

Security Proofs in the Style of *The Joy of Cryptography*

Mike Rosulek

January 4, 2025

Overview: This document contains various security proofs excerpted from (the forthcoming new edition of) *The Joy of Cryptography* [Ros]. In the textbook I take a unified, game-based approach to provable security. Thus, I hope that the examples are understandable to cryptographers who are comfortable with the game-based styles of Shoup [Sho04] and Bellare & Rogaway [BR06]. However, beware that there are some important differences in my approach, the most important of which is the following:

Every security definition is phrased as the indistinguishability of two games (which I call *libraries*). It is important that there are 2 *separate* games, each written explicitly; not a single game where the adversary’s goal is to guess a secret bit. Even “unforgeability”-style definitions are written in this way.

This principle has two important consequences:

- The goal of a security proof is to show that two particular games, say, \mathcal{L}_1 and \mathcal{L}_2 , are indistinguishable. Starting from the code \mathcal{L}_1 , a security proof consists of a sequence of small modifications to its code, eventually culminating in the code of \mathcal{L}_2 , where each step is justified to have only negligible effect on an adversary.
- When a security proof uses the security of an underlying primitive (e.g., when using a secure PRF to construct a CPA-secure encryption), there is no need to switch into the contrapositive mindset (e.g., if \mathcal{A} breaks CPA security, then there is an \mathcal{A}' that breaks the PRF). This fact and its merit are best illustrated by example, in the following sections.

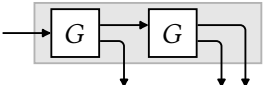
I write $\mathcal{L}_1 \approx \mathcal{L}_2$ to mean that two libraries are indistinguishable (no polynomial-time adversary can distinguish with better than negligible advantage), and $\mathcal{L}_1 \equiv \mathcal{L}_2$ to mean that the two have identical external behavior (perfectly indistinguishable).¹ Other conventions are introduced below, as needed.

1 PRG Length Extension

Claim 1. If G is a secure length-doubling PRG, then the following function H is also a secure PRG:

$H(S)$:

$A\|B := G(S)$
 $C\|D := G(B)$
return $A\|C\|D$



Each of the variables A, B, C, D are λ bits long. Thus, “ $A\|B := G(S)$ ” means: assign the first λ bits of $G(S)$ to A and the other λ bits to B .

In other words,

¹The interpretation of $\mathcal{L}_1 \approx \mathcal{L}_2$ is asymptotic, but is straightforward to adapt these proofs to a concrete style by accounting for the advantage that accumulates in the sequence of games.

$$\begin{array}{c}
\text{if} \quad \boxed{\begin{array}{c} \mathcal{L}_{\text{prg-real}}^G \\ \hline \text{PRG.SAMPLE}_G(): \\ S \leftarrow \{0, 1\}^\lambda \\ \text{return } G(S) \end{array}} \approx \boxed{\begin{array}{c} \mathcal{L}_{\text{prg-rand}}^G \\ \hline \text{PRG.SAMPLE}_G(): \\ Y \leftarrow \{0, 1\}^{2\lambda} \\ \text{return } Y \end{array}} \quad \text{then} \quad \boxed{\begin{array}{c} \mathcal{L}_{\text{prg-real}}^H \\ \hline \text{PRG.SAMPLE}_H(): \\ S \leftarrow \{0, 1\}^\lambda \\ // H(S): \\ A||B := G(S) \\ C||D := G(B) \\ \text{return } A||C||D \end{array}} \approx \boxed{\begin{array}{c} \mathcal{L}_{\text{prg-rand}}^H \\ \hline \text{PRG.SAMPLE}_H(): \\ Y \leftarrow \{0, 1\}^{3\lambda} \\ \text{return } Y \end{array}}
\end{array}$$

Proof. The starting point is $\mathcal{L}_{\text{prg-real}}^H$. Factoring out some statements into a separate subroutine has no effect on the adversary. Here we use \diamond to denote the natural composition of two programs/games. It is no coincidence that what is factored out is exactly an instance of $\mathcal{L}_{\text{prg-real}}^G$.

$$\boxed{\begin{array}{c} \mathcal{L}_{\text{prg-real}}^H \\ \hline \text{PRG.SAMPLE}_H(): \\ S \leftarrow \{0, 1\}^\lambda \\ // H(S): \\ A||B := G(S) \\ C||D := G(B) \\ \text{return } A||C||D \end{array}} \equiv \boxed{\begin{array}{c} \text{PRG.SAMPLE}_H(): \\ \\ A||B := \text{PRG.SAMPLE}_G() \\ C||D := G(B) \\ \text{return } A||C||D \end{array}} \diamond \boxed{\begin{array}{c} \mathcal{L}_{\text{prg-real}}^G \\ \hline \text{PRG.SAMPLE}_G(): \\ S \leftarrow \{0, 1\}^\lambda \\ \text{return } G(S) \end{array}}$$

Since G is a secure PRG, replacing $\mathcal{L}_{\text{prg-real}}^G$ with $\mathcal{L}_{\text{prg-rand}}^G$ has negligible effect on the adversary.

$$\boxed{\text{PRG.SAMPLE}_H(): // \dots} \diamond \boxed{\begin{array}{c} \mathcal{L}_{\text{prg-real}}^G \\ \hline \text{PRG.SAMPLE}_G(): \\ S \leftarrow \{0, 1\}^\lambda \\ \text{return } G(S) \end{array}} \approx \boxed{\text{PRG.SAMPLE}_H(): // \dots} \diamond \boxed{\begin{array}{c} \mathcal{L}_{\text{prg-rand}}^G \\ \hline \text{PRG.SAMPLE}_G(): \\ Y \leftarrow \{0, 1\}^{2\lambda} \\ \text{return } Y \end{array}}$$

Inlining a subroutine has no effect on the adversary.

$$\boxed{\begin{array}{c} \text{PRG.SAMPLE}_H(): \\ A||B := \text{PRG.SAMPLE}_G() \\ C||D := G(B) \\ \text{return } A||C||D \end{array}} \diamond \boxed{\begin{array}{c} \mathcal{L}_{\text{prg-rand}}^G \\ \hline \text{PRG.SAMPLE}_G(): \\ Y \leftarrow \{0, 1\}^{2\lambda} \\ \text{return } Y \end{array}} \equiv \boxed{\begin{array}{c} \text{PRG.SAMPLE}_H(): \\ A||B \leftarrow \{0, 1\}^{2\lambda} \\ C||D := G(B) \\ \text{return } A||C||D \end{array}}$$

Uniformly sampling 2λ bits is the same as uniformly (and independently) sampling its two halves.

$$\boxed{\begin{array}{c} \text{PRG.SAMPLE}_H(): \\ A||B \leftarrow \{0, 1\}^{2\lambda} \\ \\ C||D := G(B) \\ \text{return } A||C||D \end{array}} \equiv \boxed{\begin{array}{c} \text{PRG.SAMPLE}_H(): \\ A \leftarrow \{0, 1\}^\lambda \\ B \leftarrow \{0, 1\}^\lambda \\ C||D := G(B) \\ \text{return } A||C||D \end{array}}$$

Now a similar series of changes can be applied to the remaining call to G .

$$\boxed{\begin{array}{c} \text{PRG.SAMPLE}_H(): \\ A \leftarrow \{0, 1\}^\lambda \\ B \leftarrow \{0, 1\}^\lambda \\ C||D := G(B) \\ \text{return } A||C||D \end{array}} \equiv \boxed{\begin{array}{c} \text{PRG.SAMPLE}_H(): \\ A \leftarrow \{0, 1\}^\lambda \\ \\ C||D := \text{PRG.SAMPLE}_G() \\ \text{return } A||C||D \end{array}} \diamond \boxed{\begin{array}{c} \mathcal{L}_{\text{prg-real}}^G \\ \hline \text{PRG.SAMPLE}_G(): \\ S \leftarrow \{0, 1\}^\lambda \\ \text{return } G(S) \end{array}}$$

$$\begin{array}{c}
\approx \frac{\text{PRG.SAMPLE}_H(): // \dots}{\text{PRG.SAMPLE}_G():} \diamond \frac{\mathcal{L}_{\text{prg-rand}}^G}{Y \leftarrow \{0, 1\}^{2\lambda} \text{ return } Y} \\
\\
\equiv \frac{\text{PRG.SAMPLE}_H():}{A \leftarrow \{0, 1\}^\lambda \\ C \parallel D \leftarrow \{0, 1\}^{2\lambda} \\ \text{return } A \parallel C \parallel D}
\end{array}$$

Concatenating λ uniformly sampled bits with 2λ *independent*, uniformly sampled bits is the same as sampling 3λ uniform bits. The result of this change is $\mathcal{L}_{\text{prg-rand}}^H$, which completes the proof.

$$\begin{array}{c}
\frac{\text{PRG.SAMPLE}_H():}{A \leftarrow \{0, 1\}^\lambda \\ C \parallel D \leftarrow \{0, 1\}^{2\lambda} \\ \text{return } A \parallel C \parallel D} \equiv \frac{\mathcal{L}_{\text{prg-rand}}^H}{\text{PRG.SAMPLE}_H():} \\
\\
Y \leftarrow \{0, 1\}^{3\lambda} \\ \text{return } Y
\end{array}$$

□

1.1 Discussion

Concept: The 3-Hop Maneuver Idiom The following standard sequence of 3 steps appears in essentially every security proof in my methodology:

1. **Factor Out:** Factor out some lines of code from the current game into a separate sub-game. This changes only the internal organization of the game, so it has no effect on the adversary (calling program). Call the new sub-game \mathcal{L}_1 .
2. **Swap:** Swap \mathcal{L}_1 with a different library \mathcal{L}_2 . Thus, $\mathcal{L}_1 \approx \mathcal{L}_2$ must be an agreed-upon assumption or a lemma already proven.
3. **Inline:** Inline \mathcal{L}_2 into the main game. Again, this change has no effect on the adversary (calling program).

I call this standard idiom the **3-hop maneuver**. The preceding proof uses it twice, with $(\mathcal{L}_1, \mathcal{L}_2) = (\mathcal{L}_{\text{prg-real}}^G, \mathcal{L}_{\text{prg-rand}}^G)$.

The 3-hop maneuver is how *reductions* appear in my methodology. Suppose we want to argue that hybrids $\mathcal{L}_{\text{hyb-}i}$ and $\mathcal{L}_{\text{hyb-}(i+1)}$ are indistinguishable, because of some assumption P .

