(1^\lambda)$
$b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow\!\!\$\ \mathcal{A}_{\mathsf{cca}}^{\text{EK,ENCAPS,DECAPS,SEXP,REXP,PPRIM}}(1^\lambda)$
Return $(b = b')$

EK$(u)$
Return $ek_u$

PPRIM$(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$y \leftarrow\!\!\$\ \mathsf{P} \times \mathsf{P_{rom}}.\mathsf{Op}(1^\lambda, k, x, y : (\sigma_\mathsf{P}, \sigma_{\mathsf{P_{rom}}}))$
Return $y$

ENCAPS$(u)$
$r \leftarrow\!\!\$\ \mathsf{KEM.Rand}(\lambda)$
$(c, k_\mathsf{KEM}) \leftarrow \mathsf{KEM.Encaps}^\mathsf{P}(1^\lambda, ek_u; r)$
If $b = 1$ then $k \leftarrow \mathsf{P_{rom}}.\mathsf{Ls}(1^\lambda, k_\mathsf{KEM}, c : \sigma_{\mathsf{P_{rom}}})$
If $b = 0$ then $k \leftarrow\!\!\$\ \mathsf{P_{rom}}.\mathcal{R}_\lambda$
$M_u.\mathsf{add}(c, k)$; $R_u.\mathsf{add}(r)$; $K_u.\mathsf{add}(k_\mathsf{KEM})$
Return $(c, k)$

DECAPS$(u, c)$
$k_\mathsf{KEM} \leftarrow \mathsf{KEM.Decaps}^\mathsf{P}(1^\lambda, dk_u, c)$
If $k_\mathsf{KEM} \neq \bot$ then $k \leftarrow \mathsf{P_{rom}}.\mathsf{Ls}(1^\lambda, k_\mathsf{KEM}, c : \sigma_{\mathsf{P_{rom}}})$
$\boxed{\text{Else } k \leftarrow \mathsf{F.Ev}^{\mathsf{P}\times\mathsf{P_{rom}}}(1^\lambda, fk_u, c)}$
Else $k \leftarrow\!\!\$\ \mathsf{F.Out}(\lambda)$; $T_u[c] \leftarrow k$
Return $k$

SEXP$(u, i)$
If $b = 0$ then
    $(c, k) \leftarrow M_u[i]$; $k_\mathsf{KEM} \leftarrow K_u[i]$
    $\mathsf{P_{rom}}.\mathsf{Prog}(1^\lambda, k_\mathsf{KEM}, c, k : \sigma_{\mathsf{P_{rom}}})$
Return $R_u[i]$

REXP$(u)$
If $b = 0$ then
    For $i = 1, \ldots, |M_u|$ do
        $(c, k) \leftarrow M_u[i]$; $k_\mathsf{KEM} \leftarrow K_u[i]$
        $\mathsf{P_{rom}}.\mathsf{Prog}(1^\lambda, k_\mathsf{KEM}, c, k : \sigma_{\mathsf{P_{rom}}})$
$fk_u \leftarrow\!\!\$\ \mathsf{S_F}.\mathsf{Exp}^{\text{PPRIM}}(1^\lambda, u, T_u : \sigma_\mathsf{F})$
Return $(dk_u, fk_u)$
```

Figure 12: Games for PRF transition in the $\mathsf{U}^{\not\perp}$ security proof

becomes a challenge ciphertext, and we introduced the DC oracle to help detect different ways that random oracle queries of different users might overlap.

PRF game transition. Now consider the game G_0 shown in Fig. 12. It was obtained by plugging the code of $\mathsf{U}^{\not\perp}$ and S into $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}cca}}$. Then it was simplified by, e.g., moving code that appears in both branches of a conditional to instead occur outside of the conditional. Additionally, we made some simplifications based on our assumptions about the attacker. We assumed the attacker does not query encapsulation for exposed users and so always return a random key when $b = 0$. We also replaced S's query to PPRIM with direct access to $\mathsf{P_{rom}}.\mathsf{Prog}$. We removed the check for whether challenge ciphertexts were forwarded to DECAPS. As these modifications do not change the behavior of the game, we have that $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{U}^{\not\perp}[\mathsf{KEM},\mathsf{F}],\mathsf{S},\mathsf{P}\times\mathsf{P_{rom}},\mathcal{A}_\mathsf{cca}}(\lambda) = 2\Pr[\mathrm{G}_0(\lambda)] - 1$.

Now G_1 is identical to G_0 except the use of F inside of DECAPS is replaced with sampling a random value and fk is produced by $\mathsf{S_F}$ in REXP. Here we use the assumption that the attacker does not make decapsulation queries to exposed users or make repeated queries to avoid the need to check if the user is exposed or if the query is a repeat, in which case this would not be sampled at random.

It is straightforward to construct an efficient adversary $\mathcal{A}_\mathsf{prf}$ such that $\Pr[\mathrm{G}_0(\lambda)] - \Pr[\mathrm{G}_1(\lambda)] = \mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S_F},\mathsf{P}\times\mathsf{P_{rom}},\mathcal{A}_\mathsf{prf}}(\lambda)$. We describe it in terms of how we would modify the code of $\mathrm{G}_0/\mathrm{G}_1$ to obtain it. First, remove the boxed/grey code everywhere. Replace the return statement with returning 1 if $b = b'$ and 0 otherwise. In DECAPS, when $k_\mathsf{KEM} \neq \bot$ it queries its evaluation oracle to pick k in an else branch. Finally, in REXP it queries its own expose oracle to obtain fk. It queries PPRIM when P or $\mathsf{P_{rom}}$ are needed.

```
Games $\boxed{G_2(\lambda)}$, $G_3(\lambda)$
$(ek, dk) \leftarrow\$ \mathsf{KEM.Kg}(1^\lambda)$
$(\sigma_{\mathsf{P}}, \sigma_{\mathsf{P_{rom}}}) \leftarrow\$ \mathsf{P} \times \mathsf{P_{rom}.Init}(1^\lambda)$
$\sigma_{\mathsf{F}} \leftarrow\$ \mathsf{S_F.Init}(1^\lambda)$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow\$ \mathcal{A}_{\mathsf{cca}}^{\textsc{Ek},\textsc{Encaps},\textsc{Decaps},\textsc{SExp},\textsc{RExp},\textsc{PPrim}}(1^\lambda)$
Return $(b = b')$
```

```
$\textsc{Ek}()$
Return $ek$
```

```
$\textsc{PPrim}(\mathsf{Op}, (d, k_{\mathsf{KEM}}), x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
If $d = 1$ then $y \leftarrow\$ \mathsf{P.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P}})$
If $d = 2$ then
  If $D[x] \neq \bot$ and $\textsc{Pc}(\varepsilon, \{k_{\mathsf{KEM}}\}, x) \neq \bot$ then
    $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, x, D[x] : \sigma_{\mathsf{P_{rom}}})$
  $y \leftarrow\$ \mathsf{P_{rom}.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P_{rom}}})$
  $\mathbf{K}_x.\mathsf{add}(k_{\mathsf{KEM}})$
Return $y$
```

```
$\textsc{Encaps}()$
$r \leftarrow\$ \mathsf{KEM.Rand}(\lambda)$
$(c, k_{\mathsf{KEM}}) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek; r)$
If $D[c] \neq \bot$ then $\mathsf{bad}^* \leftarrow \mathsf{true}$
If $b = 1$ then $k \leftarrow \mathsf{P_{rom}.Ls}(1^\lambda k_{\mathsf{KEM}}, c : \sigma_{\mathsf{P_{rom}}})$
If $b = 0$ then $k \leftarrow\$ \mathsf{P_{rom}}.\mathcal{R}_\lambda$
$M.\mathsf{add}(c, k)$; $R.\mathsf{add}(r)$; $K.\mathsf{add}(k_{\mathsf{KEM}})$
Return $(c, k)$
```

```
$\textsc{Decaps}(c)$
$k_{\mathsf{KEM}} \leftarrow \textsc{Pc}(\varepsilon, \mathbf{K}_c, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
  Return $\textsc{PPrim}(\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c)$
$k_{\mathsf{KEM}} \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
  $\boxed{D[c] \leftarrow \mathsf{P_{rom}.Ls}(1^\lambda, k_{\mathsf{KEM}}, c : \sigma_{\mathsf{P_{rom}}})}$
  $D[c] \leftarrow\$ \mathsf{F.Out}(\lambda)$
Else $D[c] \leftarrow\$ \mathsf{F.Out}(\lambda)$
Return $D[c]$
```

```
$\textsc{SExp}(i)$
If $b = 0$ then
  $(c, k) \leftarrow M[i]$; $k_{\mathsf{KEM}} \leftarrow K[i]$
  $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, c, k : \sigma_{\mathsf{P_{rom}}})$
Return $R[i]$
```

```
$\textsc{RExp}()$
If $b = 0$ then
  For $i = 1, \ldots, |M|$ do
    $(c, k) \leftarrow M[i]$; $k_{\mathsf{KEM}} \leftarrow K[i]$
    $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, c, k : \sigma_{\mathsf{P_{rom}}})$
$T \leftarrow []$
For $c \in D$ if $\mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk, c) = \bot$ then
  $T[c] \leftarrow D[c]$
$fk \leftarrow\$ \mathsf{S_F.Exp}^{\textsc{PPrim}}(1^\lambda, \varepsilon, T : \sigma_{\mathsf{F}})$
Return $(dk, fk)$
```

Figure 13: Games for single-user $\mathsf{U}^{\not\perp}$ security proof

C.4.1 Single-user Proof

From here, we split into the single-use and multi-user (Section C.4.2) versions of the proof. So in this section, we assume that $\mathcal{A}_{\mathsf{cca}}$ only makes queries for a single user.

Transition to game G_2**.** We start with G_2, shown in Fig. 13. To simply notation, almost everywhere that the variable u was used, we removed the variable including as input to functions. Anywhere that a user variable is expected by functions or algorithms not defined within the figure, we use the empty string ε as the name.

We will claim that G_2 is equivalent to G_1 (restricted to a single user ε). Oracle $\textsc{Ek}$ and $\textsc{SExp}$ have not been changed. Oracle $\textsc{Encaps}$ has an additional if statement which merely sets a bad flag which is not used elsewhere. The main modifications were made to move us toward a game where decapsulation can be simulated without knowledge of the decapsulation key.

First, in $\textsc{PPrim}$, when $\mathsf{P_{rom}}$ would be called on a pair (k_{KEM}, c) we store k_{KEM} in $\mathbf{K}_c$. Now at the beginning of $\textsc{Decaps}(c)$, we use $\textsc{Pc}$ to check if any of the keys in $\mathbf{K}_c$ are consistent with c. If so, we use this key without directly running $\mathsf{KEM.Decaps}$. The code of $\textsc{Pc}$ is not explicitly defined in the figure; it is to be understood as being the same as the oracle defined in Section C.2. Here we called

PPrim rather than just calling $\mathsf{P}_{\mathsf{rom}}$ directly because we found it makes later arguments easier to write, but the proof should work either way. Oracle PPrim additionally added an if statement which does nothing because the highlighted code is not included in this game.

At the end of Decaps we stop using the table T to store outputs when $k_{\mathsf{KEM}} \neq \bot$ and instead use a table D to store outputs irrespective of whether k_{KEM} is $\bot$. Then in RExp we recover T by decapsulating each ciphertext in D to see if the corresponding k_{KEM} was $\bot$.

Recall that P is "detectable", so we have to be careful that we do not add or remove detectable queries when compared to G_1. The oracle Pc adds additional queries by running $\mathsf{KEM.Decaps}^{\mathsf{P}}$, but in Decaps decapsulation is anyway run on the same input in G_1. In PPrim, the oracle is only run when $D[x] \neq \bot$ in which case the same decapsulation was already run in Decaps. (We assume "If A and B" does not evaluate B if A evaluates to false.) We additionally add oracle queries by running decapsulation in RExp, but again this is only done for ciphertexts that are in D and hence were already queried to decapsulation.

By the above arguments, $\Pr[\mathrm{G}_1(\lambda)] = \Pr[\mathrm{G}_2(\lambda)]$.

Transition to game G_3**.** The decapsulation oracle in G_2 still needs to use dk to check whether $k_{\mathsf{KEM}} = \bot$ at the end. We want to instead simulate this by sampling the output at random in either case. In the case that the output should have come from $\mathsf{P}_{\mathsf{rom}}$ we will attempt to program the random output back into the random oracle should it ever be needed.

Consider game G_3 from Fig. 13. It is identical to G_2, except the use of $\mathsf{P}_{\mathsf{rom}}$ is replaced with random sampling in Decaps and inside PPrim if we are given inputs for which Decaps previously randomly sampled then we try to program $\mathsf{P}_{\mathsf{rom}}$ with this. Now both branches of the if statement at the end of Decaps are identical, so later in the proof we can simulate it without running decapsulation by always picking $D[c]$ at random. (Note that decapsulation is being run on the same inputs inside Pc, so we do not have issues with the detectability of P.)

To analyze the difference between games G_2 and G_3 we use arguments based on the Deferred Programming Lemma (Lemma C.1) to claim that $\Pr[\mathrm{G}_2(\lambda)] - \Pr[\mathrm{G}_3(\lambda)] \leq \Pr[\mathrm{G}_3(\lambda) \text{ sets } \mathsf{bad}^*]$. The bad flag gets set only if a ciphertext is sampled inside of Encaps that was previously queried to Decaps. We bound the probability of this by the CGUESS*-PCA security of KEM, getting $\Pr[\mathrm{G}_3(\lambda) \text{ sets } \mathsf{bad}^*] \leq \mathsf{Adv}^{\mathsf{cguess}^*\text{-}\mathsf{pca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{cguess}^*}}(\lambda)$.

The relevant adversaries are defined in Fig. 14. Let us start with $\mathcal{A}_{\mathsf{cguess}^*}$ as it is relatively straightforward. It runs $\mathcal{A}_{\mathsf{cca}}$, using its own oracles to simulate most uses of KEM and P. It simulates decapsulation without $\mathsf{KEM.Decaps}$ as discussed above. To simulate encapsulation it needs to know k_{KEM} when $b = 1$, which it obtains by a call to SExp (which does not affect setting cguess). It queries CGuess with each ciphertext added to D, so the claim follows because cguess will be set in $\mathrm{G}^{\mathsf{cguess}^*\text{-}\mathsf{pca}}$ whenever bad^* would be set in G_3. (Recall that by assumption Encaps will only be called for exposed users.)

Now consider the deferred programming adversary $\mathcal{G}_1$ from the same figure. It was obtained by copying the code of G_2 and G_3, but replacing all uses of $\mathsf{P}_{\mathsf{rom}}$ with queries to its oracles. In particular, the difference between the games in Decaps and PPrim are replaced with Query and DefProg respectively. All other uses are replaced with PPrim queries. If $\mathcal{A}_{\mathsf{cca}}$ outputs the correct bit it outputs 1, otherwise it outputs 0.

We naturally then have $\Pr[\mathrm{G}_2(\lambda)] = \Pr[\mathrm{G}^{\mathsf{def}\text{-}\mathsf{prog}}_{\mathsf{P}_{\mathsf{rom}},\mathcal{G}_1,1}(\lambda)]$ and $\Pr[\mathrm{G}_3(\lambda)] = \Pr[\mathrm{G}^{\mathsf{def}\text{-}\mathsf{prog}}_{\mathsf{P}_{\mathsf{rom}},\mathcal{G}_1,0}(\lambda)]$. Hence Lemma C.1 gives $\Pr[\mathrm{G}_2(\lambda)] - \Pr[\mathrm{G}_3(\lambda)] \leq \Pr[\mathrm{G}^{\mathsf{def}\text{-}\mathsf{prog}}_{\mathsf{P}_{\mathsf{rom}},\mathcal{G}_1,0}(\lambda) \text{ sets } \mathsf{bad}]$. So to justify our claim, we