- **Traditional, contrapositive, reduction-centered approach:** given any \mathcal{A} that distinguishes $\mathcal{L}_{\text{hyb-}i}$ from $\mathcal{L}_{\text{hyb-}(i+1)}$, construct an \mathcal{A}' that breaks P .
- **Direct-implication approach focused on game-rewriting:** Assumption P is expressed as the indistinguishability of two libraries, say, \mathcal{L}_{P-1} and \mathcal{L}_{P-2} . Thus, we directly prove that

$$\mathcal{L}_{\text{hyb-}i} \equiv \mathcal{R} \diamond \mathcal{L}_{P-1} \approx \mathcal{R} \diamond \mathcal{L}_{P-2} \equiv \mathcal{L}_{\text{hyb-}(i+1)}$$

Of course, the reduction still exists, even if it is not the center of attention. If \mathcal{A} distinguishes $\mathcal{L}_{\text{hyb-}i}$ from $\mathcal{L}_{\text{hyb-}(i+1)}$, then $\mathcal{R} \diamond \mathcal{A}$ breaks P .

2 PRF Cascade

Claim 2. If F is a secure PRF with input/output length λ , then the following function H (also with input/output length λ) is also a secure PRF:

$$\begin{array}{l} H(K_1 \| K_2, X): \\ \quad Y := F(K_1, X) \\ \quad \text{return } F(K_2, Y) \end{array}$$

In other words,

$$\text{if } \begin{array}{|c|} \hline \mathcal{L}_{\text{prf-real}}^F \\ \hline K \leftarrow \{0, 1\}^\lambda \\ \hline \text{PRF.QUERY}_F(X): \\ \quad \text{return } F(K, X) \\ \hline \end{array} \cong \begin{array}{|c|} \hline \mathcal{L}_{\text{prf-rand}}^F \\ \hline \text{PRF.QUERY}_F(X): \\ \quad \text{if } L[X] \text{ undefined:} \\ \quad \quad L[X] \leftarrow \{0, 1\}^\lambda \\ \quad \text{return } L[X] \\ \hline \end{array} \quad \text{then} \quad \begin{array}{|c|} \hline \mathcal{L}_{\text{prf-real}}^H \\ \hline K_1 \| K_2 \leftarrow \{0, 1\}^{2\lambda} \\ \hline \text{PRF.QUERY}_H(X): \\ \quad // \text{return } H(K_1 \| K_2, X) \\ \quad Y := F(K_1, X) \\ \quad \text{return } F(K_2, Y) \\ \hline \end{array} \cong \begin{array}{|c|} \hline \mathcal{L}_{\text{prf-rand}}^H \\ \hline \text{PRF.QUERY}_H(X): \\ \quad \text{if } L[X] \text{ undefined:} \\ \quad \quad L[X] \leftarrow \{0, 1\}^\lambda \\ \quad \text{return } L[X] \\ \hline \end{array} .$$

Convention: In these libraries, K, K_1, K_2 are initialized at the beginning of time and are global/static to their library. Similarly, $L[\cdot]$ is global/static dictionary (associative array) data structure, and initially undefined everywhere.

Proof. The starting point is $\mathcal{L}_{\text{prf-real}}^H$: We first separate K_1 from K_2 , since they will be treated separately during the proof. We can also add a persistent/global dictionary $L^*[\cdot]$ to cache past answers of $\text{PRF.QUERY}_H(\cdot)$, so they are not recomputed twice.

$$\begin{array}{|c|} \hline \mathcal{L}_{\text{prf-real}}^H \\ \hline K_1 \| K_2 \leftarrow \{0, 1\}^{2\lambda} \\ \hline \text{PRF.QUERY}_H(X): \\ \quad Y := F(K_1, X) \\ \quad \text{return } F(K_2, Y) \\ \hline \end{array} \equiv \begin{array}{|c|} \hline K_1 \leftarrow \{0, 1\}^\lambda \\ K_2 \leftarrow \{0, 1\}^\lambda \\ \hline \text{PRF.QUERY}_H(X): \\ \quad \text{if } L^*[X] \text{ undefined:} \\ \quad \quad Y := F(K_1, X) \\ \quad \quad L^*[X] := F(K_2, Y) \\ \quad \text{return } L^*[X] \\ \hline \end{array}$$

We can apply the PRF security of F in a 3-hop maneuver, replacing $F(K_1, \cdot)$ with a **lazy random dictionary** (see below), which we name $L_1[\cdot]$.

$$\begin{array}{|c|} \hline K_1 \leftarrow \{0, 1\}^\lambda \\ K_2 \leftarrow \{0, 1\}^\lambda \\ \hline \text{PRF.QUERY}_H(X): \\ \quad \text{if } L^*[X] \text{ undefined:} \\ \quad \quad Y := F(K_1, X) \\ \quad \quad L^*[X] := F(K_2, Y) \\ \quad \text{return } L^*[X] \\ \hline \end{array} \equiv \begin{array}{|c|} \hline K_2 \leftarrow \{0, 1\}^\lambda \\ \hline \text{PRF.QUERY}_H(X): \\ \quad \text{if } L^*[X] \text{ undefined:} \\ \quad \quad Y := \text{PRF.QUERY}_F(X) \\ \quad \quad L^*[X] := F(K_2, Y) \\ \quad \text{return } L^*[X] \\ \hline \end{array} \diamond \begin{array}{|c|} \hline \mathcal{L}_{\text{prf-real}}^F \\ \hline K \leftarrow \{0, 1\}^\lambda \\ \hline \text{PRF.QUERY}_F(X): \\ \quad \text{return } F(K, X) \\ \hline \end{array}$$

$$\begin{array}{c}
\cong \boxed{
\begin{array}{l}
K_2 \leftarrow \{0, 1\}^\lambda \\
\text{PRF.QUERY}_H(X) : // \dots
\end{array}
} \diamond \boxed{
\begin{array}{l}
\mathcal{L}_{\text{prf-rand}}^F \\
\text{if } L[X] \text{ undefined:} \\
\quad L[X] \leftarrow \{0, 1\}^\lambda \\
\quad \text{return } L[X]
\end{array}
} \\
\\
\equiv \boxed{
\begin{array}{l}
K_2 \leftarrow \{0, 1\}^\lambda \\
\text{PRF.QUERY}_H(X) : \\
\quad \text{if } L^*[X] \text{ undefined:} \\
\quad \quad \text{if } L_1[X] \text{ undefined:} \\
\quad \quad \quad L_1[X] \leftarrow \{0, 1\}^\lambda \\
\quad \quad \quad Y := L_1[X] \\
\quad \quad \quad L^*[X] := F(K_2, Y) \\
\quad \text{return } L^*[X]
\end{array}
}
\end{array}$$

We can again apply the PRF security of F , this time replacing all calls to $F(K_2, \cdot)$ with a lazy random dictionary. This is a new dictionary, which we call L_2 . The repetitive 3-hop maneuver is not shown.

$$\begin{array}{c}
\boxed{
\begin{array}{l}
K_2 \leftarrow \{0, 1\}^\lambda \\
\text{PRF.QUERY}_H(X) : \\
\quad \text{if } L^*[X] \text{ undefined:} \\
\quad \quad \text{if } L_1[X] \text{ undefined:} \\
\quad \quad \quad L_1[X] \leftarrow \{0, 1\}^\lambda \\
\quad \quad \quad Y := L_1[X] \\
\\
\quad \quad L^*[X] := F(K_2, Y) \\
\quad \text{return } L^*[X]
\end{array}
} \cong \boxed{
\begin{array}{l}
\text{PRF.QUERY}_H(X) : \\
\quad \text{if } L^*[X] \text{ undefined:} \\
\quad \quad \text{if } L_1[X] \text{ undefined:} \\
\quad \quad \quad L_1[X] \leftarrow \{0, 1\}^\lambda \\
\quad \quad \quad Y := L_1[X] \\
\quad \quad \quad \text{if } L_2[Y] \text{ undefined:} \\
\quad \quad \quad \quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad \quad \quad \quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X]
\end{array}
}
\end{array}$$

The “if $L_1[X]$ undefined” condition is always true, because $L_1[X]$ and $L^*[X]$ are always assigned during the same call to $\text{PRF.QUERY}_H(X)$, and we only reach this if-statement if $L^*[X]$ is undefined. Thus, this if-statement’s body can be made unconditional.

$$\begin{array}{c}
\boxed{
\begin{array}{l}
\text{PRF.QUERY}_H(X) : \\
\quad \text{if } L^*[X] \text{ undefined:} \\
\quad \quad \text{if } L_1[X] \text{ undefined:} \\
\quad \quad \quad L_1[X] \leftarrow \{0, 1\}^\lambda \\
\quad \quad \quad Y := L_1[X] \\
\quad \quad \quad \text{if } L_2[Y] \text{ undefined:} \\
\quad \quad \quad \quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad \quad \quad \quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X]
\end{array}
} \equiv \boxed{
\begin{array}{l}
\text{PRF.QUERY}_H(X) : \\
\quad \text{if } L^*[X] \text{ undefined:} \\
\quad \quad L_1[X] \leftarrow \{0, 1\}^\lambda \\
\quad \quad Y := L_1[X] \\
\quad \quad \text{if } L_2[Y] \text{ undefined:} \\
\quad \quad \quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad \quad \quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X]
\end{array}
}
\end{array}$$

Now $L_1[\cdot]$ is not needed, and can be eliminated.

$$\begin{array}{c}
\boxed{
\begin{array}{l}
\text{PRF.QUERY}_H(X) : \\
\quad \text{if } L^*[X] \text{ undefined:} \\
\quad \quad L_1[X] \leftarrow \{0, 1\}^\lambda \\
\quad \quad Y := L_1[X] \\
\quad \quad \text{if } L_2[Y] \text{ undefined:} \\
\quad \quad \quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad \quad \quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X]
\end{array}
} \equiv \boxed{
\begin{array}{l}
\text{PRF.QUERY}_H(X) : \\
\quad \text{if } L^*[X] \text{ undefined:} \\
\quad \quad Y \leftarrow \{0, 1\}^\lambda \\
\quad \quad \text{if } L_2[Y] \text{ undefined:} \\
\quad \quad \quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad \quad \quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X]
\end{array}
}
\end{array}$$

Intuitively, uniformly sampled Y values are unlikely to repeat. That is to say, they are indistinguishable from values that are guaranteed not to repeat. We can formalize this with a 3-hop maneuver involving the following libraries $\mathcal{L}_{\text{samp-rand}}$ and $\mathcal{L}_{\text{samp-uniq}}$, which can be proven indistinguishable (see below).