```
Adversary $\mathcal{G}_1^{\text{Query},\text{DefProg},\text{PPrim}}(1^\lambda)$
$(ek, dk) \leftarrow\$ \mathsf{KEM.Kg}(1^\lambda)$
$\sigma_\mathsf{P} \leftarrow\$ \mathsf{P.Init}(1^\lambda)$; $\sigma_\mathsf{F} \leftarrow\$ \mathsf{S_F.Init}(1^\lambda)$
$b \leftarrow \{0,1\}$
$b' \leftarrow\$ \mathcal{A}_{\mathsf{cca}}^{\text{Ek},\text{Encaps},\text{Decaps},\text{SExp},\text{RExp},\text{PPrimSim}}(1^\lambda)$
Return $\diamond$

Ek(): Return $ek$

PPrimSim$(\mathsf{Op}, (d, k_\mathsf{KEM}), x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
If $d = 1$ then $y \leftarrow\$ \mathsf{P.Op}(1^\lambda, k_\mathsf{KEM}, x, y : \sigma_\mathsf{P})$
If $d = 2$ then
    If $D[x] \neq \perp$ and $\text{Pc}(\varepsilon, \{k_\mathsf{KEM}\}, x) \neq \perp$ then
        $\boxed{\text{DefProg}(I\langle k_\mathsf{KEM}, x\rangle)}$
    $\boxed{y \leftarrow \text{PPrim}(\mathsf{Op}, k_\mathsf{KEM}, x, y)}$
    $\mathbf{K}_x.\mathsf{add}(k_\mathsf{KEM})$
Return $y$

Encaps()
$r \leftarrow\$ \mathsf{KEM.Rand}(\lambda)$
$(c, k_\mathsf{KEM}) \leftarrow \mathsf{KEM.Encaps}^\mathsf{P}(1^\lambda, ek; r)$
If $D[c] \neq \perp$ then $\mathsf{bad}^* \leftarrow \mathsf{true}$
If $b = 1$ then $\boxed{k \leftarrow \text{PPrim}(\mathsf{Ls}, k_\mathsf{KEM}, c)}$
If $b = 0$ then $k \leftarrow\$ \mathsf{P_{rom}}.\mathcal{R}_\lambda$
$M.\mathsf{add}(c, k)$; $R.\mathsf{add}(r)$; $K.\mathsf{add}(k_\mathsf{KEM})$
Return $(c, k)$

Decaps$(c)$
$k_\mathsf{KEM} \leftarrow \text{Pc}(\varepsilon, \mathbf{K}_c, c)$
If $k_\mathsf{KEM} \neq \perp$ then
    Return PPrimSim$(\mathsf{Ls}, (2, k_\mathsf{KEM}), c)$
$k_\mathsf{KEM} \leftarrow \mathsf{KEM.Decaps}^\mathsf{P}(1^\lambda, dk, c)$
If $k_\mathsf{KEM} \neq \perp$ then
    $t \leftarrow t + 1$; $I.\mathsf{add}((k_\mathsf{KEM}, c), t)$
    $\boxed{D[c] \leftarrow \text{Query}(k_\mathsf{KEM}, c)}$
Else $D[c] \leftarrow\$ \mathsf{F.Out}(\lambda)$
Return $D[c]$

SExp$(i)$
If $b = 0$ then
    $(c, k) \leftarrow M[i]$; $k_\mathsf{KEM} \leftarrow K[i]$
    $\boxed{\text{PPrim}(\mathsf{Prog}, k_\mathsf{KEM}, c, k)}$
Return $R[i]$

RExp()
If $b = 0$ then
    For $i = 1, \ldots, |M|$ do
        $(c, k) \leftarrow M[i]$; $k_\mathsf{KEM} \leftarrow K[i]$
        $\boxed{\text{PPrim}(\mathsf{Prog}, k_\mathsf{KEM}, c, k)}$
$T \leftarrow []$
For $c \in D$ s.t. $\mathsf{KEM.Decaps}^\mathsf{P}(1^\lambda, dk, c) = \perp$:
    $T[c] \leftarrow D[c]$
$fk \leftarrow\$ \mathsf{S_F.Exp}^{\text{PPrimSim}}(1^\lambda, \varepsilon, T : \sigma_\mathsf{F})$
Return $(dk, fk)$
```

```
Adversary $\mathcal{A}_{\mathsf{cguess}^*}^{\text{Ek},\text{Encaps},\text{SExp},\text{RExp},\text{CGuess},\text{Pc},\text{PPrim}}$
$\sigma_{\mathsf{P_{rom}}} \leftarrow\$ \mathsf{P_{rom}.Init}(1^\lambda)$; $\sigma_\mathsf{F} \leftarrow\$ \mathsf{S_F.Init}(1^\lambda)$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow\$ \mathcal{A}_{\mathsf{cca}}^{\text{Ek},\text{EncapsSim},\text{DecapsSim},\ldots}(1^\lambda)$
If $(b = b')$ then return 1 else return 0

PPrimSim$(\mathsf{Op}, (d, k_\mathsf{KEM}), x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
If $d = 1$ then $y \leftarrow\$ \text{PPrim}(\mathsf{Op}, k_\mathsf{KEM}, x, y)$
If $d = 2$ then
    If $D[x] \neq \perp$ and $\text{Pc}(\varepsilon, \{k_\mathsf{KEM}\}, x) \neq \perp$ then
        $\mathsf{P_{rom}.Prog}(1^\lambda, k_\mathsf{KEM}, x, D[x] : \sigma_{\mathsf{P_{rom}}})$
    $y \leftarrow\$ \mathsf{P_{rom}.Op}(1^\lambda, k_\mathsf{KEM}, x, y : \sigma_{\mathsf{P_{rom}}})$
    $\mathbf{K}_x.\mathsf{add}(k_\mathsf{KEM})$
Return $y$

EncapsSim()
$c \leftarrow \text{Encaps}(\varepsilon)$
$(r, k_\mathsf{KEM}) \leftarrow \text{SExp}(\varepsilon, |R| + 1)$
If $b = 1$ then $k \leftarrow \mathsf{P_{rom}.Ls}(1^\lambda, k_\mathsf{KEM}, c : \sigma_{\mathsf{P_{rom}}})$
If $b = 0$ then $k \leftarrow\$ \mathsf{P_{rom}}.\mathcal{R}_\lambda$
$M.\mathsf{add}(c, k)$; $R.\mathsf{add}(r)$; $K.\mathsf{add}(k_\mathsf{KEM})$
Return $(c, k)$

DecapsSim$(c)$
$k_\mathsf{KEM} \leftarrow \text{Pc}(\varepsilon, \mathbf{K}_c, c)$
If $k_\mathsf{KEM} \neq \perp$ then
    Return PPrimSim$(\mathsf{Ls}, (2, k_\mathsf{KEM}), c)$
CGuess$(c)$
$D[c] \leftarrow\$ \mathsf{F.Out}(\lambda)$
Return $D[c]$

SExpSim$(i)$
If $b = 0$ then
    $(c, k) \leftarrow M[i]$; $k_\mathsf{KEM} \leftarrow K[i]$
    $\mathsf{P_{rom}.Prog}(1^\lambda, k_\mathsf{KEM}, c, k : \sigma_{\mathsf{P_{rom}}})$
Return $R[i]$

RExpSim()
$(dk, \cdot) \leftarrow \text{RExp}(\varepsilon)$
If $b = 0$ then
    For $i = 1, \ldots, |M|$ do
        $(c, k) \leftarrow M[i]$; $k_\mathsf{KEM} \leftarrow K[i]$
        $\mathsf{P_{rom}.Prog}(1^\lambda, k_\mathsf{KEM}, c, k : \sigma_{\mathsf{P_{rom}}})$
For $c \in D$ s.t. $\mathsf{KEM.Decaps}^\mathsf{P}(1^\lambda, dk, c) = \perp$:
    $T[c] \leftarrow D[c]$
$fk \leftarrow\$ \mathsf{S_F.Exp}^{\text{PPrimSim}}(1^\lambda, \varepsilon, T : \sigma_\mathsf{F})$
Return $(dk, fk)$
```

Figure 14: Deferred programming and ciphertext guessing adversaries used to transition from G_2 to G_3 in the single-user $\mathsf{U}^{\not\perp}$ security proof

```
Adversary $\mathcal{G}_2^{\text{QUERY},\text{DEFPROG},\text{PPRIM}}(1^\lambda)$
    $(ek, dk) \leftarrow\$ \mathsf{KEM.Kg}(1^\lambda)$
    $\sigma_{\mathsf{P}} \leftarrow\$ \mathsf{P.Init}(1^\lambda)$
    $\sigma_{\mathsf{F}} \leftarrow\$ \mathsf{S_F.Init}(1^\lambda)$
    $b' \leftarrow\$ \mathcal{A}_{\mathsf{cca}}^{\text{EK},\text{ENCAPS},\text{DECAPS},\text{SEXP},\text{REXP},\text{PPRIMSIM}}(1^\lambda)$
    Return $b'$

EK()
    Return $ek$

PPRIMSIM($\mathsf{Op}, (d, k_{\mathsf{KEM}}), x, y$)
    Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
    If $d = 1$ then $y \leftarrow\$ \mathsf{P.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P}})$
    If $d = 2$ then
        If $D[x] \neq \bot$ and $\text{PC}(\varepsilon, \{k_{\mathsf{KEM}}\}, x) \neq \bot$ then
            $\boxed{\text{PPRIM}(\mathsf{Prog}, k_{\mathsf{KEM}}, x, D[x])}$
        $\boxed{y \leftarrow \text{PPRIM}(\mathsf{Op}, k_{\mathsf{KEM}}, x, y)}$
        $\mathbf{K}_x.\mathsf{add}(k_{\mathsf{KEM}})$
    Return $y$

ENCAPS()
    $r \leftarrow\$ \mathsf{KEM.Rand}(\lambda)$
    $(c, k_{\mathsf{KEM}}) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek; r)$
    $\boxed{k \leftarrow \text{QUERY}(k_{\mathsf{KEM}}, c)}$
    $R.\mathsf{add}(r)$
    Return $(c, k)$

DECAPS($c$)
    $k_{\mathsf{KEM}} \leftarrow \text{PC}(\varepsilon, \mathbf{K}_c, c)$
    If $k_{\mathsf{KEM}} \neq \bot$ then
        Return $\text{PPRIMSIM}(\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c)$
    $D[c] \leftarrow\$ \mathsf{F.Out}(\lambda)$
    Return $D[c]$

SEXP($i$)
    $\boxed{\text{DEFPROG}(i)}$; Return $R_u[i]$

REXP()
    For $i = 1, \ldots, |R|$ do $\boxed{\text{DEFPROG}(i)}$
    $T \leftarrow []$
    For $c \in D$ s.t. $\mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk, c) = \bot$:
        $T[c] \leftarrow D[c]$
    $fk \leftarrow\$ \mathsf{S_F.Exp}^{\text{PPRIMSIM}}(1^\lambda, \varepsilon, T : \sigma_{\mathsf{F}})$
    Return $(dk, fk)$
```