$$\begin{array}{c}
\boxed{\text{PRF.QUERY}_H(X):} \\
\text{if } L^*[X] \text{ undefined:} \\
\quad Y \leftarrow \{0, 1\}^\lambda \\
\quad \text{if } L_2[Y] \text{ undefined:} \\
\quad \quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad \quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X] \\
\equiv \\
\boxed{\text{PRF.QUERY}_H(X):} \\
\text{if } L^*[X] \text{ undefined:} \\
\quad Y := \text{BDAY.SAMP}() \\
\quad \text{if } L_2[Y] \text{ undefined:} \\
\quad \quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad \quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X] \\
\diamond \\
\boxed{\mathcal{L}_{\text{samp-rand}}} \\
\text{BDAY.SAMP}(): \\
\quad R \leftarrow \{0, 1\}^\lambda \\
\quad \text{return } R \\
\cong \\
\boxed{\text{PRF.QUERY}_H(X): // \dots} \diamond \boxed{\mathcal{L}_{\text{samp-uniq}}} \\
\text{BDAY.SAMP}(): \\
\quad R \leftarrow \{0, 1\}^\lambda \setminus \mathcal{R} \\
\quad \mathcal{R} := \mathcal{R} \cup \{R\} \\
\quad \text{return } R \\
\equiv \\
\boxed{\text{PRF.QUERY}_H(X):} \\
\text{if } L^*[X] \text{ undefined:} \\
\quad Y \leftarrow \{0, 1\}^\lambda \setminus \mathcal{Y} \\
\quad \mathcal{Y} := \mathcal{Y} \cup \{Y\} \\
\quad \text{if } L_2[Y] \text{ undefined:} \\
\quad \quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad \quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X]
\end{array}$$

In this hybrid, the Y -values are guaranteed to not repeat. Thus, the inner if-statement is *always* taken, and its body can be made unconditional.

$$\begin{array}{c}
\boxed{\text{PRF.QUERY}_H(X):} \\
\text{if } L^*[X] \text{ undefined:} \\
\quad Y \leftarrow \{0, 1\}^\lambda \setminus \mathcal{Y} \\
\quad \mathcal{Y} := \mathcal{Y} \cup \{Y\} \\
\quad \text{if } L_2[Y] \text{ undefined:} \\
\quad \quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad \quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X] \\
\equiv \\
\boxed{\text{PRF.QUERY}_H(X):} \\
\text{if } L^*[X] \text{ undefined:} \\
\quad Y \leftarrow \{0, 1\}^\lambda \setminus \mathcal{Y} \\
\quad \mathcal{Y} := \mathcal{Y} \cup \{Y\} \\
\quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X]
\end{array}$$

Now the overall effect of the if-statement's body is to assign a uniformly sampled value to $L^*[X]$. The same logic can be written more directly, without Y , \mathcal{Y} , or $L_2[\cdot]$. The result of these simplification is $\mathcal{L}_{\text{prf-rand}}^H$, which completes the proof.

$$\begin{array}{c}
\boxed{\text{PRF.QUERY}_H(X):} \\
\text{if } L^*[X] \text{ undefined:} \\
\quad Y \leftarrow \{0, 1\}^\lambda \setminus \mathcal{Y} \\
\quad \mathcal{Y} := \mathcal{Y} \cup \{Y\} \\
\quad L_2[Y] \leftarrow \{0, 1\}^\lambda \\
\quad L^*[X] := L_2[Y] \\
\quad \text{return } L^*[X] \\
\equiv \\
\boxed{\mathcal{L}_{\text{prf-rand}}^H} \\
\text{PRF.QUERY}_H(X): \\
\text{if } L^*[X] \text{ undefined:} \\
\quad L^*[X] \leftarrow \{0, 1\}^\lambda \\
\quad \text{return } L^*[X]
\end{array}$$

□

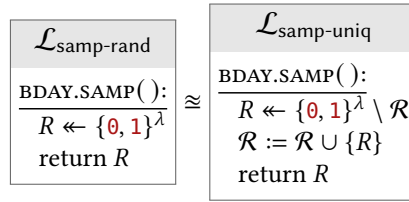
2.1 Discussion

Concept: Statefulness. The PRF security definition is inherently stateful. Both libraries have persistent state (K in the case of $\mathcal{L}_{\text{prf-real}}$ and the dictionary $L[\cdot]$ in $\mathcal{L}_{\text{prf-rand}}$).

Concept: Lazy random dictionary. I call the “ideal PRF” a *lazy random dictionary*. Whenever a PRF is used in a security proof, expect a lot of reasoning about lazy random dictionaries, especially regarding whether certain entries are defined.

Concept: birthday bound and simple bad events. The proof used the following fact, which can be seen as a formalization of the birthday bound in terms of indistinguishable libraries.

Lemma 3. *The following libraries are indistinguishable:*



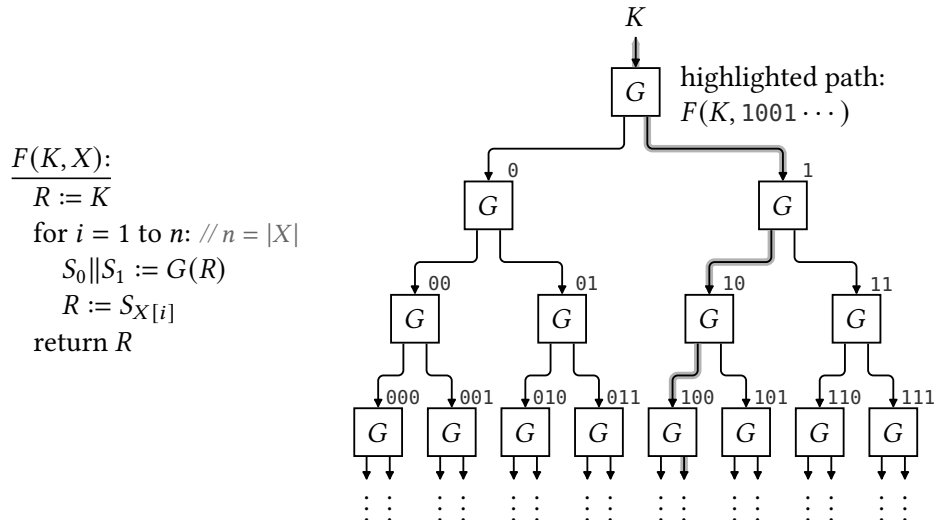
The lemma can be proven using the standard bad event technique. The idea that “uniformly sampled values don’t repeat” is common in security proofs. Thus, there is value in enshrining the preceding lemma. When applying this birthday lemma, the use of bad events is abstracted away, and does not appear in main flow of the proof.

In later examples, however, bad event reasoning is more subtle and more specialized to the proof at hand, and thus there is less to be gained by explicitly abstracting a lemma about two libraries.

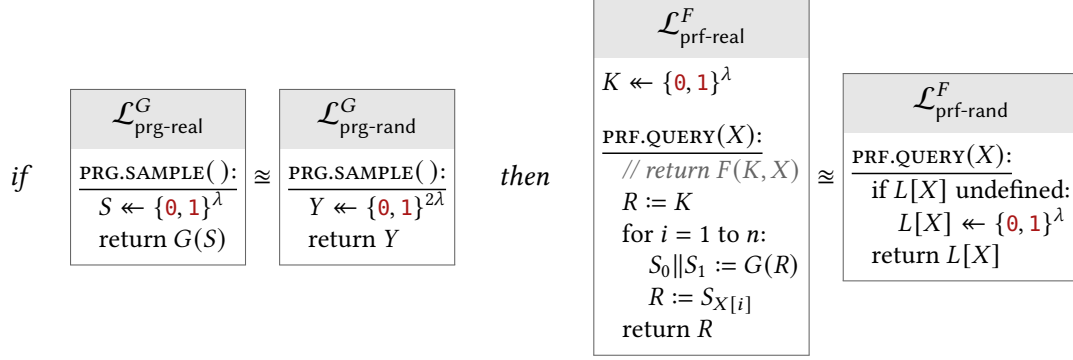
Bonus: The same construction is secure when using key schedule $K_1 = K_2$. The proof requires more sophisticated bad event reasoning.

3 The GGM Construction

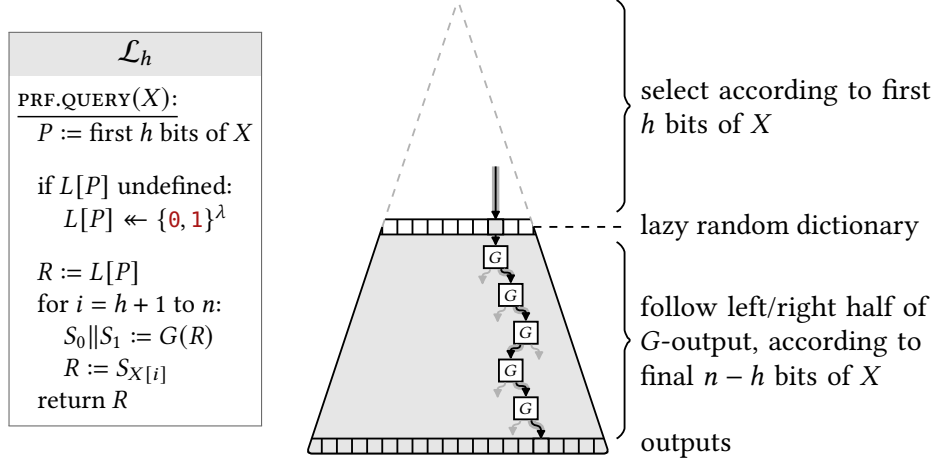
Claim 4. *If G is a secure length-doubling PRG, then the GGM construction (below) is a secure PRF.*



In other words,

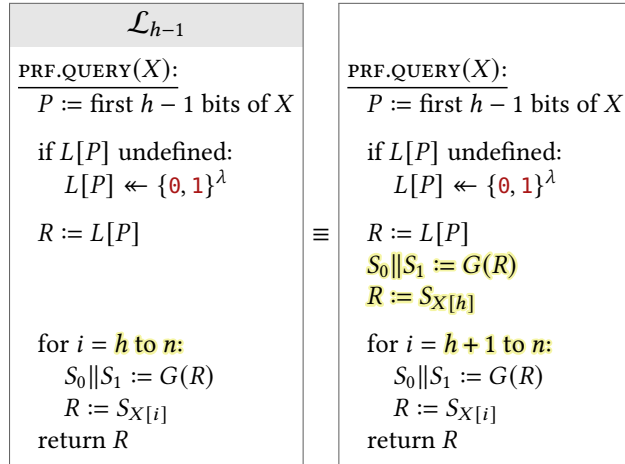


Proof. The number of hybrids in the proof depends on the input-length parameter n . Define the following hybrid library \mathcal{L}_h , where $h \in \{0, \dots, n\}$ is a fixed parameter:



It suffices to show that $\mathcal{L}_{i-1} \approx \mathcal{L}_i$ for all $i \in \{1, \dots, n\}$, because $\mathcal{L}_0 \equiv \mathcal{L}_{\text{prf-real}}^F$ and $\mathcal{L}_n \equiv \mathcal{L}_{\text{prf-rand}}^F$.

The starting point is \mathcal{L}_{h-1} . Unroll the first iteration of the for-loop ($i = h$). Since $h \in \{1, \dots, n\}$, the loop indeed has a first iteration. The change has no effect on the calling program.