```
Adversary $\mathcal{A}_{\mathsf{ow}}^{\text{EK},\text{ENCAPS},\text{SEXP},\text{REXP},\text{GUESS},\text{PC},\text{PPRIM}}$
    $\sigma_{\mathsf{P_{rom}}} \leftarrow\$ \mathsf{P_{rom}.Init}(1^\lambda)$
    $\sigma_{\mathsf{F}} \leftarrow\$ \mathsf{S_F.Init}(1^\lambda)$
    $b' \leftarrow\$ \mathcal{A}_{\mathsf{cca}}^{\text{EK},\text{ENCAPSSIM},\text{DECAPSSIM},\ldots}(1^\lambda)$
    Return $\diamond$

PPRIMSIM($\mathsf{Op}, (d, k_{\mathsf{KEM}}), x, y$)
    If $d = 1$ then $y \leftarrow \text{PPRIM}(\mathsf{Op}, k_{\mathsf{KEM}}, x, y)$
    If $d = 2$ then
        $\text{GUESS}(k_{\mathsf{KEM}})$
        If $D[x] \neq \bot$ and $\text{PC}(\varepsilon, \{k_{\mathsf{KEM}}\}, x) \neq \bot$ then
            $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, x, D[x] : \sigma_{\mathsf{P_{rom}}})$
        $y \leftarrow\$ \mathsf{P_{rom}.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P_{rom}}})$
        $\mathbf{K}_x.\mathsf{add}(k_{\mathsf{KEM}})$
    Return $y$

ENCAPSSIM($u$)
    $c \leftarrow \text{ENCAPS}(u)$; $k \leftarrow\$ \mathsf{P_{rom}}.\mathcal{R}_\lambda$
    $M.\mathsf{add}(c, k)$; Return $(c, k)$

DECAPSSIM($c$)
    $k_{\mathsf{KEM}} \leftarrow \text{PC}(\varepsilon, \mathbf{K}_c, c)$
    If $k_{\mathsf{KEM}} \neq \bot$ then return $\text{PPRIM}(\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c)$
    $D[c] \leftarrow\$ \mathsf{F.Out}(\lambda)$
    Return $D[c]$

SEXPSIM($i$)
    $(c, k) \leftarrow M[i]$; $(r, k_{\mathsf{KEM}}) \leftarrow \text{SEXP}(\varepsilon, i)$
    $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, c, k : \sigma_{\mathsf{P_{rom}}})$
    Return $r$

REXPSIM()
    $(dk, K) \leftarrow \text{REXP}(\varepsilon)$
    For $i = 1, \ldots, |M|$ do
        $(c, k) \leftarrow M[i]$; $k_{\mathsf{KEM}} \leftarrow K[i]$
        $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, c, k : \sigma_{\mathsf{P_{rom}}})$
    For $c \in D$ s.t. $\mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk, c) = \bot$:
        $T[c] \leftarrow D[c]$
    $fk \leftarrow\$ \mathsf{S_F.Exp}^{\text{PPRIMSIM}}(1^\lambda, \varepsilon, T : \sigma_{\mathsf{F}})$
    Return $(dk, fk)$
```

Figure 15: Deferred programming and one-wayness adversaries used to analyze G_3 in the single-user $\mathsf{U}^{\not\perp}$ security proof

argue that bad^* will be set in G_3 whenever bad would be set in $G^{\mathsf{def\text{-}prog}}_{\mathsf{P_{rom}},\mathcal{G}_1,0}$.

By our discussion in Section C.1 the deferred programming bad flag corresponds to either two calls to Query being made with the same input or Query and PPrim being called with the same (k_{KEM}, c) before the corresponding DefProg query is made. By our assumption that $\mathcal{A}_{\mathsf{cca}}$ never repeats Decaps queries, the first case will not occur. For analyzing the second case, there are four different place PPrim is queried.

- PPrimSim: If the Query query happens first (inside Decaps) then c will be added to D and the DefProg query (inside PPrimSim) prevents bad being set. If the PPrim query happens first (inside PPrimSim), then k_{KEM} is added to $\mathbf{K}_c$ and the check at the beginning of Decaps prevents the Query query from being made.
- Encaps, SExp, or RExp: Each of these oracles only query PPrim on ciphertexts that were sampled by Encaps. By our assumption that the adversary does not forward challenge ciphertexts to Decaps, the corresponding Query query can only occur before the Encaps query, in which case bad^* would be set.

Analysis of game G_3. Now the decapsulation oracle can be simulated without use of the decapsulation key (except via Pc) and the proof concludes in a manner quite similar to the security proof from Hashed KEM given in Section C. Considering G_3, we can see that the way its behavior depends on the bit b is by either directly using $\mathsf{P_{rom}}$ inside Encaps to set k or sampling k at random and then later attempting to program it into $\mathsf{P_{rom}}$ when exposures happen. This is exactly the distinction that deferred programming analyzes.

In particular, consider the deferred programming adversary $\mathcal{G}_2$ in Fig. 15. It simulates G_3 for $\mathcal{A}_{\mathsf{cca}}$ using its oracles for any calls to $\mathsf{P_{rom}}$, in particular using Query and DefProg where the behavior of G_3 depends on b. It uses the previously discussed shorthand for simulating Decaps and outputs the same guess that $\mathcal{A}_{\mathsf{cca}}$ does. Consequently, $2\Pr[G_3(\lambda)] - 1 = \mathsf{Adv}^{\mathsf{def\text{-}prog}}_{\mathsf{P_{rom}},\mathcal{G}_2}(\lambda) \leq \Pr[G^{\mathsf{def\text{-}prog}}_{\mathsf{P_{rom}},\mathcal{G}_2,0}(\lambda) \text{ sets } \mathsf{bad}]$.

We will use the OW*-PCA adversary $\mathcal{A}_{\mathsf{ow}}$ from the same figure to bound the probability of this bad flag. It simulates the $b = 0$ case of G_3, using its oracles to simulate most uses of KEM and P while locally simulating the behavior of $\mathsf{P_{rom}}$. It makes a guess for each key that is queried to $\mathsf{P_{rom}}$. We analyze the different ways bad could be set by $\mathcal{G}_2$ and show each results in $\mathcal{A}_{\mathsf{ow}}$ setting win, coll, or early. Hence $\Pr[G^{\mathsf{def\text{-}prog}}_{\mathsf{P_{rom}},\mathcal{G}_2,0}(\lambda) \text{ sets } \mathsf{bad}] \leq \mathsf{Adv}^{\mathsf{ow^*\text{-}pca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) + \Pr[G^{\mathsf{ow^*\text{-}pca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) \text{ sets } \mathsf{coll} \text{ or } \mathsf{early}]$.

Clearly, if $\mathcal{G}_2$ calls Query on the same inputs twice, then coll will be set by $\mathcal{A}_{\mathsf{ow}}$. If $\mathcal{G}_2$ calls PPrim on some (k_{KEM}, c), then $\mathcal{A}_{\mathsf{ow}}$ will guess this key. A later Query call for these values will result in early being set. If the Query call happened before the PPrim call (and the corresponding DefProg call has not happened yet), then the key guess will set win.

Putting all of our bounds together gives

$$\begin{aligned}\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{U}^{\not\perp}[\mathsf{KEM},\mathsf{F}],\mathsf{S},\mathsf{P}\times\mathsf{P_{rom}},\mathcal{A}_{\mathsf{cca}},}(\lambda) \leq & 2\left(\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S_F},\mathsf{P}\times\mathsf{P_{rom}},\mathcal{A}_{\mathsf{prf}}}(\lambda) + \mathsf{Adv}^{\mathsf{cguess^*\text{-}pca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{cguess^*}}}(\lambda)\right)\\ & + \mathsf{Adv}^{\mathsf{ow^*\text{-}pca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) + \Pr[G^{\mathsf{ow^*\text{-}pca}}_{\mathsf{KEM},\mathsf{P},\mathcal{A}_{\mathsf{ow}}}(\lambda) \text{ sets } \mathsf{coll} \text{ or } \mathsf{early}].\end{aligned}$$

C.4.2 Multi-user Proof

We provide an alternate proof for when $\mathcal{A}_{\mathsf{cca}}$ is a multi-user adversary. Much of the analysis remains the same, but now we have to account for $\mathsf{P_{rom}}$ query collisions between different users.

```
Games $\boxed{\mathrm{G}'_2(\lambda)}$, $\mathrm{G}'_3(\lambda)$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow\!\$\ \mathsf{KEM.Kg}(1^\lambda)$
$(\sigma_{\mathsf{P}}, \sigma_{\mathsf{P_{rom}}}) \leftarrow\!\$\ \mathsf{P} \times \mathsf{P_{rom}.Init}(1^\lambda)$
$\sigma_{\mathsf{F}} \leftarrow\!\$\ \mathsf{S_F.Init}(1^\lambda)$
$b \leftarrow\!\$\ \{0,1\}$
$b' \leftarrow\!\$\ \mathcal{A}_{\mathsf{cca}}^{\text{EK,ENCAPS,DECAPS,SEXP,REXP,PPRIM}}(1^\lambda)$
Return $(b = b')$

EK($u$)
Return $ek_u$

PPRIM($\mathsf{Op}, (d, k_{\mathsf{KEM}}), x, y$)
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
If $d = 1$ then $y \leftarrow\!\$\ \mathsf{P.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P}})$
If $d = 2$ then
  $\mathbf{U} \leftarrow \{\, u \,:\, D_u[x] \neq \bot \,\}$
  For $u \in \mathbf{U}$ if $\mathrm{PC}(u, \{k_{\mathsf{KEM}}\}, x) \neq \bot$ then
    $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, x, D_u[x] : \sigma_{\mathsf{P_{rom}}})$
  $y \leftarrow\!\$\ \mathsf{P_{rom}.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P_{rom}}})$
  $\mathbf{K}_x.\mathsf{add}(k_{\mathsf{KEM}})$
Return $y$

ENCAPS($u$)
$r \leftarrow\!\$\ \mathsf{KEM.Rand}(\lambda)$
$(c, k_{\mathsf{KEM}}) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek_u; r)$
If $\exists v : D_v[c] \neq \bot$ then $\mathsf{bad}^* \leftarrow \mathsf{true}$
If $b = 1$ then $k \leftarrow \mathsf{P_{rom}.Ls}(1^\lambda, k_{\mathsf{KEM}}, c : \sigma_{\mathsf{P_{rom}}})$
If $b = 0$ then $k \leftarrow\!\$\ \mathsf{P_{rom}}.\mathcal{R}_\lambda$
$M_u.\mathsf{add}(c, k)$; $R_u.\mathsf{add}(r)$; $K_u.\mathsf{add}(k_{\mathsf{KEM}})$
Return $(c, k)$
```

```
DECAPS($u, c$)
$k_{\mathsf{KEM}} \leftarrow \mathrm{PC}(u, \mathbf{K}_c, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
  Return PPRIM($\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c$)
$\mathbf{V} \leftarrow \{\, v \,:\, M_v\langle c\rangle \neq \bot \,\}$
$(\cdot, k_{\mathsf{KEM}}) \leftarrow \mathrm{DC}(u, \mathbf{V}, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
  Return PPRIM($\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c$)
$\mathbf{V} \leftarrow \{\, v \,:\, D_v[c] \neq \bot \,\}$
$(v, \cdot) \leftarrow \mathrm{DC}(u, \mathbf{V}, c)$
If $v \neq \bot$ then return $D_v[c]$
$k_{\mathsf{KEM}} \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
  $\boxed{D_u[c] \leftarrow \mathsf{P_{rom}.Ls}(1^\lambda, k_{\mathsf{KEM}}, c : \sigma_{\mathsf{P_{rom}}})}$
  $D_u[c] \leftarrow\!\$\ \mathsf{F.Out}(\lambda)$
Else $D_u[c] \leftarrow\!\$\ \mathsf{F.Out}(\lambda)$
Return $D_u[c]$

SEXP($u, i$)
If $b = 0$ then
  $(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow K_u[i]$
  $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, c, k : \sigma_{\mathsf{P_{rom}}})$
Return $R_u[i]$

REXP($u$)
If $b = 0$ then
  For $i = 1, \ldots, |M_u|$ do
    $(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow K_u[i]$
    $\mathsf{P_{rom}.Prog}(1^\lambda, k_{\mathsf{KEM}}, c, k : \sigma_{\mathsf{P_{rom}}})$
For $c \in D_u$ if $\mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c) = \bot$ then
  $T_u[c] \leftarrow D_u[c]$
$fk_u \leftarrow\!\$\ \mathsf{S_F.Exp}^{\text{PPRIM}}(1^\lambda, u, T_u : \sigma_{\mathsf{F}})$
Return $(dk_u, fk_u)$
```

Figure 16: Games for multi-user $\mathsf{U}^{\not\perp}$ security proof

Transition to game G'_2. We start with game G'_2, shown in Fig. 16. We claim that this game is equivalent to G_1. Oracles EK and SEXP remain unchanged. In PPRIM we added the set $\mathbf{K}$ to store keys that were queried to $\mathsf{P_{rom}}$ and we added a for loop which does nothing because the highlighted code is omitted.

Oracle DECAPS starts by checking three ways it can correctly respond to the query without directly decrypting the ciphertext. First, we use PC to check if the ciphertext was previously queried to PPRIM with the correct k_{KEM}. Next, via M, it uses DC to learn if the ciphertext was returned by a prior ENCAPS query for a *different* user that happens to to decapsulate it to the same k_{KEM}. If either of these checks succeed, the oracle uses $\mathsf{P_{rom}}$ honestly (via PPRIM) to respond to the query. Finally, it checks if c was previously queried to DECAPS for a different user that happens to decapsulate it to the same k_{KEM} and if so uses that user's D to respond to the query.

If these all fail, then the oracle resorts to performing decapsulation and responding in the correct way. In this case, we store the result in a table D which is parameterized by the user. It is used in REXP to recompute T_u when needed.

```
Adversary $\mathcal{H}_1^{\text{QUERY},\text{DEFPROG},\text{PPRIM}}(1^\lambda)$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow\!\!\$\ \mathsf{KEM.Kg}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow\!\!\$\ \mathsf{P.Init}(1^\lambda)$; $\sigma_{\mathsf{F}} \leftarrow\!\!\$\ \mathsf{S_F.Init}(1^\lambda)$
$b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow\!\!\$\ \mathcal{A}_{\mathsf{cca}}^{\text{EK},\text{ENCAPS},\text{DECAPS},\text{SEXP},\text{REXP},\text{PPRIMSIM}}(1^\lambda)$
Return $\diamond$

EK$(u)$
Return $ek_u$

PPRIM$(\mathsf{Op}, (d, k_{\mathsf{KEM}}), x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
If $d = 1$ then $y \leftarrow\!\!\$\ \mathsf{P.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P}})$
If $d = 2$ then
    $\mathbf{U} \leftarrow \{\, u : D_u[x] \neq \bot \,\}$
    For $u \in \mathbf{U}$ if PC$(u, \{k_{\mathsf{KEM}}\}, x) \neq \bot$ then
        $\boxed{\text{DEFPROG}(I\langle u, k_{\mathsf{KEM}}, x\rangle)}$
    $\boxed{y \leftarrow \text{PPRIM}(\mathsf{Op}, k_{\mathsf{KEM}}, x, y)}$
    $\mathbf{K}_x.\mathsf{add}(k_{\mathsf{KEM}})$
Return $y$

ENCAPS$(u)$
$r \leftarrow\!\!\$\ \mathsf{KEM.Rand}(\lambda)$
$(c, k_{\mathsf{KEM}}) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek_u; r)$
If $\exists v : D_v[c] \neq \bot$ then $\mathsf{bad}^* \leftarrow \mathsf{true}$
If $b = 1$ then $\boxed{k \leftarrow \text{PPRIM}(\mathsf{Ls}, k_{\mathsf{KEM}}, c)}$
If $b = 0$ then $k \leftarrow\!\!\$\ \mathsf{P_{rom}}.\mathcal{R}_\lambda$
$M_u.\mathsf{add}(c, k)$; $R_u.\mathsf{add}(r)$; $K_u.\mathsf{add}(k_{\mathsf{KEM}})$
Return $(c, k)$

DECAPS$(u, c)$
$k_{\mathsf{KEM}} \leftarrow$ PC$(u, \mathbf{K}_c, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
    Return PPRIMSIM$(\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c)$
$\mathbf{V} \leftarrow \{\, v : M_v\langle c\rangle \neq \bot \,\}$
$(\cdot, k_{\mathsf{KEM}}) \leftarrow$ DC$(u, \mathbf{V}, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
    Return PPRIMSIM$(\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c)$
$\mathbf{V} \leftarrow \{\, v : D_v[c] \neq \bot \,\}$
$(v, \cdot) \leftarrow$ DC$(u, \mathbf{V}, c)$
If $v \neq \bot$ then return $D_v[c]$
$k_{\mathsf{KEM}} \leftarrow \mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
    $t \leftarrow t + 1$; $I.\mathsf{add}((u, k_{\mathsf{KEM}}, c), t)$
    $\boxed{D_u[c] \leftarrow \text{QUERY}(k_{\mathsf{KEM}}, c)}$
Else $D_u[c] \leftarrow\!\!\$\ \mathsf{F.Out}(\lambda)$
Return $D_u[c]$

SEXP$(u, i)$
If $b = 0$ then
    $(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow K_u[i]$
    $\boxed{\text{PPRIM}(\mathsf{Prog}, k_{\mathsf{KEM}}, c, k)}$
Return $R_u[i]$

REXP$(u)$
If $b = 0$ then
    For $i = 1, \ldots, |M_u|$ do
        $(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow K_u[i]$
        $\boxed{\text{PPRIM}(\mathsf{Prog}, k_{\mathsf{KEM}}, c, k)}$
For $c \in D_u$ if $\mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c) = \bot$ then
    $T_u[c] \leftarrow D_u[c]$
$fk_u \leftarrow\!\!\$\ \mathsf{S_F.Exp}^{\text{PPRIM}}(1^\lambda, u, T_u : \sigma_{\mathsf{F}})$
Return $(dk_u, fk_u)$
```

Figure 17: Deferred programming adversary used to transition from G_2' to G_3' in the multi-user $\mathsf{U}^{\not\perp}$ security proof

---

Extra queries to P are added by the calls to PC and DC in PPRIM and DECAPS, as well as the use of $\mathsf{KEM.Decaps}^{\mathsf{P}}$ in REXP. In almost all cases these queries were already made inside of DECAPS in G_1 and so cannot be detected. The exception is the first call to DC in DECAPS. This instead relies on the fact that the ciphertext was produced in ENCAPS and that KEM is query consistent.

Transition to game G_3'. Next we switch to G_3' wherein the $\mathsf{KEM.Decaps}^{\mathsf{P}}$ is no longer needed because the value put in D_u will be uniformly random whether or not k_{KEM} is $\bot$. Note that G_3' is identical to G_2' except that use of $\mathsf{P_{rom}}$ is replaced with random sampling in DECAPS and inside of PPRIM if we are given inputs for which DECAPS previously randomly sampled then we try to program $\mathsf{P_{rom}}$ to be consistent. This difference is exhibited by the deferred programming adversary $\mathcal{H}_1$ shown in Fig. 17 for which we claim $\Pr[G_2'(\lambda)] - \Pr[G_3'(\lambda)] \leq \Pr[G^{\mathsf{def\text{-}prog}}_{\mathsf{P_{rom}},\mathcal{H}_1,0}(\lambda) \text{ sets } \mathsf{bad}] \leq \Pr[G_3' \text{ sets } \mathsf{bad}^*]$. The reasoning for this follows analogously to that used in the single-user case, with some additional subtleties that required the addition of the two uses of DC in DECAPS.

```
Adversary $\mathcal{B}_{\mathsf{cguess}^*}^{\text{Ek,Encaps,SExp,RExp,CGuess,Pc,Dc,PPrim}}$
    $\sigma_{\mathsf{P}_{\mathsf{rom}}} \leftarrow\!\$\ \mathsf{P}_{\mathsf{rom}}.\mathsf{Init}(1^\lambda)$
    $\sigma_{\mathsf{F}} \leftarrow\!\$\ \mathsf{S}_{\mathsf{F}}.\mathsf{Init}(1^\lambda)$
    $b \leftarrow\!\$\ \{0,1\}$
    $b' \leftarrow\!\$\ \mathcal{A}_{\mathsf{cca}}^{\text{Ek,EncapsSim,DecapsSim}}(1^\lambda)$
    If $(b = b')$ then return 1 else return 0

PPrimSim($\mathsf{Op}, (d, k_{\mathsf{KEM}}), x, y$)
    Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
    If $d = 1$ then $y \leftarrow\!\$\ \text{PPrim}(\mathsf{Op}, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P}})$
    If $d = 2$ then
        $\mathbf{U} \leftarrow \{\ u\ :\ D_u[x] \neq \bot\ \}$
        For $u \in \mathbf{U}$ if $\text{Pc}(u, \{k_{\mathsf{KEM}}\}, x) \neq \bot$ then
            $\mathsf{P}_{\mathsf{rom}}.\mathsf{Prog}(1^\lambda, k_{\mathsf{KEM}}, x, D_u[x] : \sigma_{\mathsf{P}_{\mathsf{rom}}})$
        $y \leftarrow\!\$\ \mathsf{P}_{\mathsf{rom}}.\mathsf{Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P}_{\mathsf{rom}}})$
        $\mathbf{K}_x.\mathsf{add}(k_{\mathsf{KEM}})$
    Return $y$

EncapsSim($u$)
    $c \leftarrow \text{Encaps}(u)$
    $(r, k_{\mathsf{KEM}}) \leftarrow \text{SExp}(u, |R_u| + 1)$
    If $b = 1$ then $k \leftarrow \mathsf{P}_{\mathsf{rom}}.\mathsf{Ls}(1^\lambda, k_{\mathsf{KEM}}, c : \sigma_{\mathsf{P}_{\mathsf{rom}}})$
    If $b = 0$ then $k \leftarrow\!\$\ \mathsf{P}_{\mathsf{rom}}.\mathcal{R}_\lambda$
    $M_u.\mathsf{add}(c, k)$; $R_u.\mathsf{add}(r)$; $K_u.\mathsf{add}(k_{\mathsf{KEM}})$
    Return $(c, k)$

DecapsSim($u, c$)
    $k_{\mathsf{KEM}} \leftarrow \text{Pc}(u, \mathbf{K}_c, c)$
    If $k_{\mathsf{KEM}} \neq \bot$ then
        Return PPrimSim($\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c$)
    $\mathbf{V} \leftarrow \{\ v\ :\ M_v\langle c\rangle \neq \bot\ \}$
    $(\cdot, k_{\mathsf{KEM}}) \leftarrow \text{Dc}(u, \mathbf{V}, c)$
    If $k_{\mathsf{KEM}} \neq \bot$ then
        Return PPrimSim($\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c$)
    $\mathbf{V} \leftarrow \{\ v\ :\ D_v[c] \neq \bot\ \}$
    $(v, \cdot) \leftarrow \text{Dc}(u, \mathbf{V}, c)$
    If $v \neq \bot$ then return $D_v[c]$
    CGuess($c$)
    $D_u[c] \leftarrow\!\$\ \mathsf{F}.\mathsf{Out}(\lambda)$
    Return $D_u[c]$

SExpSim($u, i$)
    If $b = 0$ then
        $(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow K_u[i]$
        $\mathsf{P}_{\mathsf{rom}}.\mathsf{Prog}(1^\lambda, k_{\mathsf{KEM}}, c, k : \sigma_{\mathsf{P}_{\mathsf{rom}}})$
    Return $R_u[i]$

RExpSim($u$)
    $(dk_u, \cdot) \leftarrow \text{RExp}(u)$
    If $b = 0$ then
        For $i = 1, \ldots, |M_u|$ do
            $(c, k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow K_u[i]$
            $\mathsf{P}_{\mathsf{rom}}.\mathsf{Prog}(1^\lambda, k_{\mathsf{KEM}}, c, k : \sigma_{\mathsf{P}_{\mathsf{rom}}})$
    For $c \in D_u$ if $\mathsf{KEM}.\mathsf{Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c) = \bot$ then
        $T_u[c] \leftarrow D_u[c]$
    $fk_u \leftarrow\!\$\ \mathsf{S}_{\mathsf{F}}.\mathsf{Exp}^{\text{PPrimSim}}(1^\lambda, u, T_u : \sigma_{\mathsf{F}})$
    Return $(dk_u, fk_u)$
```

Figure 18: Ciphertext guessing adversary used to transition from G_2' to G_3' in the multi-user $\mathsf{U}^{\not\perp}$ security proof

Recall that the deferred programming bad flag corresponds to either two calls to Query being made with the same input or Query and PPrim being called with the same (k_{KEM}, c) before the corresponding DefProg query is made. In the single-user case, our assumption that $\mathcal{A}_{\mathsf{cca}}$ never repeats Decaps queries implied that the first case will not occur. In the multi-user case, we have to consider the case that $\mathcal{A}_{\mathsf{cca}}$ queried the same c to two different users. The second use of Dc in Decaps ensures that if this occurs (and the same k_{KEM} underlies the ciphertext both times) then the repeat query to Query is avoided because we respond early using D_v during the second query. For analyzing the second case, there are four different place PPrim is queried.

- PPrimSim: If the Query query happens first (inside Decaps) then c will be added to D_u and the DefProg query (inside PPrimSim) prevents bad being set. Here it is import that we checked all users to which c has been queried. If the PPrim query happens first (inside PPrimSim), then k_{KEM} is added to $\mathbf{K}_c$ and the check at the beginning of Decaps prevents the Query query from being made.
- Encaps, SExp, or RExp: Each of these oracles only query PPrim on ciphertexts that were

```
Adversary $\mathcal{H}_2^{\text{QUERY},\text{DEFPROG},\text{PPRIM}}(1^\lambda)$
$(ek_{(\cdot)}, dk_{(\cdot)}) \leftarrow\!\!\$\ \mathsf{KEM.Kg}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow\!\!\$\ \mathsf{P.Init}(1^\lambda)$
$\sigma_{\mathsf{F}} \leftarrow\!\!\$\ \mathsf{S_F.Init}(1^\lambda)$
$b' \leftarrow\!\!\$\ \mathcal{A}_{\mathsf{cca}}^{\text{EK},\text{ENCAPS},\text{DECAPS},\text{SEXP},\text{REXP},\text{PPRIMSIM}}(1^\lambda)$
Return $b'$