Replace a redundant variable to clean things up.

$\begin{array}{l} \text{PRF.QUERY}(X): \\ \hline P := \text{first } h - 1 \text{ bits of } X \\ \text{if } L[P] \text{ undefined:} \\ \quad L[P] \leftarrow \{\mathbf{0}, \mathbf{1}\}^\lambda \\ \\ R := L[P] \\ S_0 \ S_1 := G(R) \\ R := S_{X[h]} \\ \\ \text{for } i = h + 1 \text{ to } n: \\ \quad S_0 \ S_1 := G(R) \\ \quad R := S_{X[i]} \\ \text{return } R \end{array}$	\equiv	$\begin{array}{l} \text{PRF.QUERY}(X): \\ \hline P := \text{first } h - 1 \text{ bits of } X \\ \text{if } L[P] \text{ undefined:} \\ \quad L[P] \leftarrow \{\mathbf{0}, \mathbf{1}\}^\lambda \\ \\ S_0 \ S_1 := G(L[P]) \\ R := S_{X[h]} \\ \\ \text{for } i = h + 1 \text{ to } n: \\ \quad S_0 \ S_1 := G(R) \\ \quad R := S_{X[i]} \\ \text{return } R \end{array}$
--	----------	--

We can rename S_0 and S_1 to $L[P\|\mathbf{0}]$ and $L[P\|\mathbf{1}]$. Then we can select the correct one using the entire h -bit prefix P' of X — not just the h -th individual bit.

$\begin{array}{l} \text{PRF.QUERY}(X): \\ \hline P := \text{first } h - 1 \text{ bits of } X \\ \\ \text{if } L[P] \text{ undefined:} \\ \quad L[P] \leftarrow \{\mathbf{0}, \mathbf{1}\}^\lambda \\ \\ S_0 \ S_1 := G(L[P]) \\ \\ R := S_{X[h]} \\ \\ \text{for } i = h + 1 \text{ to } n: \\ \quad S_0 \ S_1 := G(R) \\ \quad R := S_{X[i]} \\ \text{return } R \end{array}$	\equiv	$\begin{array}{l} \text{PRF.QUERY}(X): \\ \hline P := \text{first } h - 1 \text{ bits of } X \\ P' := \text{first } h \text{ bits of } X \\ \text{if } L[P] \text{ undefined:} \\ \quad L[P] \leftarrow \{\mathbf{0}, \mathbf{1}\}^\lambda \\ \\ L[P\ \mathbf{0}] \ L[P\ \mathbf{1}] := G(L[P]) \\ R := L[P'] \\ \\ \text{for } i = h + 1 \text{ to } n: \\ \quad S_0 \ S_1 := G(R) \\ \quad R := S_{X[i]} \\ \text{return } R \end{array}$
--	----------	---

The library samples $L[P]$ and then later — perhaps many times — computes $G(L[P])$. We could instead compute $G(L[P])$ immediately when $L[P]$ is sampled, and store the results for later.

$\begin{array}{l} \text{PRF.QUERY}(X): \\ \hline P := \text{first } h - 1 \text{ bits of } X \\ P' := \text{first } h \text{ bits of } X \\ \\ \text{if } L[P] \text{ undefined:} \\ \quad L[P] \leftarrow \{\mathbf{0}, \mathbf{1}\}^\lambda \\ \\ L[P\ \mathbf{0}] \ L[P\ \mathbf{1}] := G(L[P]) \\ R := L[P'] \\ \\ \text{for } i = h + 1 \text{ to } n: \\ \quad S_0 \ S_1 := G(R) \\ \quad R := S_{X[i]} \\ \text{return } R \end{array}$	\equiv	$\begin{array}{l} \text{PRF.QUERY}(X): \\ \hline P := \text{first } h - 1 \text{ bits of } X \\ P' := \text{first } h \text{ bits of } X \\ \\ \text{if } L[P] \text{ undefined:} \\ \quad L[P] \leftarrow \{\mathbf{0}, \mathbf{1}\}^\lambda \\ \quad L[P\ \mathbf{0}] \ L[P\ \mathbf{1}] := G(L[P]) // \rightarrow \\ \\ R := L[P'] \\ \\ \text{for } i = h + 1 \text{ to } n: \\ \quad S_0 \ S_1 := G(R) \\ \quad R := S_{X[i]} \\ \text{return } R \end{array}$
--	----------	---

The library assigns $L[P]$ and $L[P']$ at the same time, for all possible pairs P and P' , so it doesn't matter

which of $L[P]$ or $L[P']$ we use in the if-condition.

<pre> PRF.QUERY(X): $P :=$ first $h - 1$ bits of X $P' :=$ first h bits of X if $L[P]$ undefined: $L[P] \leftarrow \{0, 1\}^\lambda$ $L[P\ 0] \parallel L[P\ 1] := G(L[P])$ // \rightarrow $R := L[P']$ for $i = h + 1$ to n: $S_0 \parallel S_1 := G(R)$ $R := S_{X[i]}$ return R </pre>	\equiv	<pre> PRF.QUERY(X): $P :=$ first $h - 1$ bits of X $P' :=$ first h bits of X if $L[P']$ undefined: $L[P] \leftarrow \{0, 1\}^\lambda$ $L[P\ 0] \parallel L[P\ 1] := G(L[P])$ $R := L[P']$ for $i = h + 1$ to n: $S_0 \parallel S_1 := G(R)$ $R := S_{X[i]}$ return R </pre>
--	----------	---

Now $L[P]$ is used *only* as the seed to the PRG, so we can apply the security of the PRG G . The standard 3-hop maneuver is not shown.

<pre> PRF.QUERY(X): $P :=$ first $h - 1$ bits of X $P' :=$ first h bits of X if $L[P']$ undefined: $L[P] \leftarrow \{0, 1\}^\lambda$ $L[P\ 0] \parallel L[P\ 1] := G(L[P])$ \approx $R := L[P']$ for $i = h + 1$ to n: $S_0 \parallel S_1 := G(R)$ $R := S_{X[i]}$ return R </pre>	\approx	<pre> PRF.QUERY(X): $P :=$ first $h - 1$ bits of X $P' :=$ first h bits of X if $L[P']$ undefined: $L[P\ 0] \parallel L[P\ 1] \leftarrow \{0, 1\}^{2\lambda}$ $R := L[P']$ for $i = h + 1$ to n: $S_0 \parallel S_1 := G(R)$ $R := S_{X[i]}$ return R </pre>
--	-----------	---

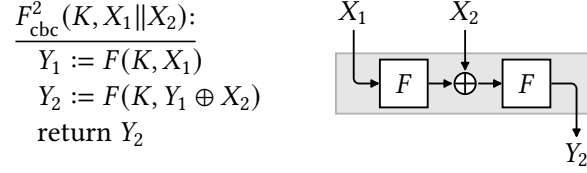
The library simultaneously samples both $L[P']$ and its “sibling” — the same string as P' with last bit flipped. But only one of these values is needed in a single call to PRF.QUERY. Since these values are sampled independently, we can sample only the needed one and defer the sibling value until later (if needed). The result is \mathcal{L}_h , which completes the proof.

<pre> PRF.QUERY(X): $P :=$ first $h - 1$ bits of X $P' :=$ first h bits of X if $L[P']$ undefined: $L[P\ 0] \parallel L[P\ 1] \leftarrow \{0, 1\}^{2\lambda}$ $R := L[P']$ for $i = h + 1$ to n: $S_0 \parallel S_1 := G(R)$ $R := S_{X[i]}$ return R </pre>	\equiv	<div style="background-color: #f0f0f0; padding: 5px; text-align: center;">\mathcal{L}_h</div> <pre> PRF.QUERY(X): $P' :=$ first h bits of X if $L[P']$ undefined: $L[P'] \leftarrow \{0, 1\}^\lambda$ $R := L[P']$ for $i = h + 1$ to n: $S_0 \parallel S_1 := G(R)$ $R := S_{X[i]}$ return R </pre>
---	----------	---

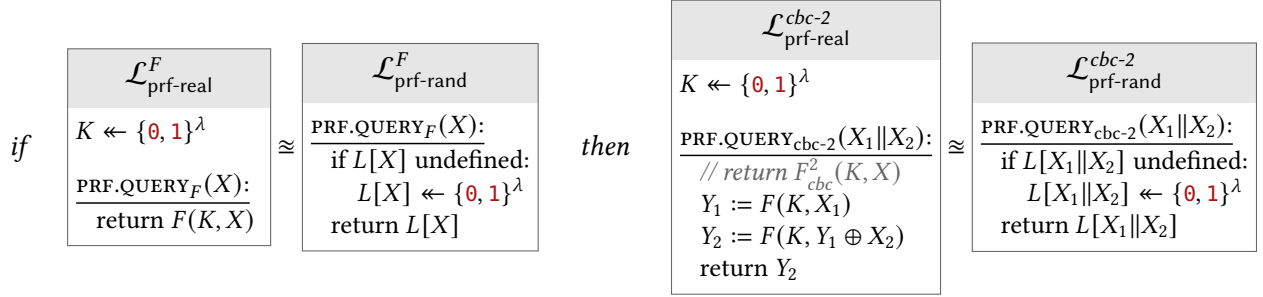
□

4 (2-block) CBC-MAC

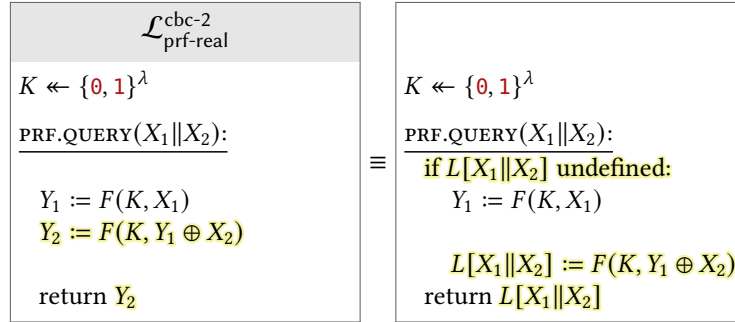
Claim 5. If F is a secure PRF with input/output length λ , then 2-block CBC-MAC (below) is also a secure PRF.



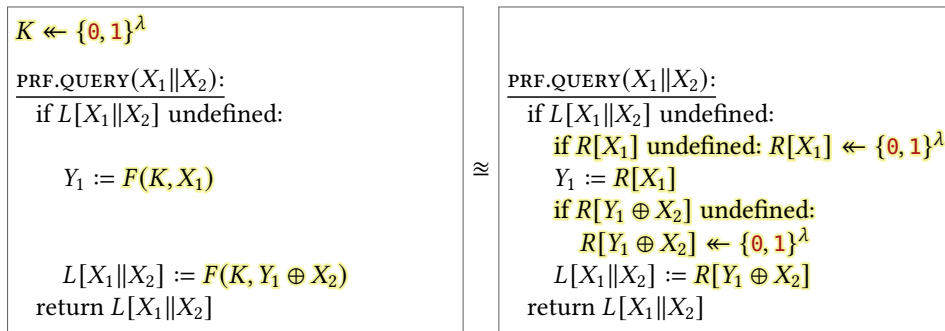
In other words,



Proof. The starting point is $\mathcal{L}_{\text{prf-real}}^{\text{cbc-2}}$. We can add a cache $L[\cdot]$ so that each output is computed only once.



We can replace the calls to PRF F with corresponding lookups in a lazy random dictionary $R[\cdot]$. Note: there is a *single* dictionary $R[\cdot]$, which the library will access on both X_1 and $Y_1 \oplus X_2$. The standard 3-hop maneuver has been omitted.



This hybrid contains an if-statement “if $R[Y_1 \oplus X_2]$ undefined.” Consider another hybrid in which the body of this if-statement is performed *unconditionally*. The two hybrids behave identically unless/until $R[Y_1 \oplus X_2]$

is already defined. So, to show that the hybrids are indistinguishable, we can trigger a bad event in that case and later show that the bad event has negligible probability.

<pre> PRF.QUERY($X_1 \ X_2$): if $L[X_1 \ X_2]$ undefined: if $R[X_1]$ undefined: $R[X_1] \leftarrow \{0, 1\}^\lambda$ $Y_1 := R[X_1]$ if $R[Y_1 \oplus X_2]$ undefined: $R[Y_1 \oplus X_2] \leftarrow \{0, 1\}^\lambda$ $L[X_1 \ X_2] := R[Y_1 \oplus X_2]$ return $L[X_1 \ X_2]$ </pre>	\approx	<pre> PRF.QUERY($X_1 \ X_2$): if $L[X_1 \ X_2]$ undefined: if $R[X_1]$ undefined: $R[X_1] \leftarrow \{0, 1\}^\lambda$ $Y_1 := R[X_1]$ if $R[Y_1 \oplus X_2]$ defined: bad := true $R[Y_1 \oplus X_2] \leftarrow \{0, 1\}^\lambda // \leftarrow$ $L[X_1 \ X_2] := R[Y_1 \oplus X_2]$ return $L[X_1 \ X_2]$ </pre>
---	-----------	--

We can swap the assignment order of $L[X_1 \| X_2]$ and $R[Y_1 \oplus X_2]$.

<pre> PRF.QUERY($X_1 \ X_2$): if $L[X_1 \ X_2]$ undefined: if $R[X_1]$ undefined: $R[X_1] \leftarrow \{0, 1\}^\lambda$ $Y_1 := R[X_1]$ if $R[Y_1 \oplus X_2]$ defined: bad := true $R[Y_1 \oplus X_2] \leftarrow \{0, 1\}^\lambda // \leftarrow$ $L[X_1 \ X_2] := R[Y_1 \oplus X_2]$ return $L[X_1 \ X_2]$ </pre>	\equiv	<pre> PRF.QUERY($X_1 \ X_2$): if $L[X_1 \ X_2]$ undefined: if $R[X_1]$ undefined: $R[X_1] \leftarrow \{0, 1\}^\lambda$ $Y_1 := R[X_1]$ if $R[Y_1 \oplus X_2]$ defined: bad := true $L[X_1 \ X_2] \leftarrow \{0, 1\}^\lambda$ $R[Y_1 \oplus X_2] := L[X_1 \ X_2]$ return $L[X_1 \ X_2]$ </pre>
--	----------	---

We can move $L[X_1 \| X_2] \leftarrow \{0, 1\}^\lambda$ earlier. After doing so, it is clear that $R[\cdot]$ does not affect the output of PRF.QUERY — it is used only to determine whether to trigger the bad event.

<pre> PRF.QUERY($X_1 \ X_2$): if $L[X_1 \ X_2]$ undefined: if $R[X_1]$ undefined: $R[X_1] \leftarrow \{0, 1\}^\lambda$ $Y_1 := R[X_1]$ if $R[Y_1 \oplus X_2]$ defined: bad := true $L[X_1 \ X_2] \leftarrow \{0, 1\}^\lambda$ $R[Y_1 \oplus X_2] := L[X_1 \ X_2]$ return $L[X_1 \ X_2]$ </pre>	\equiv	<pre> PRF.QUERY($X_1 \ X_2$): if $L[X_1 \ X_2]$ undefined: $L[X_1 \ X_2] \leftarrow \{0, 1\}^\lambda$ if $R[X_1]$ undefined: $R[X_1] \leftarrow \{0, 1\}^\lambda$ $Y_1 := R[X_1]$ if $R[Y_1 \oplus X_2]$ defined: bad := true $R[Y_1 \oplus X_2] := L[X_1 \ X_2]$ return $L[X_1 \ X_2]$ </pre>
---	----------	--

Instead of triggering the bad event as PRF.QUERY is called, we can do so at the end of time (*i.e.*, as the adversary terminates, and after it has finished making calls to PRF.QUERY). This will change *when* the bad

event is triggered, but will not affect its overall probability.

<pre> PRF.QUERY($X_1 \ X_2$): if $L[X_1 \ X_2]$ undefined: $L[X_1 \ X_2] \leftarrow \{0, 1\}^\lambda$ if $R[X_1]$ undefined: $R[X_1] \leftarrow \{0, 1\}^\lambda$ $Y_1 := R[X_1]$ if $R[Y_1 \oplus X_2]$ defined: bad := true $R[Y_1 \oplus X_2] := L[X_1 \ X_2]$ return $L[X_1 \ X_2]$ </pre>	\equiv	<div style="background-color: #f0f0f0; padding: 5px; text-align: center; margin-bottom: 5px;"> $\mathcal{L}_{\text{prf-rand}}^{\text{cbc-2}}$ </div> <pre> PRF.QUERY($X_1 \ X_2$): if $L[X_1 \ X_2]$ undefined: $L[X_1 \ X_2] \leftarrow \{0, 1\}^\lambda$ $\mathcal{X} := \mathcal{X} \cup \{X_1 \ X_2\}$ return $L[X_1 \ X_2]$ END OF TIME: for $X_1 \ X_2 \in \mathcal{X}$: if $R[X_1]$ undefined: $R[X_1] \leftarrow \{0, 1\}^\lambda$ $Y_1 := R[X_1]$ if $R[Y_1 \oplus X_2]$ defined: bad := true $R[Y_1 \oplus X_2] := L[X_1 \ X_2]$ </pre>
---	----------	---

The final library is $\mathcal{L}_{\text{prf-rand}}^{\text{cbc-2}}$, along with some extra steps that happen at the end of time, which don't affect the adversary's view (the bad event flag is entirely internal to the library). The purpose of moving the bad event to the end of time is to make the analysis of its probability simpler. Now, values in $R[\cdot]$ are sampled *after* all X_1, X_2 values have been chosen. It is not hard to show that the bad event has probability at most $q^2/2^{\lambda+1}$, but deriving this bound is not the purpose of this document. Since the bad event has negligible probability, the step of the proof that invoked the bad event technique was justified. \square

4.1 Discussion

Concept: end-of-time strategy for bad events. It would be possible to analyze the bad event's probability at the moment it is invoked in the proof. Instead, I have chosen to keep the bad event around until the end of the proof and analyze its probability only in the final hybrid. The reason for doing this is mainly pedagogical. These libraries sample values of the form $R[\cdot]$, and even give some of these values to the adversary, while the adversary adaptively chooses values of X_1 and X_2 . The bad event is triggered based on a complicated condition of these $R[\cdot]$ values and X_1, X_2 . It is tricky to fully understand which values are influenced by others. By moving as much as possible to the end of time — and in particular, after the adversary has finished choosing its inputs to PRF.QUERY — it becomes clear which values are independent of the adversarially chosen ones. While someone with a PhD in cryptography may be able to conclude (correctly) which values are distributed independently of the adversary's view in the penultimate 3 hybrids, the end-of-time strategy is more accessible and less error-prone.

Concept: the Golden Rule of PRFs. In *The Joy of Cryptography*, the Golden Rule of PRFs is:

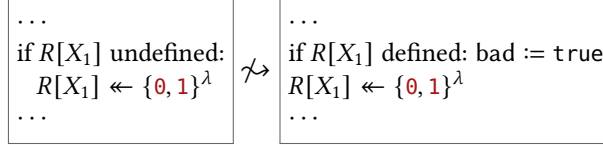
If a PRF F is being used as a component in a larger construction H , then security usually rests on how well H can ensure *distinct* inputs to F .

This rule manifests in the preceding example in the following step:

<pre> ... if $R[Y_1 \oplus X_2]$ undefined: $R[Y_1 \oplus X_2] \leftarrow \{0, 1\}^\lambda$... </pre>	\leadsto	<pre> ... if $R[Y_1 \oplus X_2]$ defined: bad := true $R[Y_1 \oplus X_2] \leftarrow \{0, 1\}^\lambda$... </pre>
--	------------	--

The bad event corresponds precisely to the event that the PRF input (now an index of the lazy random dictionary) $Y_1 \oplus X_2$ repeats. The reasoning of this step of the proof is that we expect such a repeat to be unlikely (indeed, by introducing a bad event, we are now obligated to prove that the bad event has negligible probability), so we can modify the library's behavior to act as if such a repeat never happens. In other words, we make the line “ $R[Y_1 \oplus X_2] \leftarrow \{0, 1\}^\lambda$ ” unconditional.

Note that CBC-MAC involves 2 calls to the PRF, and the Golden Rule applies only to the second one. *I.e.*, we do **not** make the following change:



This is because in CBC-MAC it is easy to force a repeated input to this first PRF call, by invoking CBC-MAC on inputs with a repeated first block. We *do* expect inputs to the first PRF call to repeat, but not inputs to the second PRF call, and security rests only on the latter.

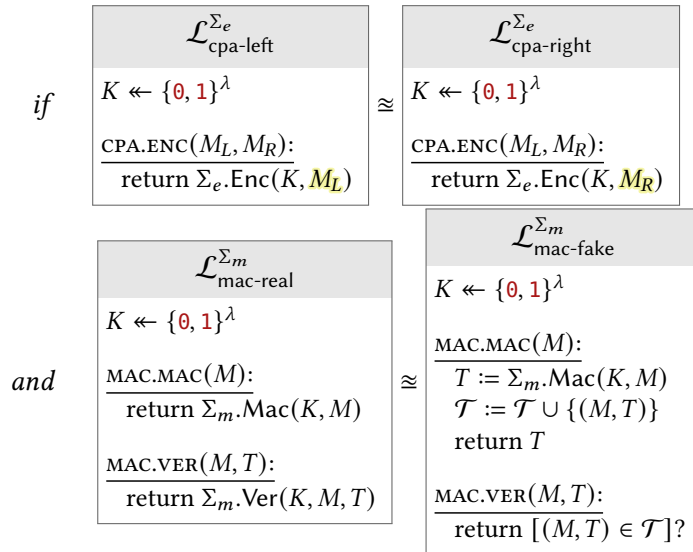
Essentially every proof involving PRFs features this idiom, where we “unconditionalize” a lazy random dictionary assignment and introduce a bad event.