EK($u$)
Return $ek_u$

PPRIMSIM($\mathsf{Op}, (d, k_{\mathsf{KEM}}), x, y$)
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
If $d = 1$ then $y \leftarrow\!\!\$\ \mathsf{P.Op}(1^\lambda, k_{\mathsf{KEM}}, x, y : \sigma_{\mathsf{P}})$
If $d = 2$ then
    $\mathbf{U} \leftarrow \{\, u : D_u[x] \neq \bot \,\}$
    For $u \in \mathbf{U}$ if $\text{PC}(u, \{k_{\mathsf{KEM}}\}, x) \neq \bot$ then
        $\boxed{\text{PPRIM}(\mathsf{Prog}, k_{\mathsf{KEM}}, x, D_u[x])}$
    $\boxed{y \leftarrow \text{PPRIM}(\mathsf{Op}, k_{\mathsf{KEM}}, x, y)}$
    $\mathbf{K}_x.\mathsf{add}(k_{\mathsf{KEM}})$
Return $y$

ENCAPS($u$)
$r \leftarrow\!\!\$\ \mathsf{KEM.Rand}(\lambda)$
$(c, k_{\mathsf{KEM}}) \leftarrow \mathsf{KEM.Encaps}^{\mathsf{P}}(1^\lambda, ek_u; r)$
If $\exists \mathsf{v} : D_{\mathsf{v}}[c] \neq \bot$ then $\mathsf{bad}^* \leftarrow \mathsf{true}$
$\boxed{k \leftarrow \text{QUERY}(k_{\mathsf{KEM}}, c)}$
$t \leftarrow t + 1$; $I.\mathsf{add}((u, |R_u| + 1), t)$
$M_u.\mathsf{add}(c, k)$; $R_u.\mathsf{add}(r)$; $K_u.\mathsf{add}(k_{\mathsf{KEM}})$
Return $(c, k)$

DECAPS($u, c$)
$k_{\mathsf{KEM}} \leftarrow \text{PC}(u, \mathbf{K}_c, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
    Return $\text{PPRIMSIM}(\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c)$
$\mathbf{V} \leftarrow \{\, \mathsf{v} : M_{\mathsf{v}}\langle c \rangle \neq \bot \,\}$
$(\cdot, k_{\mathsf{KEM}}) \leftarrow \text{DC}(u, \mathbf{V}, c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
    Return $\text{PPRIMSIM}(\mathsf{Ls}, (2, k_{\mathsf{KEM}}), c)$
$\mathbf{V} \leftarrow \{\, \mathsf{v} : D_{\mathsf{v}}[c] \neq \bot \,\}$
$(\mathsf{v}, \cdot) \leftarrow \text{DC}(u, \mathbf{V}, c)$
If $\mathsf{v} \neq \bot$ then return $D_{\mathsf{v}}[c]$
$D_u[c] \leftarrow\!\!\$\ \mathsf{F.Out}(\lambda)$
Return $D_u[c]$

SEXP($u, i$)
$\boxed{\text{DEFPROG}(I\langle u, i \rangle)}$
Return $R_u[i]$

REXP($u$)
For $i = 1, \ldots, |M_u|$ do $\boxed{\text{DEFPROG}(I\langle u, i \rangle)}$
For $c \in D_u$ if $\mathsf{KEM.Decaps}^{\mathsf{P}}(1^\lambda, dk_u, c) = \bot$ then
    $T_u[c] \leftarrow D_u[c]$
$fk_u \leftarrow\!\!\$\ \mathsf{S_F.Exp}^{\text{PPRIMSIM}}(1^\lambda, u, T_u : \sigma_{\mathsf{F}})$
Return $(dk_u, fk_u)$
```

Figure 19: Deferred programming adversary used to analyze G_3' in the multi-user $\mathsf{U}^{\not\perp}$ security proof

sampled by ENCAPS. In the single-user case, our assumption that the adversary does not forward challenge ciphertexts to DECAPS implied that the corresponding QUERY query can only occur before the ENCAPS query, in which case bad^* would be set. Now we must also consider the case that the challenge ciphertext was forwarded to a different user's decapsulation oracle. The first use of DC will prevent this from resulting in a matching call to QUERY because it instead recovers k_{KEM} and computes the output directly.

The above analysis covers the bulk of why the game transitions needed to be modified in the multi-user case. These differences are mostly "carried along" in the rest of the proof without significant difference in analysis from the single-user case.

Now we bound the probability of this bad flag using the CGUESS*-PCDCA security of KEM, getting $\Pr[\mathrm{G}_3'(\lambda) \text{ sets } \mathsf{bad}^*] \leq \mathsf{Adv}^{\mathsf{cguess}^*\text{-}\mathsf{pca}}_{\mathsf{KEM},\mathsf{P},\mathcal{B}_{\mathsf{cguess}^*}}(\lambda)$ where $\mathcal{B}_{\mathsf{cguess}^*}$ is defined in Fig. 18. It runs $\mathcal{A}_{\mathsf{cca}}$, using its own oracles to simulate most uses of KEM and P. It simulates decapsulation without $\mathsf{KEM.Decaps}$. To simulate encapsulation it needs to know k_{KEM} when $b = 1$, which it obtains by a call to SEXP (which does not affect setting cguess). It queries CGUESS with each ciphertext added to a table D_u, so the claim follows because cguess will be set in $\mathrm{G}^{\mathsf{cguess}^*\text{-}\mathsf{pcdca}}$ whenever bad^* would be set in G_3'.

```
Adversary $\mathcal{B}_{\mathsf{ow}}^{\textsc{Ek},\textsc{Encaps},\textsc{SExp},\textsc{RExp},\textsc{Guess},\textsc{Pc},\textsc{Dc},\textsc{PPrim}}$
$\sigma_{\mathsf{P}_{\mathsf{rom}}} \leftarrow\$ \mathsf{P}_{\mathsf{rom}}.\mathsf{Init}(1^\lambda)$
$\sigma_{\mathsf{F}} \leftarrow\$ \mathsf{S}_{\mathsf{F}}.\mathsf{Init}(1^\lambda)$
$b' \leftarrow\$ \mathcal{A}_{\mathsf{cca}}^{\textsc{Ek},\textsc{EncapsSim},\textsc{DecapsSim},\ldots}(1^\lambda)$
Return $\diamond$

$\textsc{PPrimSim}(\mathsf{Op},(d,k_{\mathsf{KEM}}),x,y)$
Require $\mathsf{Op} \in \{\mathsf{Ls},\mathsf{Prog}\}$
If $d=1$ then $y \leftarrow\$ \textsc{PPrim}(\mathsf{Op},k_{\mathsf{KEM}},x,y)$
If $d=2$ then
    $\textsc{Guess}(k_{\mathsf{KEM}})$
    $\mathbf{U} \leftarrow \{\, u : D_u[x] \neq \bot \,\}$
    For $u \in \mathbf{U}$ if $\textsc{Pc}(u,\{k_{\mathsf{KEM}}\},x) \neq \bot$ then
        $\mathsf{P}_{\mathsf{rom}}.\mathsf{Prog}(1^\lambda,k_{\mathsf{KEM}},x,D_u[x] : \sigma_{\mathsf{P}_{\mathsf{rom}}})$
    $y \leftarrow\$ \mathsf{P}_{\mathsf{rom}}.\mathsf{Op}(1^\lambda,k_{\mathsf{KEM}},x,y : \sigma_{\mathsf{P}_{\mathsf{rom}}})$
    $\mathbf{K}_x.\mathsf{add}(k_{\mathsf{KEM}})$
Return $y$

$\textsc{EncapsSim}(u)$
$c \leftarrow \textsc{Encaps}(u)$
$k \leftarrow\$ \mathsf{P}_{\mathsf{rom}}.\mathcal{R}_\lambda$
$M_u.\mathsf{add}(c,k)$
Return $(c,k)$

$\textsc{DecapsSim}(u,c)$
$k_{\mathsf{KEM}} \leftarrow \textsc{Pc}(u,\mathbf{K}_c,c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
    Return $\textsc{PPrim}(\mathsf{Ls},(2,k_{\mathsf{KEM}}),c)$
$\mathbf{V} \leftarrow \{\, v : M_v\langle c\rangle \neq \bot \,\}$
$(\cdot,k_{\mathsf{KEM}}) \leftarrow \textsc{Dc}(u,\mathbf{V},c)$
If $k_{\mathsf{KEM}} \neq \bot$ then
    Return $\textsc{PPrim}(\mathsf{Ls},(2,k_{\mathsf{KEM}}),c)$
$\mathbf{V} \leftarrow \{\, v : D_v[c] \neq \bot \,\}$
$(v,\cdot) \leftarrow \textsc{Dc}(u,\mathbf{V},c)$
If $v \neq \bot$ then return $D_v[c]$
$D_u[c] \leftarrow\$ \mathsf{F}.\mathsf{Out}(\lambda)$
Return $D_u[c]$

$\textsc{SExpSim}(u,i)$
$(c,k) \leftarrow M_u[i]$; $(r,k_{\mathsf{KEM}}) \leftarrow \textsc{SExp}(u,i)$
$\mathsf{P}_{\mathsf{rom}}.\mathsf{Prog}(1^\lambda,k_{\mathsf{KEM}},c,k : \sigma_{\mathsf{P}_{\mathsf{rom}}})$
Return $r$

$\textsc{RExpSim}(u)$
$(dk_u,K) \leftarrow \textsc{RExp}(u)$
For $i=1,\ldots,|M_u|$ do
    $(c,k) \leftarrow M_u[i]$; $k_{\mathsf{KEM}} \leftarrow K_u[i]$
    $\mathsf{P}_{\mathsf{rom}}.\mathsf{Prog}(1^\lambda,k_{\mathsf{KEM}},c,k : \sigma_{\mathsf{P}_{\mathsf{rom}}})$
For $c \in D_u$ if $\mathsf{KEM}.\mathsf{Decaps}^{\mathsf{P}}(1^\lambda,dk_u,c) = \bot$ then
    $T_u[c] \leftarrow D_u[c]$
$fk_u \leftarrow\$ \mathsf{S}_{\mathsf{F}}.\mathsf{Exp}^{\textsc{PPrim}}(1^\lambda,u,T_u : \sigma_{\mathsf{F}})$
Return $(dk_u,fk_u)$
```

Figure 20: One-wayness adversary used to analyze G_3' in the multi-user $\mathsf{U}^{\not\perp}$ security proof

Analysis of game G_3'. Now the decapsulation oracle can be simulated without use of the decapsulation key (except via $\textsc{Pc}$ and $\textsc{Dc}$). Considering G_3', we can see that the way its behavior depends on the bit b is by either directly using $\mathsf{P}_{\mathsf{rom}}$ inside $\textsc{Encaps}$ to set k or sampling k at random and then later attempting to program it into $\mathsf{P}_{\mathsf{rom}}$ when exposures happen. This is exactly the distinction that deferred programming analyzes.

In particular, consider the deferred programming adversary $\mathcal{H}_2$ in Fig. 19. It simulates G_3' for $\mathcal{A}_{\mathsf{cca}}$ using its oracles for any calls to $\mathsf{P}_{\mathsf{rom}}$, in particular using $\textsc{Query}$ and $\textsc{DefProg}$ where the behavior of G_3 depends on b. It outputs the same guess that $\mathcal{A}_{\mathsf{cca}}$ does. Consequently, $2\Pr[\mathrm{G}_3'(\lambda)] - 1 = \mathsf{Adv}^{\mathsf{def\text{-}prog}}_{\mathsf{P}_{\mathsf{rom}},\mathcal{H}_2}(\lambda) \le \Pr[\mathrm{G}^{\mathsf{def\text{-}prog}}_{\mathsf{P}_{\mathsf{rom}},\mathcal{H}_2,0}(\lambda) \text{ sets } \mathsf{bad}]$.

We will use the OW*-PCDCA adversary $\mathcal{B}_{\mathsf{ow}}$ from Fig. 20 to bound the probability of this bad flag. It simulates the $b=0$ case of G_3', using its oracles to simulate most uses of KEM and P while locally simulating the behavior of $\mathsf{P}_{\mathsf{rom}}$. It makes a guess for each key that is queried to $\mathsf{P}_{\mathsf{rom}}$. We analyze the different ways bad could be set by $\mathcal{G}_2$ and show each results in $\mathcal{A}_{\mathsf{ow}}$ setting win, coll, or early. Hence $\Pr[\mathrm{G}^{\mathsf{def\text{-}prog}}_{\mathsf{P}_{\mathsf{rom}},\mathcal{H}_2,0}(\lambda) \text{ sets } \mathsf{bad}] \le \mathsf{Adv}^{\mathsf{ow^*\text{-}pcdca}}_{\mathsf{KEM},\mathsf{P},\mathcal{B}_{\mathsf{ow}}}(\lambda) + \Pr[\mathrm{G}^{\mathsf{ow^*\text{-}pcdca}}_{\mathsf{KEM},\mathsf{P},\mathcal{B}_{\mathsf{ow}}}(\lambda) \text{ sets } \mathsf{coll} \text{ or } \mathsf{early}]$.

Clearly, if $\mathcal{H}_2$ calls $\textsc{Query}$ on the same inputs twice, then coll will be set by $\mathcal{B}_{\mathsf{ow}}$. If $\mathcal{H}_2$ calls $\textsc{PPrim}$ on some (k_{KEM},c), then $\mathcal{B}_{\mathsf{ow}}$ will guess this key. A later $\textsc{Query}$ for these values will result in early being set. If the $\textsc{Query}$ call happened before the $\textsc{PPrim}$ call (and the corresponding $\textsc{DefProg}$ call has not happened yet), then the key guess will set win.

Putting all of our bounds together gives

$$\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}cca}}_{\mathsf{U}^{\not\perp}[\mathsf{KEM},\mathsf{F}],\mathsf{S},\mathsf{P}\times\mathsf{P}_{\mathsf{rom}},\mathcal{A}_{\mathsf{cca}}}(\lambda) \leq 2\left(\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{F},\mathsf{S}_{\mathsf{F}},\mathsf{P}\times\mathsf{P}_{\mathsf{rom}},\mathcal{A}_{\mathsf{prf}}}(\lambda) + \mathsf{Adv}^{\mathsf{cguess^*\text{-}pcdca}}_{\mathsf{KEM},\mathsf{P},\mathcal{B}_{\mathsf{cguess^*}}}(\lambda)\right)$$
$$+ \mathsf{Adv}^{\mathsf{ow^*\text{-}pcdca}}_{\mathsf{KEM},\mathsf{P},\mathcal{B}_{\mathsf{ow}}}(\lambda) + \Pr[\mathrm{G}^{\mathsf{ow^*\text{-}pcdca}}_{\mathsf{KEM},\mathsf{P},\mathcal{B}_{\mathsf{ow}}}(\lambda) \text{ sets } \mathsf{coll} \text{ or } \mathsf{early}].$$

Combining this bound with Lemma C.2 and Lemma C.3 gives the desired relationship between the SIM*-AC-CCA security of $\mathsf{U}^{\not\perp}$ and the SIM*-AC-PRF security of F and the OW*-PCA security of KEM. ∎

D (Generalized) Pseudorandom Functions

In this section we provide the details of the ideas sketched in Section 6 for proving that a random oracle gives a good SIM*-AC-PRF secure function family. For generality, we will actually prove as more general pseudorandomness result that (up to a change of notation) also gives an alternate proof of Theorem E.2, which claims that ideal ciphers give SIM*-AC-SPRP secure blockciphers.

```
Game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}}}(\lambda)$
$k_{(\cdot)} \leftarrow\$ \mathsf{F.Kg}(1^\lambda)$
$g_{(\cdot)} \leftarrow\$ \mathcal{F}_\lambda$
$\sigma_{\mathsf{P}} \leftarrow\$ \mathsf{P.Init}(1^\lambda)$
$\sigma \leftarrow\$ \mathsf{S.Init}(1^\lambda)$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow\$ \mathcal{A}_{\mathsf{pr}}^{\mathrm{Ev},\mathrm{Exp},\mathrm{PPrim}}(1^\lambda)$
Return $(b = b')$

$\mathrm{PPrim}(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$y \leftarrow\$ \mathsf{P.Op}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})$
Return $y$

$\mathrm{Ev}(u, x)$
If $T_u[x] \neq \perp$ then return $T_u[x]$
If $b = 1$ then $y \leftarrow \mathsf{F.Ev}^{\mathsf{P}}(1^\lambda, k_u, x)$
If $b = 0$ then
    If $X_u$ then $y \leftarrow\$ \mathsf{S.Ev}^{\mathrm{PPrim}}(1^\lambda, u, x : \sigma)$
    Else $y \leftarrow g_u(x)$
$T_u[x] \leftarrow y$
Return $y$

$\mathrm{Exp}(u)$
If $b = 1$ then $k' \leftarrow k_u$
If $b = 0$ then $k' \leftarrow\$ \mathsf{S.Exp}^{\mathrm{PPrim}}(1^\lambda, u, T_u : \sigma)$
$X_u \leftarrow \mathsf{true}$
Return $k'$
```

Figure 21: Game defining SIM*-AC-PR[$\mathcal{F}$] security of function family F

Pseudorandom security. Let $\mathcal{F}_\lambda$ be a distribution over functions $g : \mathcal{D}_\lambda \to \mathcal{R}_\lambda$ and F be a function family. We will consider a SIM*-AC definition which asks whether oracle access to F with an unknown key looks like oracle access to a random function chosen according to g. Appropriate choices of $\mathcal{F}$ captures SIM*-AC-PRF, SIM*-AC-PRP, or SIM*-AC-SPRP security (up to minor notational changes). Then consider the game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}}}$ shown in Fig. 21. It works similarly to the SIM*-AC-PRF game, except that for notational simplicity we sample the entire random function g_u for user u all at once. Note it is still the case that the table T_u (and hence the view of S) only depends on the values of this function that correspond to the queries made by $\mathcal{A}_{\mathsf{pr}}$.

We define $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}}}(\lambda) = 2\Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}}}(\lambda)] - 1$ and say that F is SIM*-AC-PR[$\mathcal{F}$] secure with P if there exists a PPT S such that for all PPT $\mathcal{A}_{\mathsf{pr}}$, the advantage function $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}}}(\cdot)$ is negligible. We say that F is wSIM*-AC-PR[$\mathcal{F}$] secure with P if for all PPT $\mathcal{A}_{\mathsf{pr}}$ there exists a PPT S such that $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}}}(\cdot)$ is negligible.

Random keyed $\mathcal{F}$ functions are secure. For a given $\mathcal{F}$, let $\mathsf{P}_{\mathcal{F}}$ denote an efficient ideal primitive for which the distribution $\mathsf{P}.\mathcal{P}_\lambda$ samples functions f where the induced functions $f(k,\cdot)$ for all keys

$k \in \mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda$ are independently distributed according to $\mathcal{F}_\lambda$. From such a P, we can naturally construction a function family F as follows.

```
$\underline{\mathsf{F.Kg}(1^\lambda)}$
$k \leftarrow\!\!\$\ \mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda$
Return $k$

$\underline{\mathsf{F.Ev}^{\mathsf{P}_{\mathcal{F}}}(1^\lambda, k, x)}$
$y \leftarrow \mathsf{P}_{\mathcal{F}}(k, x)$
Return $y$
```

Generalizing the ideas from Section 6 we get the following result.

Theorem D.1 *Function family* F *is SIM*-AC-PR[$\mathcal{F}$] secure as long as* $|\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda|$ *is super-polynomial.*

Proof: We will define a simulator S for which we prove that

$$\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}}}(\lambda) \le \frac{u_\lambda(0.5u_\lambda + p_\lambda + p'_\lambda)}{|\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda|}$$

holds for all $\mathcal{A}_{\mathsf{pr}}$ making queries for at most u_λ distinct users, at most p_λ lazy sampling queries to P, and at most p'_λ programming queries to P. If $u_\lambda > |\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda|$ the bound holds vacuously, so assume this is not the case. Because $\mathcal{A}_{\mathsf{pr}}$ makes a polynomial number of queries and $|\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda|$ is super-polynomial, this bound is negligible.

The simulator S is as follows. It picks keys for exposed users uniformly at random and then programs the ideal primitive at that key to be consistent with the table T_u that it was given.

```
$\underline{\mathsf{S.Init}(1^\lambda)}$
$k_{(\cdot)} \leftarrow [\cdot]$
Return $k_{(\cdot)}$

$\underline{\mathsf{S.Ev}^{\text{PPRIM}}(1^\lambda, u, x : k_{(\cdot)})}$
$y \leftarrow \text{PPRIM}(\mathsf{Ls}, k_u, x, \diamond)$
Return $y$

$\underline{\mathsf{S.Exp}^{\text{PPRIM}}(1^\lambda, u, T_u : k_{(\cdot)})}$
If $k_u = \bot$ then
  $k_u \leftarrow\!\!\$\ \mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda$
  For $x$ s.t. $T_u[x] \neq \bot$ do
    $\text{PPRIM}(\mathsf{Prog}, k_u, x, T_u[x])$
Return $k_u$
```

We analyze this simulator by considering the games defined in Fig. 22. To simplify notation in these games, we assume that $\mathcal{A}_{\mathsf{pr}}$ only queries users with identifiers $u \in [u_\lambda] = \{1, \ldots, u_\lambda\}$. Intuitively, we use them to show that the only way to detect the simulation is if two

Of these games, we will prove the following bounds from which the bound claimed above follows.

1. $\Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}},1}(\lambda)] = \Pr[\mathrm{G}'_1(\lambda)]$
2. $\Pr[\mathrm{G}'_1(\lambda)] = \Pr[\mathrm{G}_1(\lambda)]$
3. $\Pr[\mathrm{G}_1(\lambda)] \le \Pr[\mathrm{G}_0(\lambda)] + u_\lambda(p_\lambda + p'_\lambda)/|\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda|$
4. $\Pr[\mathrm{G}_0(\lambda)] = \Pr[\mathrm{G}'_0(\lambda)]$
5. $\Pr[\mathrm{G}'_0(\lambda)] \le \Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}},0}(\lambda)] + 0.5u_\lambda^2/|\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda|$

Claims 1 and 5. We compare the games G'_b to the original security games $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}},b}$. First let us consider games G''_b defined identically to G'_b except that the highlighted code is removed. We claim these games are equivalent to the original security games. They were created by plugging the code of F and S into the original security games and then making some simplifying notational changes. Using the assumption that $u \in [u_\lambda]$ always holds, we sample all user keys explicitly at the beginning of the game. Rather than sampling an entire function g for each g_u for each u,

```
Game $G'_b(\lambda)$ for $b \in \{0,1\}$
For $u = 1, \ldots, u_\lambda$ do
    $k_u \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda$
    If $k_u \in \{ k_i : 1 \le i < u \}$ then
        $\mathsf{bad}_{\mathsf{coll}} \leftarrow \mathsf{true}$
        $k_u \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda \setminus \{ k_i : 1 \le i < u \}$
$\sigma_{g,(\cdot)} \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Init}(1^\lambda)$
$\sigma_{\mathsf{P},(\cdot)} \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Init}(1^\lambda)$
$b' \leftarrow\$ \mathcal{A}_{\mathsf{pr}}^{\mathrm{Ev},\mathrm{Exp},\mathrm{PPrim}}(1^\lambda)$
Return $(b' = 1) \vee \mathsf{bad}_{\mathsf{coll}}$
```

```
PPrim(Op, $k, x, y$)
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$y \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Op}(1^\lambda, k, x, y : \sigma_{\mathsf{P},k})$
Return $y$
```

```
Ev($u, x$)
If $T_u[x] \neq \bot$ then return $T_u[x]$
If $b = 1$ then $y \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Ls}(1^\lambda, k_u, x : \sigma_{\mathsf{P},k_u})$
If $b = 0$ then
    If $X_u$ then $y \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Ls}(1^\lambda, k_u, x : \sigma_{\mathsf{P},k_u})$
    Else $y \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Ls}(1^\lambda, k_u, x : \sigma_{g,u})$
$T_u[x] \leftarrow y$
Return $y$
```

```
Exp($u$)
If $b = 0$ then
    If $\neg X_u$ then
        For $x$ s.t. $T_u[x] \neq \bot$ do
            $\mathsf{P}_{\mathcal{F}}.\mathsf{Prog}(1^\lambda, k_u, x, T_u[x] : \sigma_{\mathsf{P},k_u})$
$X_u \leftarrow \mathsf{true}$
Return $k_u$
```

```
Game $G_0(\lambda)$, $G_1(\lambda)$
For $u = 1, \ldots, u_\lambda$ do
    $k_u \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda$
    If $k_u \in \{ k_i : 1 \le i < u \}$ then
        $\mathsf{bad}_{\mathsf{coll}} \leftarrow \mathsf{true}$
        $k_u \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda \setminus \{ k_i : 1 \le i < u \}$
$\sigma_{g,(\cdot)} \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Init}(1^\lambda)$
$\sigma_{\mathsf{P},(\cdot)} \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Init}(1^\lambda)$
$b' \leftarrow\$ \mathcal{A}_{\mathsf{pr}}^{\mathrm{Ev},\mathrm{Exp},\mathrm{PPrim}}(1^\lambda)$
Return $(b' = 1) \vee \mathsf{bad}_{\mathsf{coll}}$
```

```
PPrim(Op, $k, x, y$)
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$u \leftarrow \min\{ u : k_u = k \}$
If $u \neq \bot$ and $\neg X_u$ then
    $\mathsf{bad}_{\mathsf{guess}} \leftarrow \mathsf{true}$
    For $x'$ s.t. $\sigma_{g,u}[k, x'] \neq \bot$ do
        $\sigma_{\mathsf{P},k}[k, x'] \leftarrow \sigma_{g,u}[k, x']$
$y \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Op}(1^\lambda, k, x, y : \sigma_{\mathsf{P},k})$
Return $y$
```

```
Ev($u, x$)
If $T_u[x] \neq \bot$ then return $T_u[x]$
If $X_u$ then $y \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Ls}(1^\lambda, k_u, x : \sigma_{\mathsf{P},k_u})$
Else
    If $\sigma_{\mathsf{P},k_u} \neq [\cdot,\cdot]$ then
        $\mathsf{bad}_{\mathsf{guess}} \leftarrow \mathsf{true}$
        For $x'$ s.t. $\sigma_{g,u}[k, x'] \neq \bot$ do
            $\sigma_{g,u}[k_u, x'] \leftarrow \sigma_{\mathsf{P},k_u}[k_u, x']$
    $y \leftarrow\$ \mathsf{P}_{\mathcal{F}}.\mathsf{Ls}(1^\lambda, k_u, x : \sigma_{g,u})$
$T_u[x] \leftarrow y$
Return $y$
```