5 Encrypt-then-MAC

Claim 6. *If Σ_e is a CPA-secure symmetric-key encryption scheme and Σ_m is a secure MAC with $\Sigma_e.C \subseteq \Sigma_m.M$ (i.e., ciphertexts from Σ_e are suitable inputs to Σ_m), then encrypt-then-MAC (below) is a CCA-secure symmetric-key encryption scheme:*

$\frac{\text{Enc}(K_e \ K_m, M):}{\begin{array}{l} C' := \Sigma_e.\text{Enc}(K_e, M) \\ T := \Sigma_m.\text{Mac}(K_m, C) \\ \text{return } (C, T) \end{array}}$	$\frac{\text{Dec}(K_e \ K_m, (C', T))}{\begin{array}{l} \text{if } \Sigma_m.\text{Ver}(K_m, C', T): \\ \quad \text{return } \Sigma_e.\text{Dec}(K_e, C') \\ \text{else: return } \perp \end{array}}$
--	---

In other words,



	$\mathcal{L}_{cca-left}$		$\mathcal{L}_{cca-right}$
	$K_e \ K_m \leftarrow \{0, 1\}^{2\lambda}$		$K_e \ K_m \leftarrow \{0, 1\}^{2\lambda}$
	$\text{CCA.ENC}(M_L, M_R):$		$\text{CCA.ENC}(M_L, M_R):$
	$C' := \Sigma_e.\text{Enc}(K_e, M_L)$		$C' := \Sigma_e.\text{Enc}(K_e, M_R)$
	$T := \Sigma_m.\text{Mac}(K_m, C')$		$T := \Sigma_m.\text{Mac}(K_m, C')$
	$C := (C', T)$		$C := (C', T)$
	$\mathcal{D} := \mathcal{D} \cup \{C\}$		$\mathcal{D} := \mathcal{D} \cup \{C\}$
	return C		return C
	$\text{CCA.DEC}(C):$		$\text{CCA.DEC}(C):$
	if $C \in \mathcal{D}$: return \perp		if $C \in \mathcal{D}$: return \perp
	$(C', T) := C$		$(C', T) := C$
	if $\Sigma_m.\text{Ver}(K_m, C', T)$:		if $\Sigma_m.\text{Ver}(K_m, C', T)$:
	return $\Sigma_e.\text{Dec}(K_e, C')$		return $\Sigma_e.\text{Dec}(K_e, C')$
	else: return \perp		else: return \perp

then

\cong

Proof. The starting point is $\mathcal{L}_{cca-left}$. First, apply the security of the MAC scheme; the 3-hop maneuver is not shown.

$\mathcal{L}_{cca-left}$		
$K_e \ K_m \leftarrow \{0, 1\}^{2\lambda}$		$K_e \ K_m \leftarrow \{0, 1\}^{2\lambda}$
$\text{CCA.ENC}(M_L, M_R):$		$\text{CCA.ENC}(M_L, M_R):$
$C' := \Sigma_e.\text{Enc}(K_e, M_L)$		$C' := \Sigma_e.\text{Enc}(K_e, M_L)$
$T := \Sigma_m.\text{Mac}(K_m, C')$		$T := \Sigma_m.\text{Mac}(K_m, C')$
$C := (C', T)$		$\mathcal{T} := \mathcal{T} \cup \{(C', T)\}$
$\mathcal{D} := \mathcal{D} \cup \{C\}$		$C := (C', T)$
return C		$\mathcal{D} := \mathcal{D} \cup \{C\}$
$\text{CCA.DEC}(C):$		return C
if $C \in \mathcal{D}$: return \perp		$\text{CCA.DEC}(C):$
$(C', T) := C$		if $C \in \mathcal{D}$: return \perp
if $\Sigma_m.\text{Ver}(K_m, C', T)$:		$(C', T) := C$
return $\Sigma_e.\text{Dec}(K_e, C')$		if $(C', T) \in \mathcal{T}$:
else: return \perp		return $\Sigma_e.\text{Dec}(K_e, C')$
		else: return \perp

Next, observe that \mathcal{T} and \mathcal{D} have exactly the same contents. Thus, checking membership in \mathcal{T} is equivalent

to checking membership in \mathcal{D} . After replacing all references to \mathcal{T} with \mathcal{D} , variable \mathcal{T} is no longer needed.

$ \begin{array}{l} K_e \ K_m \leftarrow \{0, 1\}^{2\lambda} \\ \text{CCA.ENC}(M_L, M_R): \\ \quad C' := \Sigma_e.\text{Enc}(K_e, M_L) \\ \quad T := \Sigma_m.\text{Mac}(K_m, C') \\ \quad \mathcal{T} := \mathcal{T} \cup \{(C', T)\} \\ \quad C := (C', T) \\ \quad \mathcal{D} := \mathcal{D} \cup \{C\} \\ \quad \text{return } C \\ \text{CCA.DEC}(C): \\ \quad \text{if } C \in \mathcal{D}: \text{return } \perp \\ \quad (C', T) := C \\ \quad \text{if } (C', T) \in \mathcal{T}: \\ \quad \quad \text{return } \Sigma_e.\text{Dec}(K_e, C') \\ \quad \text{else: return } \perp \end{array} $	\equiv	$ \begin{array}{l} K_e \ K_m \leftarrow \{0, 1\}^{2\lambda} \\ \text{CCA.ENC}(M_L, M_R): \\ \quad C' := \Sigma_e.\text{Enc}(K_e, M_L) \\ \quad T := \Sigma_m.\text{Mac}(K_m, C') \\ \quad C := (C', T) \\ \quad \mathcal{D} := \mathcal{D} \cup \{C\} \\ \quad \text{return } C \\ \text{CCA.DEC}(C): \\ \quad \text{if } C \in \mathcal{D}: \text{return } \perp \\ \quad (C', T) := C \\ \quad \text{if } (C', T) \in \mathcal{D}: \\ \quad \quad \text{return } \Sigma_e.\text{Dec}(K_e, C') \\ \quad \text{else: return } \perp \end{array} $
---	----------	--

Now the second if-statement in CCA.DEC is unreachable: if its condition were true, then the subroutine would have already exited in the first if-statement. This unreachable if-statement can be removed.

$ \begin{array}{l} K_e \ K_m \leftarrow \{0, 1\}^{2\lambda} \\ \text{CCA.ENC}(M_L, M_R): // \dots \\ \text{CCA.DEC}(C): \\ \quad \text{if } C \in \mathcal{D}: \text{return } \perp \\ \quad (C', T) := C \\ \quad \text{if } (C', T) \in \mathcal{D}: \\ \quad \quad \text{return } \Sigma_e.\text{Dec}(K_e, C') \\ \quad \text{else: return } \perp \end{array} $	\equiv	$ \begin{array}{l} K_e \ K_m \leftarrow \{0, 1\}^{2\lambda} \\ \text{CCA.ENC}(M_L, M_R): // \dots \\ \text{CCA.DEC}(C): \\ \quad \text{if } C \in \mathcal{D}: \text{return } \perp \\ \quad \text{return } \perp \end{array} $
--	----------	--

Now that K_e is used only for encryption, we can apply CPA security of Σ_e . The standard 3-hop maneuver is not shown.

$ \begin{array}{l} K_e \ K_m \leftarrow \{0, 1\}^{2\lambda} \\ \text{CCA.ENC}(M_L, M_R): \\ \quad C' := \Sigma_e.\text{Enc}(K_e, M_L) \\ \quad T := \Sigma_m.\text{Mac}(K_m, C') \\ \quad C := (C', T) \\ \quad \mathcal{D} := \mathcal{D} \cup \{C\} \\ \quad \text{return } C \\ \text{CCA.DEC}(C): // \dots \end{array} $	\cong	$ \begin{array}{l} K_e \ K_m \leftarrow \{0, 1\}^{2\lambda} \\ \text{CCA.ENC}(M_L, M_R): \\ \quad C' := \Sigma_e.\text{Enc}(K_e, M_R) \\ \quad T := \Sigma_m.\text{Mac}(K_m, C') \\ \quad C := (C', T) \\ \quad \mathcal{D} := \mathcal{D} \cup \{C\} \\ \quad \text{return } C \\ \text{CCA.DEC}(C): // \dots \end{array} $
---	---------	---

Apply exactly the same steps as before, in reverse order, now with M_R in place of M_L . The result is $\mathcal{L}_{\text{cca-right}}$.

which completes the proof.

$K_e \ K_m \leftarrow \{0, 1\}^{2\lambda}$ $\text{CCA.ENC}(M_L, M_R):$ $\begin{aligned} C' &:= \Sigma_e.\text{Enc}(K_e, M_R) \\ T &:= \Sigma_m.\text{Mac}(K_m, C') \\ C &:= (C', T) \\ \mathcal{D} &:= \mathcal{D} \cup \{C\} \\ &\text{return } C \end{aligned}$ $\text{CCA.DEC}(C):$ $\text{if } C \in \mathcal{D}: \text{return } \perp$ $\text{return } \perp$	\approx	$\mathcal{L}_{\text{cca-right}}$ $K_e \ K_m \leftarrow \{0, 1\}^{2\lambda}$ $\text{CCA.ENC}(M_L, M_R):$ $\begin{aligned} C' &:= \Sigma_e.\text{Enc}(K_e, M_R) \\ T &:= \Sigma_m.\text{Mac}(K_m, C') \\ C &:= (C', T) \\ \mathcal{D} &:= \mathcal{D} \cup \{C\} \\ &\text{return } C \end{aligned}$ $\text{CCA.DEC}(C):$ $\text{if } C \in \mathcal{D}: \text{return } \perp$ $(C', T) := C$ $\text{if } \Sigma_m.\text{Ver}(K_m, C', T):$ $\text{return } \Sigma_e.\text{Dec}(K_e, C')$ $\text{else: return } \perp$
---	-----------	---

□

5.1 Discussion

Concept: authenticity properties. Even security for MACs is defined in terms of two indistinguishable libraries. Specifically, the definition is:

$\mathcal{L}_{\text{mac-real}}^{\Sigma_m}$ $K \leftarrow \{0, 1\}^\lambda$ $\text{MAC.MAC}(M):$ $\text{return } \Sigma_m.\text{Mac}(K, M)$ $\text{MAC.VER}(M, T):$ $\text{return } \Sigma_m.\text{Ver}(K, M, T)$	\approx	$\mathcal{L}_{\text{mac-fake}}^{\Sigma_m}$ $K \leftarrow \{0, 1\}^\lambda$ $\text{MAC.MAC}(M):$ $\begin{aligned} T &:= \Sigma_m.\text{Mac}(K, M) \\ \mathcal{T} &:= \mathcal{T} \cup \{(M, T)\} \\ &\text{return } T \end{aligned}$ $\text{MAC.VER}(M, T):$ $\text{return } [(M, T) \in \mathcal{T}]?$
--	-----------	--

The two libraries behave differently *only* if MAC.VER is called with (M, T) such that $\text{Ver}(K, M, T) = \text{true}$ (so that $\mathcal{L}_{\text{mac-real}}$ returns true) but $(M, T) \notin \mathcal{T}$ (so that $\mathcal{L}_{\text{mac-fake}}$ returns false). — in other words, only when MAC.VER is called with a *forgery*. Thus, the libraries are indistinguishable if and only if it is hard to construct a forgery.

6 Simple Random-Oracle PRF

Claim 7. The function $F(K, X) = \mathbb{H}(K \| X)$ is a secure PRF if \mathbb{H} is modeled as a random oracle. In other words, the following libraries are indistinguishable:

$\mathcal{L}_{\text{prf-real+ro}}$		$\mathcal{L}_{\text{prf-rand+ro}}$
$K \leftarrow \{0, 1\}^\lambda$ $\text{PRF.QUERY}(X):$ $\quad // F(K, X) = \mathbb{H}(K\ X)$ $\quad \text{return } \mathbb{H}(K\ X)$ $\mathbb{H}(A):$ $\quad \text{if } H[A] \text{ undefined:}$ $\quad \quad H[A] \leftarrow \{0, 1\}^n$ $\quad \text{return } H[A]$	\cong	$K \leftarrow \{0, 1\}^\lambda$ $\text{PRF.QUERY}(X):$ $\quad \text{if } L[X] \text{ undefined:}$ $\quad \quad L[X] \leftarrow \{0, 1\}^n$ $\quad \text{return } L[X]$ $\mathbb{H}(A):$ $\quad \text{if } H[A] \text{ undefined:}$ $\quad \quad H[A] \leftarrow \{0, 1\}^n$ $\quad \text{return } H[A]$

Proof. The starting point is $\mathcal{L}_{\text{prf-real+ro}}$. Inline the call to \mathbb{H} made by PRF.QUERY .

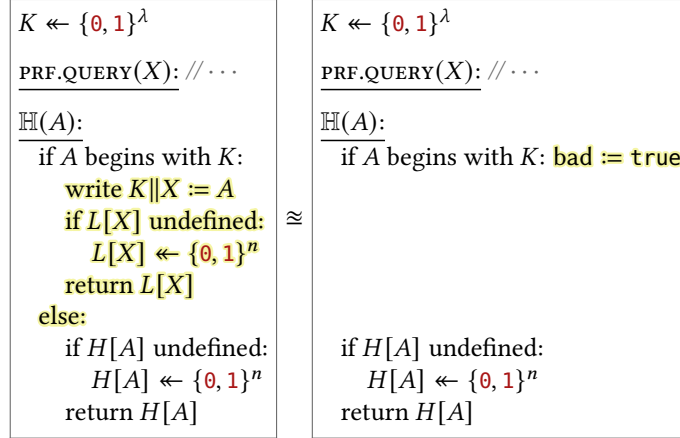
$\mathcal{L}_{\text{prf-real+ro}}$		
$K \leftarrow \{0, 1\}^\lambda$ $\text{PRF.QUERY}(X):$ $\quad // F(K, X) = \mathbb{H}(K\ X)$ $\quad \text{return } \mathbb{H}(K\ X)$ $\mathbb{H}(A):$ $\quad \text{if } H[A] \text{ undefined:}$ $\quad \quad H[A] \leftarrow \{0, 1\}^n$ $\quad \text{return } H[A]$	\equiv	$K \leftarrow \{0, 1\}^\lambda$ $\text{PRF.QUERY}(X):$ $\quad \text{if } H[K\ X] \text{ undefined:}$ $\quad \quad H[K\ X] \leftarrow \{0, 1\}^n$ $\quad \text{return } H[K\ X]$ $\mathbb{H}(A):$ $\quad \text{if } H[A] \text{ undefined:}$ $\quad \quad H[A] \leftarrow \{0, 1\}^n$ $\quad \text{return } H[A]$

\mathbb{H} is implemented by a lazy random dictionary H . We can partition H into two separate dictionaries, based on whether the input starts with the prefix K . Values of the form $H[K\|X]$ are now stored in $L[X]$. Calls to \mathbb{H} made within PRF.QUERY *always* include the K prefix, but the adversary's direct calls to \mathbb{H} may or may not, so the code of \mathbb{H} must check for and support both cases.

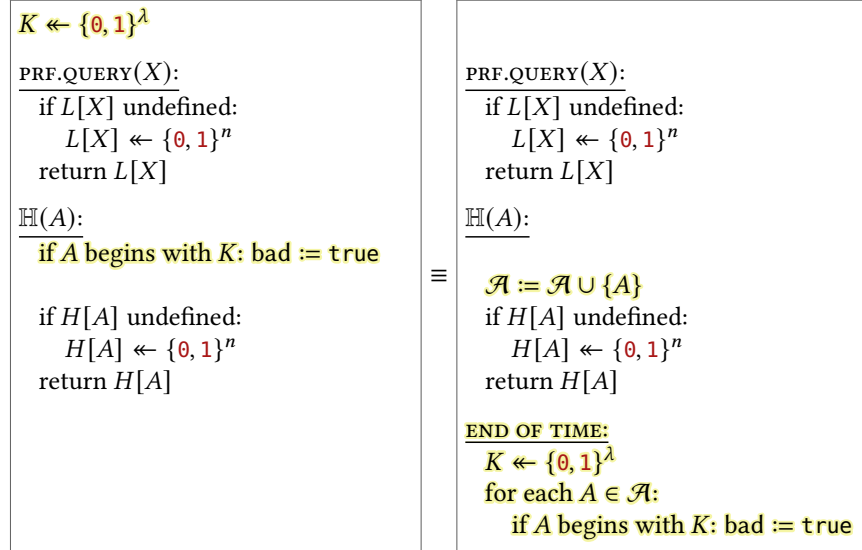
$K \leftarrow \{0, 1\}^\lambda$ $\text{PRF.QUERY}(X):$ $\quad \text{if } H[K\ X] \text{ undefined:}$ $\quad \quad H[K\ X] \leftarrow \{0, 1\}^n$ $\quad \text{return } H[K\ X]$ $\mathbb{H}(A):$ $\quad \text{if } H[A] \text{ undefined:}$ $\quad \quad H[A] \leftarrow \{0, 1\}^n$ $\quad \text{return } H[A]$	\equiv	$K \leftarrow \{0, 1\}^\lambda$ $\text{PRF.QUERY}(X):$ $\quad \text{if } L[X] \text{ undefined:}$ $\quad \quad L[X] \leftarrow \{0, 1\}^n$ $\quad \text{return } L[X]$ $\mathbb{H}(A):$ $\quad \text{if } A \text{ begins with } K:$ $\quad \quad \text{write } K\ X := A$ $\quad \quad \text{if } L[X] \text{ undefined:}$ $\quad \quad \quad L[X] \leftarrow \{0, 1\}^n$ $\quad \quad \text{return } L[X]$ $\quad \text{else:}$ $\quad \quad \text{if } H[A] \text{ undefined:}$ $\quad \quad \quad H[A] \leftarrow \{0, 1\}^n$ $\quad \quad \text{return } H[A]$

Modify \mathbb{H} to act as if its inputs never have a prefix K . The library's behavior will change only in the bad event that the calling program calls \mathbb{H} on an input that begins with K . We must later show that this bad

event has negligible probability.



The value K is used *only* to decide whether to trigger the bad event, so we can move the choice of K and all the bad event logic to the end of time without affecting its probability.



The final library is our desired target $\mathcal{L}_{\text{prf-rand+ro}}$, with some extra logic to determine a bad event at the end of time. We complete the proof by showing that the bad event has negligible probability. Since K is sampled uniformly, *after* \mathcal{A} has been determined, the bad event has probability $|\mathcal{A}|/2^\lambda$, which is negligible since $|\mathcal{A}|$ is polynomial in the security parameter. \square

6.1 Discussion

Concept: random oracles. We model a random oracle as a lazy random dictionary, which is accessible to the adversary directly as well as to the construction.

Concept: partitioning. In the random oracle model, the adversary makes *direct* queries to the random oracle, while the construction makes its own *internal* queries. Most security proofs in the random oracle rely on the fact that there is no overlap between these two sets of queries — *i.e.*, the adversary never manages to directly make an internal query.

This concept manifests in security proofs by partitioning the random oracle's storage according to the direct-vs-internal classification. In the preceding example, we introduced a hybrid that stored direct queries and internal queries in separate data structures.

7 RSA-KEM CCA

Claim 8. *If the RSA assumption holds, then RSA-KEM (below) is a (IND\$-)-CCA-secure KEM in the random oracle model. Without loss of generality, it suffices to prove security of a KEM with respect to a single challenge ciphertext.*

$$\begin{array}{l} \text{Encaps}(PK = (n, e)): \\ \hline R \leftarrow \mathbb{Z}_n \\ C := R^e \% n \\ M := \mathbb{H}(R) \\ \text{return } (C, M) \end{array} \quad \begin{array}{l} \text{Decaps}(SK = (n, d), C): \\ \hline R := C^d \% n \\ \text{return } \mathbb{H}(R) \end{array}$$