```
Exp($u$)
If $\neg X_u$ then
    For $x$ s.t. $T_u[x] \neq \bot$ do
        $\mathsf{P}_{\mathcal{F}}.\mathsf{Prog}(1^\lambda, k_u, x, T_u[x] : \sigma_{\mathsf{P},k_u})$
$X_u \leftarrow \mathsf{true}$
Return $k_u$
```

Figure 22: Games used to prove Theorem D.1, that random keys $\mathcal{F}$ functions are secure

we instead lazily sample the function using $\mathsf{P}_{\mathcal{F}}$ applied to the state $\sigma_{g,u}$. For the ideal primitive accessed through PPrim, rather than using a single σ_{P} we use separate $\sigma_{\mathsf{P},k}$ for each k (recall the functions for each k were anyway independent when using a single σ_{P}). Note that $\sigma_{g,u}$ is only ever accessed in the form $\sigma_{g,u}[k_u, \cdot]$ and $\sigma_{\mathsf{P},k}$ is only ever accessed in the form $\sigma_{\mathsf{P},k}[k, \cdot]$. The $k_u = \perp$ check in $\mathsf{S.Exp}$ was used to check if the user was previously exposed, so we replace this with $\neg X_u$. These changes do not modify the overall behavior of the game, so $\Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}pr}}_{\mathcal{F},\mathsf{F},\mathsf{S},\mathsf{P},\mathcal{A}_{\mathsf{pr}},b}(\lambda)] = \Pr[\mathrm{G}''_b(\lambda)]$.

Now consider adding back the highlighted code to obtain G'_b. This code only ever changes behavior after the flag $\mathsf{bad_{coll}}$ is set. It is set if there is a collision between the keys sampled by two different users. Then the key is resampled to ensure all user keys are distinct and furthermore the game will necessarily output true. Clearly, this can only increase the probability the game outputs true, so $\Pr[\mathrm{G}''_b(\lambda)] \leq \Pr[\mathrm{G}'_b(\lambda)]$ giving claim 1. For claim 5, we use that the games G''_b and G'_b are identical until the $\mathsf{bad_{coll}}$ flag is set, so $\Pr[\mathrm{G}'_b(\lambda)] \leq \Pr[\mathrm{G}''_b(\lambda)] + \Pr[\mathrm{G}''_b(\lambda) \text{ sets } \mathsf{bad_{coll}}] \leq \Pr[\mathrm{G}''_b(\lambda)] + \binom{u_\lambda}{2}/|\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda|$. The last term follows from a union bound over all pairs of users.

Claims 2 and 4. We wish to argue that the games G'_b are hard to distinguish from each other. Note that the map $u \to k_u$ is now an injection. In G'_1, a single state σ_{P,k_u} is used to respond to all queries associated with k_u whether they come through PPrim or Ev. In G'_0, there are two separated states σ_{P,k_u} and $\sigma_{g,u}$ that are used (for PPrim and Ev respectively) until an Exp query is made, at which point the game attempts to program σ_{P,k_u} to be consistent with the values chosen with $\sigma_{g,u}$. Henceforth, only σ_{P,k_u} is used. The games G_0 and G_1 rewrite these games to make them more similar so we can precisely exam the difference.

The easier to consider is G_0 which does not include the highlighted code. Comparing it to G'_0 we can see that it is basically line-for-line identical, except that in G_0 we have added if statements to PPrim and Ev which set the flag $\mathsf{bad_{guess}}$. Because the highlighted code is omitted, neither if statement modifies the behavior of the game, so $\Pr[\mathrm{G}_0(\lambda)] = \Pr[\mathrm{G}'_0(\lambda)]$ holds, giving claim 4.

The more interesting comparison is with G_1 which does include the highlighted code. Compared to G'_1, it now uses separate σ_{P,k_u} and $\sigma_{g,u}$ before exposures. For an unexposed user, if one of these tables is about to be used, we first copy into it any values that are currently stored in its "partner" table.[23] Consequently, at any point in time, the most recently accessed table associated with u and k_u will exactly match what the current value of σ_{P,k_u} would be in G'_1. The "partner" table will be defined consistently except by being $\perp$ in some entries where the more recently accessed table was defined. In the first Exp query for u, the table σ_{P,k_u} is updated to be what it would have been in G'_1 (note that $T_u[\cdot]$ exactly matches $\sigma_{g,u}[k_u, \cdot]$ at this time). Henceforth, only it is used (as in G'_1). So $\Pr[\mathrm{G}'_1(\lambda)] = \Pr[\mathrm{G}_1(\lambda)]$ holds, giving claim 2.

Claim 3. The point of rewriting into the games G_0 and G_1 was to make them more similar. In particular, they are identical until the $\mathsf{bad_{guess}}$ is set, so $\Pr[\mathrm{G}_1(\lambda)] \leq \Pr[\mathrm{G}_0(\lambda)] + \Pr[\mathrm{G}_0(\lambda) \text{ sets } \mathsf{bad_{guess}}]$.

In Ev, the flag is set if $\neg X_u$ and $\sigma_{\mathsf{P},k_u} \neq [\cdot, \cdot]$ hold. These condition require the attacker to have queried PPrim using the key k_u of an unexposed user. Similarly, the checks before setting the flag in PPrim check if this is a query for the key k_u of an unexposed user. For an unexposed user, the oracles Ev and PPrim never used k_u in a manner that effects the distributions of their output.[24] So the view of $\mathcal{A}_{\mathsf{pr}}$ only depends on unexposed users' keys from the fact that they are sample to be distinct from any other keys (some of which may have been exposed). Consider a PPrim made

[23]For the line $u \leftarrow \min\{\, u : k_u = k \,\}$ note that the set on the right has size either 0 or 1 because $u \to k_u$ is injective. This line of code sets u to be the user which has key k, unless none exist in which case $u = \perp$.

[24]Note, this is false in G_1. (Where the value of k_u effects the output in the event that the bad flag is set.)

with key k at a time where $\mathcal{K}_X$ is the set of keys corresponding to users that have been exposed so far. Then from the perspective of $\mathcal{A}_{\mathsf{pr}}$, the set of unexposed users' keys $\{k_1, \ldots, k_{u_\lambda}\} \setminus \mathcal{K}_X$ is uniformly (without repetition) distributed in $\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda \setminus \mathcal{K}_X$. Thus the probability this particular query uses the key of an unexposed user is bounded by

$$\Pr[k \in \{k_1, \ldots, k_{u_\lambda}\} \setminus \mathcal{K}_X] \le \frac{|\{k_1, \ldots, k_{u_\lambda}\} \setminus \mathcal{K}_X|}{|\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda \setminus \mathcal{K}_X|} = \frac{u_\lambda - |\mathcal{K}_X|}{|\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda| - |\mathcal{K}_X|} \le \frac{u_\lambda}{|\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda|}.^{25}$$

Applying a union bound over the $p_\lambda + p'_\lambda$) queries to PPRIM gives $\Pr[\mathrm{G}_0(\lambda)$ sets $\mathsf{bad}_{\mathsf{guess}}] \le u_\lambda(p_\lambda + p'_\lambda)/|\mathsf{P}_{\mathcal{F}}.\mathcal{K}_\lambda|$ to establish claim 3. ▮

E Pseudorandom Permutations

In this section we give SIM*-AC security definitions for pseudorandom permutations and show the expected result that pseudorandom permutation security implies pseudorandom function security (up to the birthday bound).

```
Game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prp}}_{\mathsf{B,S,P},\mathcal{A}_{\mathsf{prp}}}(\lambda)$
$k_{(\cdot)} \leftarrow_\$ \mathsf{B.Kg}(1^\lambda)$
$\sigma_{\mathsf{P}} \leftarrow_\$ \mathsf{P.Init}(1^\lambda)$
$\sigma \leftarrow_\$ \mathsf{S.Init}(1^\lambda)$
$b \leftarrow_\$ \{0,1\}$
$b' \leftarrow_\$ \mathcal{A}_{\mathsf{prp}}^{\mathrm{Ev,Inv,Exp,PPrim}}(1^\lambda)$
Return $(b = b')$

PPrim$(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$y \leftarrow_\$ \mathsf{P.Op}(1^\lambda, k, x, y : \sigma_{\mathsf{P}})$
Return $y$

Exp$(u)$
If $b = 1$ then $k' \leftarrow k_u$
If $b = 0$ then $k' \leftarrow_\$ \mathsf{S.Exp}^{\mathrm{PPrim}}(1^\lambda, u, T_u : \sigma)$
$X_u \leftarrow \mathsf{true}$
Return $k'$

Ev$(u, x)$
If $b = 1$ then $y \leftarrow \mathsf{B.Ev}^{\mathsf{P}}(1^\lambda, k_u, x)$
If $b = 0$ then
    If $X_u$ then $y \leftarrow_\$ \mathsf{S.Ev}^{\mathrm{PPrim}}(1^\lambda, u, x : \sigma)$
    Else
        If $T_u[x] = \bot$ then $y \leftarrow_\$ \mathsf{B.Out}(\lambda) \setminus \{\, y : T_u^{-1}[y] \ne \bot \,\}$
        Else $y \leftarrow T_u[x]$
$T_u[x] \leftarrow y\,;\, T_u^{-1}[y] \leftarrow x$
Return $y$

Inv$(u, y)$
If $b = 1$ then $x \leftarrow \mathsf{B.Inv}^{\mathsf{P}}(1^\lambda, k_u, y)$
If $b = 0$ then
    If $X_u$ then $x \leftarrow_\$ \mathsf{S.Inv}^{\mathrm{PPrim}}(1^\lambda, u, y : \sigma)$
    Else
        If $T_u^{-1}[y] = \bot$ then $x \leftarrow_\$ \mathsf{B.Out}(\lambda) \setminus \{\, x : T_u[x] \ne \bot \,\}$
        Else $x \leftarrow T_u^{-1}[y]$
$T_u[x] \leftarrow y\,;\, T_u^{-1}[y] \leftarrow x$
Return $x$
```

Figure 23: Game defining SIM*-AC-PRP and SIM*-AC-SPRP security of blockcipher B

Pseudorandom permutation security. Let B be a blockcipher (a function family for which $\mathsf{B.Ev}(1^\lambda, k, \cdot)$ is a permutation with efficient inverse $\mathsf{B.Inv}(1^\lambda, k, \cdot)$). Our pseudorandom permutation (PRP) security definition is captured by the game $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prp}}_{\mathsf{B,S,P},\mathcal{A}_{\mathsf{prp}}}$ shown in Fig. 23. It differs from $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{B,S,P},\mathcal{A}_{\mathsf{prp}}}$ in that random responses to a particular user are sampled without repetition in EV and the attacker has access to an inverse oracle INV which is analogous to EV.

We define $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prp}}_{\mathsf{B,S,P},\mathcal{A}_{\mathsf{prp}}}(\lambda) = 2\Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prp}}_{\mathsf{B,S,P},\mathcal{A}_{\mathsf{prp}}}(\lambda)] - 1$ and say that B is SIM*-AC-PRP secure with P if there exists a PPT S such that for all PPT $\mathcal{A}_{\mathsf{prp}}$ *that never query* INV, the advantage function $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prp}}_{\mathsf{B,S,P},\mathcal{A}_{\mathsf{prp}}}(\cdot)$ is negligible. We say B is wSIM*-AC-PRP secure with P if for all PPT $\mathcal{A}_{\mathsf{prp}}$ there exists a PPT S such that $\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prp}}_{\mathsf{B,S,P},\mathcal{A}_{\mathsf{prp}}}(\cdot)$ is negligible.

[25] Here we use that $(x-n)/(y-n) \le x/y$ whenever $y \ge x$ and $y > n \ge 0$.

Strong pseudorandom function security (SIM*-AC-SPRP/wSIM*-AC-SPRP) is defined as above except the adversary is not required to ignore its Inv oracle.

PRPs are good PRFs. A classic result tells us that PRP security implies PRF security. By the same reasoning, this holds with SIM*-AC variants of these definitions.

Lemma E.1 *For $x \in \{\varepsilon, \mathrm{w}\}$, xSIM*-AC-PRP security of* B *and xSIM*-AC-PRF security of* B *are equivalent if* $|\mathsf{B.Inp}(\cdot)|$ *is super-polynomial.*

Proof: The games $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prp}}$ and $\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prf}}$ differ only in the existence of Inv (which is never queried) and whether the outputs of Ev for unexposed users are uniformly random with or without replacement when $b = 0$. Hence if q_λ is the number of oracles made by an adversary $\mathcal{A}$, we have that $|\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prp}}_{\mathsf{B,S,P},\mathcal{A}}(\lambda) - \mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prf}}_{\mathsf{B,S,P},\mathcal{A}}(\lambda)| \leq 0.5q_\lambda^2/|\mathsf{B.Inp}(\lambda)|$.∎

Ideal ciphers are Strong PRPs. Fix an ideal cipher $\mathsf{P_{icm}}$ and define blockcipher B as follows, expecting access to $\mathsf{P_{icm}}$.

```
$\underline{\mathsf{B.Kg}(1^\lambda)}$
$k \leftarrow\$ \mathsf{P_{icm}}.\mathcal{K}_\lambda$
Return $k$

$\underline{\mathsf{B.Ev}^{\mathsf{P_{icm}}}(1^\lambda, k, x)}$
$y \leftarrow \mathsf{P_{icm}}(k, (+, x))$
Return $y$

$\underline{\mathsf{B.Inv}^{\mathsf{P_{icm}}}(1^\lambda, k, y)}$
$x \leftarrow \mathsf{P_{icm}}(k, (-, y))$
Return $x$
```

Using ideas from Section 6 (namely by tweaking the analysis Jaeger and Tyagi [JT20] did for ideal ciphers being SIM-AC-PRF secure and then accounting as well for programming queries) we get the following result. Below we write a direct proof of this. It also can be shown to follow from Theorem D.1.

Theorem E.2 *Blockcipher* B *is SIM*-AC-SPRP secure as long as* $|\mathsf{P_{icm}}.\mathcal{K}_\lambda|$ *is super-polynomial.*

Proof: We will define a simulator S for which we prove that

$$\mathsf{Adv}^{\mathsf{sim^*\text{-}ac\text{-}prp}}_{\mathsf{B,S,P_{icm}},\mathcal{A}_{\mathsf{prp}}}(\lambda) \leq \frac{u_\lambda(u_\lambda + p_\lambda + p'_\lambda)}{|\mathsf{P_{icm}}.\mathcal{K}_\lambda|}$$

holds for all $\mathcal{A}_{\mathsf{prp}}$ making queries for at most u_λ distinct users, at most p_λ lazy sampling queries to $\mathsf{P_{icm}}$, and at most p'_λ programming queries to $\mathsf{P_{icm}}$. If $u_\lambda > |\mathsf{P_{icm}}.\mathcal{K}_\lambda|$ the bound holds vacuously, so assume this is not the case. Assuming that $\mathcal{A}_{\mathsf{prp}}$ makes a polynomial number of queries and that $|\mathsf{P_{icm}}.\mathcal{K}_\lambda|$ is super-polynomial gives us that this bound is negligible, as desired.

The simulator S is as follows. It picks keys for exposed users uniformly at random and then programs the ideal cipher at that key to be consistent with the table T_u that it was given.

```
$\underline{\mathsf{S.Init}(1^\lambda)}$
$k_{(\cdot)} \leftarrow [\cdot]$
Return $k_{(\cdot)}$

$\underline{\mathsf{S.Ev}^{\mathrm{PPRIM}}(1^\lambda, u, x : k_{(\cdot)})}$
$y \leftarrow \mathrm{PPRIM}(\mathsf{Ls}, k_u, (+, x), \diamond)$
Return $y$

$\underline{\mathsf{S.Inv}^{\mathrm{PPRIM}}(1^\lambda, u, y : k_{(\cdot)})}$
$y \leftarrow \mathrm{PPRIM}(\mathsf{Ls}, k_u, (-, y), \diamond)$
Return $y$

$\underline{\mathsf{S.Exp}^{\mathrm{PPRIM}}(1^\lambda, u, T_u : k_{(\cdot)})}$
If $k_u = \bot$ then
    $k_u \leftarrow\$ \mathsf{P_{icm}}.\mathcal{K}_\lambda$
    For $x$ s.t. $T_u[x] \neq \bot$ do
        $\mathrm{PPRIM}(\mathsf{Prog}, k_u, (+, x), T_u[x])$
Return $k_u$
```

```
Games $G_1(\lambda)$, $G_2(\lambda)$
For $u = 1, \ldots, u_\lambda$ do
    $k_u \leftarrow\!\!\$\ \mathsf{P_{icm}}.\mathcal{K}_\lambda \setminus \{\, k_i \,:\, 1 \le i < u \,\}$
$b' \leftarrow\!\!\$\ \mathcal{A}_{\mathsf{prp}}^{\text{Ev,Inv,Exp,PPrim}}(1^\lambda)$
Return $(b' = 1)$

PPrim$(\mathsf{Op}, k, x, y)$
Require $\mathsf{Op} \in \{\mathsf{Ls}, \mathsf{Prog}\}$
$(\circ, x) \leftarrow x$; $u \leftarrow \min\{\, u \,:\, k_u = k \,\}$
If $u \neq \bot$ and $\neg X_u$ then
    $\mathsf{bad} \leftarrow \mathsf{true}$
    For $x'$ s.t. $T_u^+[x'] \neq \bot$ do
        $\sigma^+[k, x'] \leftarrow T_u^+[x']$
If $\mathsf{Op} = \mathsf{Prog}$ then
    If $\sigma^\circ[k, x] = \bot$ and $\sigma^{\bar\circ}[k, y] = \bot$ then
        $\sigma^\circ[k, x] \leftarrow y$
    Return $\diamond$
If $\sigma^\circ[k, x] = \bot$ then
    $\sigma^\circ[k, x] \leftarrow\!\!\$\ \mathsf{B.Out}(\lambda) \setminus \{\, y \,:\, \sigma^{\bar\circ}[k, y] \neq \bot \,\}$
Return $\sigma^\circ[k, x]$

Exp$(u)$
If $\neg X_u$ then
    For $x$ s.t. $T_u^+[x] \neq \bot$ do
        If $\sigma^+[k, x] = \bot$ and $\sigma^-[k, T_u^+[x]] = \bot$ then
            $\sigma^+[k, x] \leftarrow T_u^+[x]$
$X_u \leftarrow \mathsf{true}$
Return $k_u$
```

```
Ev$(u, x)$
If $X_u$ then return PPrim$(\mathsf{Ls}, k_u, (+, x), \diamond)$
If $\sigma^+[k_u, \cdot] \neq [\cdot]$ then
    $\mathsf{bad}' \leftarrow \mathsf{true}$
    For $x'$ s.t. $\sigma^+[k_u, x'] \neq \bot$ do
        $T_u^+[x'] \leftarrow \sigma^+[k_u, x']$
If $T_u^+[x] = \bot$ then
    $y \leftarrow\!\!\$\ \mathsf{B.Out}(\lambda) \setminus \{\, y \,:\, T_u^-[y] \neq \bot \,\}$
Else
    $y \leftarrow T_u^+[x]$
$T_u^+[x] \leftarrow y$
Return $y$

Inv$(u, y)$
If $X_u$ then return PPrim$(\mathsf{Ls}, k_u, (-, y), \diamond)$
If $\sigma^-[k_u, \cdot] \neq [\cdot]$ then
    $\mathsf{bad}' \leftarrow \mathsf{true}$
    For $y'$ s.t. $\sigma^-[k_u, y'] \neq \bot$ do
        $T_u^-[y'] \leftarrow \sigma^+[k_u, y']$
If $T_u^-[y] = \bot$ then
    $x \leftarrow\!\!\$\ \mathsf{B.Out}(\lambda) \setminus \{\, x \,:\, T_u^+[x] \neq \bot \,\}$
Else
    $x \leftarrow T_u^-[y]$
$T_u^-[y] \leftarrow x$
Return $x$
```

Figure 24: Games used to prove SIM*-AC-SPRP security of ideal cipher

We analyze the success of this simulator via the pair of games shown in Fig. 24. To simplify notation in these games, we assume that $\mathcal{A}_{\mathsf{prp}}$ only queries users with identifiers $u \in [u_\lambda] = \{1, \ldots, u_\lambda\}$. We additionally define tables σ^+ and T^+ so that for all k the map $x \mapsto \sigma^+[k, x]$ has no collisions and for all u the map $x \mapsto T_u^+[x]$ has no collisions. Then we define $\sigma^-[k, \cdot]$ and $T_u^-[\cdot]$ to have the inverse map. Updating σ^+ or T^+ should be interpreted as also updating the corresponding σ^- or T^- and vice versa. If $\circ \in \{+, -\}$, we let $\bar\circ$ denote the element contained in $\{+, -\} \setminus \{\circ\}$.

Of these games, we will prove the following bounds from which the bound claimed above follows.

1. $\Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prp}}_{\mathsf{B,S,P},\mathcal{A}_{\mathsf{prp}},1}(\lambda)] \le \Pr[\mathrm{G}_1(\lambda)] + \binom{u_\lambda}{2}/|\mathsf{P_{icm}}.\mathcal{K}_\lambda|$
2. $\Pr[\mathrm{G}_1(\lambda)] \le \Pr[\mathrm{G}_2(\lambda)] + u_\lambda(p_\lambda + p'_\lambda)/|\mathsf{P_{icm}}.\mathcal{K}_\lambda|$
3. $\Pr[\mathrm{G}_2(\lambda)] = \Pr[\mathrm{G}^{\mathsf{sim^*\text{-}ac\text{-}prp}}_{\mathsf{B,S,P},\mathcal{A}_{\mathsf{prp}},0}(\lambda)] + \binom{u_\lambda}{2}/|\mathsf{P_{icm}}.\mathcal{K}_\lambda|$

Claim 1. We will establish the claimed bound by arguing that $\mathrm{G}_1(\lambda)$ is completely equivalent to the real SIM*-AC-SPRP world except for the fact that the user keys are sampled without replacement. The statistical distance between sampling with or without replacement can be bounded by $\binom{u_\lambda}{2}/|\mathsf{P_{icm}}.\mathcal{K}_\lambda|$.

Note that $\mathrm{G}_1(\lambda)$ includes all of the highlighted code. For unexposed users, the game stores a table T_u^+ for each user which is used for the lazy sampling of the permutation returned by Ev and Inv.

In the real SIM*-AC-SPRP world, the oracles would instead have called $\textsc{PPrim}$. To match this behavior, game $\mathrm{G}_1(\lambda)$ maintains consistency between T_u^+ and $\sigma^+[k_u, \cdot]$. Whenever a $\textsc{PPrim}$ query is made for an unexposed k_u, we first copy values from T_u^+ into $\sigma^+[k_u, \cdot]$. For $\textsc{Ev}$ and $\textsc{Inv}$ queries, we copy in the opposite direction. At any point in time, T_u^+ may contain values not stored in $\sigma^+[k_u, \cdot]$ or vice versa. However both cannot be the case, so the way new entries are added maintains the fact that these are supposed to define a permutation. Note that distinct users cannot share a key, so we do not need to maintain any consistency between the T tables for different users. When a user is exposed for the first time, we copy values from T_u^+ into $\sigma^+[k_u, \cdot]$ and will henceforth never again user T_u. This copying checks that the values being added to σ are consistent with it defining a permutation. These checks are unnecessary in this game, but will matter when we transition to the next game.

Claim 2. Now we compare the game $\mathrm{G}_2(\lambda)$ to the game $\mathrm{G}_1(\lambda)$. They differ only in that $\mathrm{G}_2(\lambda)$ omits the highlighted code which happens after one of the bad flags is set. The flag bad is set if the adversary ever makes a $\textsc{PPrim}$ query with a key k_u belonging to an unexposed user u. The flag bad' in $\textsc{Ev}$ or $\textsc{Inv}$ is set only when they are queries for an unexposed user u such that k_u has previously been queried to $\textsc{PPrim}$ (filling corresponding entries of σ). Note then that bad' can only occur if bad has already occurred, giving us that $\Pr[\mathrm{G}_1(\lambda)] \leq \Pr[\mathrm{G}_2(\lambda)] + \Pr[\mathrm{G}_2(\lambda) \text{ sets } \mathsf{bad}]$. Consider a particular point in time when $\mathcal{A}_{\mathsf{prp}}$ queries $\textsc{PPrim}(\mathsf{Op}, k, x, y)$. If k is equal to a previously exposed user key, then this query will definitely not set the flag. Otherwise, it sets the flag if k is included in the set $\{\, k_u : \neg X_u \,\}$ which is a uniformly random subset of $\mathsf{P}_{\mathsf{icm}}.\mathcal{K}_\lambda \setminus \{\, k_u : X_u \,\}$ of size $u_\lambda - n$ where n denotes the number of currently exposed users. Hence the probability that this query sets the bad flag is bounded by $|\{\, k_u : \neg X_u \,\}|/|\mathsf{P}_{\mathsf{icm}}.\mathcal{K}_\lambda \setminus \{\, k_u : X_u \,\}| \leq (u_\lambda - n)/(|\mathsf{P}_{\mathsf{icm}}.\mathcal{K}_\lambda| - n) \leq u_\lambda/|\mathsf{P}_{\mathsf{icm}}.\mathcal{K}_\lambda|$.[26] Applying a union bound over the $p_\lambda + p'_\lambda$ queries to $\textsc{PPrim}$ gives the claimed bound.

Claim 3. For claim 3, we argue that the game $\mathrm{G}_2(\lambda)$ is equivalent to the ideal SIM*-AC-SPRP world with simulator S, except for the fact that the keys are sampled without repetition. Note that (for unexposed users) the table T_u is now used by $\textsc{Ev}$ and $\textsc{Inv}$ to lazily sample a permutation which is completely separate from σ. The oracle $\textsc{PPrim}$ honestly uses σ as the state for an ideal cipher. On the exposure of a user u, $\sigma^+[k_u, \cdot]$ is reprogrammed to match T_u *anywhere that the two are not inconsistent*. Unlike in $\mathrm{G}_1(\lambda)$, such inconsistencies are possible. This matches what would be achieved by S's programming queries on exposures. After exposure, $\textsc{Ev}$ and $\textsc{Inv}$ honestly call $\textsc{PPrim}$ as S would. Hence, the claim follows because the statistical distance between sampling with or without replacement can be bounded by $\binom{u_\lambda}{2}/|\mathsf{P}_{\mathsf{icm}}.\mathcal{K}_\lambda|$.[27] ∎

[26]Here we use that $(x - n)/(y - n) \leq x/y$ whenever $y \geq x$ and $y > n \geq 0$.

[27]Tweaking the proof would allow us to avoiding adding this term twice. See $\mathsf{bad}_{\mathsf{coll}}$ in the proof of Theorem D.1. Alternatively, we could define the simulator to pick keys without replacement.