In other words,

$$\begin{array}{l} \text{if} \\ \quad \begin{array}{|l} \hline \mathcal{L}_{\text{rsa-real}} \\ \hline (n, e, d) := \text{RSA.KeyGen}() \\ Y \leftarrow \mathbb{Z}_n \\ \hline \text{RSA.CHALLENGE}(): \\ \quad \text{return } (n, e, Y) \\ \hline \text{RSA.CHECK}(X): \\ \quad \text{return } Y \equiv_n X^e \\ \hline \end{array} \end{array} \quad \cong \quad \begin{array}{|l} \hline \mathcal{L}_{\text{rsa-fake}} \\ \hline (n, e, d) := \text{RSA.KeyGen}() \\ Y \leftarrow \mathbb{Z}_n \\ \hline \text{RSA.CHALLENGE}(): \\ \quad \text{return } (n, e, Y) \\ \hline \text{RSA.CHECK}(X): \\ \quad \text{return false} \\ \hline \end{array}$$

$$\begin{array}{l} \text{then} \\ \quad \begin{array}{|l} \hline \mathcal{L}_{\text{kem-1cca-real+ro}} \\ \hline (n, e, d) := \text{RSA.KeyGen}() \\ PK := (n, e) \\ \hline // (C^*, M^*) := \text{Encaps}(PK): \\ R^* \leftarrow \mathbb{Z}_n \\ C^* := (R^*)^e \% n \\ M^* := \mathbb{H}(R^*) \\ \hline \text{1CCA.PK}(): \\ \quad \text{return } PK \\ \hline \text{1CCA.ENC}(): \\ \quad \text{return } (C^*, M^*) \\ \hline \text{1CCA.DEC}(C): \\ \quad // \text{Decaps}(C): \\ \quad \text{return } \mathbb{H}(C^d \% n) \\ \hline \text{H}(A): \\ \quad \text{if } H[A] \text{ undefined:} \\ \quad \quad H[A] \leftarrow \{0, 1\}^\lambda \\ \quad \text{return } H[A] \\ \hline \end{array} \end{array} \quad \cong \quad \begin{array}{|l} \hline \mathcal{L}_{\text{kem-1cca-rand+ro}} \\ \hline (n, e, d) := \text{RSA.KeyGen}() \\ PK := (n, e) \\ \hline C^* \leftarrow \mathbb{Z}_n \\ M^* \leftarrow \{0, 1\}^\lambda \\ \hline \text{1CCA.PK}(): \\ \quad \text{return } PK \\ \hline \text{1CCA.ENC}(): \\ \quad \text{return } (C^*, M^*) \\ \hline \text{1CCA.DEC}(C): \\ \quad \text{if } C == C^*: \text{return } M^* \\ \quad \text{return } \mathbb{H}(C^d \% n) \\ \hline \text{H}(A): \\ \quad \text{if } H[A] \text{ undefined:} \\ \quad \quad H[A] \leftarrow \{0, 1\}^\lambda \\ \quad \text{return } H[A] \\ \hline \end{array}$$

Proof. The starting point of the hybrid proof is $\mathcal{L}_{\text{kem-1cca-real+ro}}$. We can inline all of the calls to \mathbb{H} . The call

to \mathbb{H} that determines M^* is the first, so its result is always sampled uniformly.

$\mathcal{L}_{\text{kem-1cca-real+ro}}$	
$(n, e, d) := \text{RSA.KeyGen}()$ $PK := (n, e)$ $// (C^*, M^*) := \text{Encaps}(PK):$ $R^* \leftarrow \mathbb{Z}_n$ $C^* := (R^*)^e \% n$ $M^* := \mathbb{H}(R^*)$ $\text{1CCA.PK}():$ $\quad \text{return } PK$ $\text{1CCA.ENC}():$ $\quad \text{return } (C^*, M^*)$ $\text{1CCA.DEC}(C):$ $\quad // \text{Decaps}(C):$ $\quad \text{return } \mathbb{H}(C^d \% n)$ $\mathbb{H}(A):$ $\quad \text{if } H[A] \text{ undefined:}$ $\quad \quad H[A] \leftarrow \{0, 1\}^\lambda$ $\quad \text{return } H[A]$	$(n, e, d) := \text{RSA.KeyGen}()$ $PK := (n, e)$ $R^* \leftarrow \mathbb{Z}_n$ $C^* := (R^*)^e \% n$ $M^* := H[R^*] \leftarrow \{0, 1\}^\lambda$ $\text{1CCA.PK}():$ $\quad \text{return } PK$ $\text{1CCA.ENC}():$ $\quad \text{return } (C^*, M^*)$ $\text{1CCA.DEC}(C):$ $\quad \text{if } H[C^d \% n] \text{ undefined:}$ $\quad \quad H[C^d \% n] \leftarrow \{0, 1\}^\lambda$ $\quad \text{return } H[C^d \% n]$ $\mathbb{H}(A):$ $\quad \text{if } H[A] \text{ undefined:}$ $\quad \quad H[A] \leftarrow \{0, 1\}^\lambda$ $\quad \text{return } H[A]$

Now we can perform the renaming suggested above. Whatever was previously stored in $H[A]$ is now stored in $\mathcal{D}[A^e]$. Thus $H[C^d]$ becomes $\mathcal{D}[C]$.

$(n, e, d) := \text{RSA.KeyGen}()$ $PK := (n, e)$ $R^* \leftarrow \mathbb{Z}_n$ $C^* := (R^*)^e \% n$ $M^* := H[R^*] \leftarrow \{0, 1\}^\lambda$ $\text{1CCA.PK}(): // \dots$ $\text{1CCA.ENC}(): // \dots$ $\text{1CCA.DEC}(C):$ $\quad \text{if } H[C^d \% n] \text{ undefined:}$ $\quad \quad H[C^d \% n] \leftarrow \{0, 1\}^\lambda$ $\quad \text{return } H[C^d \% n]$ $\mathbb{H}(A):$ $\quad \text{if } H[A] \text{ undefined:}$ $\quad \quad H[A] \leftarrow \{0, 1\}^\lambda$ $\quad \text{return } H[A]$	$(n, e, d) := \text{RSA.KeyGen}()$ $PK := (n, e)$ $R^* \leftarrow \mathbb{Z}_n$ $C^* := (R^*)^e \% n$ $M^* := \mathcal{D}[C^*] \leftarrow \{0, 1\}^\lambda$ $\text{1CCA.PK}(): // \dots$ $\text{1CCA.ENC}(): // \dots$ $\text{1CCA.DEC}(C):$ $\quad \text{if } \mathcal{D}[C] \text{ undefined:}$ $\quad \quad \mathcal{D}[C] \leftarrow \{0, 1\}^\lambda$ $\quad \text{return } \mathcal{D}[C]$ $\mathbb{H}(A):$ $\quad \text{if } \mathcal{D}[A^e \% n] \text{ undefined:}$ $\quad \quad \mathcal{D}[A^e \% n] \leftarrow \{0, 1\}^\lambda$ $\quad \text{return } \mathcal{D}[A^e \% n]$
--	--

We can carve out an exception for the challenge ciphertext C^* , so that $\mathcal{D}[C^*]$ is never used. Additionally, now the value R^* is used only to compute C^* . Since R^* is uniform, and exponentiation-by- e is a 1-to-1

function, the resulting distribution on C^* is uniform.

$(n, e, d) := \text{RSA.KeyGen}()$ $PK := (n, e)$ $R^* \leftarrow \mathbb{Z}_n$ $C^* := (R^*)^e \% n$ $M^* := \mathcal{D}[C^*] \leftarrow \{0, 1\}^\lambda$ $\text{1CCA.PK}(): // \dots$ $\text{1CCA.ENC}(): // \dots$ $\text{1CCA.DEC}(C):$ if $\mathcal{D}[C]$ undefined: $\mathcal{D}[C] \leftarrow \{0, 1\}^\lambda$ return $\mathcal{D}[C]$ $\mathbb{H}(A):$ if $\mathcal{D}[A^e \% n]$ undefined: $\mathcal{D}[A^e \% n] \leftarrow \{0, 1\}^\lambda$ return $\mathcal{D}[A^e \% n]$	\equiv	$(n, e, d) := \text{RSA.KeyGen}()$ $PK := (n, e)$ $C^* \leftarrow \mathbb{Z}_n$ $M^* \leftarrow \{0, 1\}^\lambda$ $\text{1CCA.PK}(): // \dots$ $\text{1CCA.ENC}(): // \dots$ $\text{1CCA.DEC}(C):$ if $C == C^*$: return M^* if $\mathcal{D}[C]$ undefined: $\mathcal{D}[C] \leftarrow \{0, 1\}^\lambda$ return $\mathcal{D}[C]$ $\mathbb{H}(A):$ if $A^e \equiv_n C^*$: return M^* if $\mathcal{D}[A^e \% n]$ undefined: $\mathcal{D}[A^e \% n] \leftarrow \{0, 1\}^\lambda$ return $\mathcal{D}[A^e \% n]$
--	----------	---

The special case in $\mathbb{H}(A)$ is triggered when the calling program provides an e -th root of C^* . Since d is no longer used anywhere in the library, we can apply the RSA assumption to argue that this special case never happens.

$(n, e, d) := \text{RSA.KeyGen}()$ $PK := (n, e)$ $C^* \leftarrow \mathbb{Z}_n$ $M^* \leftarrow \{0, 1\}^\lambda$ $\text{1CCA.PK}(): // \dots$ $\text{1CCA.ENC}(): // \dots$ $\text{1CCA.DEC}(C): // \dots$ $\mathbb{H}(A):$ if $A^e \equiv_n C^*$: return M^* if $\mathcal{D}[A^e \% n]$ undefined: $\mathcal{D}[A^e \% n] \leftarrow \{0, 1\}^\lambda$ return $\mathcal{D}[A^e \% n]$	\equiv	$(n, e, C^*) \leftarrow \text{RSA.CHALLENGE}()$ $PK := (n, e)$ $M^* \leftarrow \{0, 1\}^\lambda$ $\text{1CCA.PK}(): // \dots$ $\text{1CCA.ENC}(): // \dots$ $\text{1CCA.DEC}(C): // \dots$ $\mathbb{H}(A):$ if $\text{RSA.CHECK}(A)$: return M^* if $\mathcal{D}[A^e \% n]$ undefined: $\mathcal{D}[A^e \% n] \leftarrow \{0, 1\}^\lambda$ return $\mathcal{D}[A^e \% n]$	\diamond	$\mathcal{L}_{\text{rsa-real}}$ $(n, e, d) := \text{RSA.KeyGen}()$ $Y \leftarrow \mathbb{Z}_n$ $\text{RSA.CHALLENGE}():$ return (n, e, Y) $\text{RSA.CHECK}(X):$ return $Y \equiv_n X^e$
	\cong	$(n, e, C^*) \leftarrow \text{RSA.CHALLENGE}()$ $PK := (n, e)$ $M^* \leftarrow \{0, 1\}^\lambda$ $\text{1CCA.PK}(): // \dots$ $\text{1CCA.ENC}(): // \dots$ $\text{1CCA.DEC}(C): // \dots$ $\mathbb{H}(A): // \dots$	\diamond	$\mathcal{L}_{\text{rsa-fake}}$ $(n, e, d) := \text{RSA.KeyGen}()$ $Y \leftarrow \mathbb{Z}_n$ $\text{RSA.CHALLENGE}():$ return (n, e, Y) $\text{RSA.CHECK}(X):$ return false

```

 $(n, e, d) := \text{RSA.KeyGen}()$ 
 $PK := (n, e)$ 

 $C^* \leftarrow \mathbb{Z}_n$ 
 $M^* \leftarrow \{0, 1\}^\lambda$ 

 $\frac{1\text{CCA.PK}(): // \dots}{\equiv \frac{1\text{CCA.ENC}(): // \dots}{1\text{CCA.DEC}(C): // \dots}}$ 

 $\mathbb{H}(A):$ 
  if false: return  $M^*$ 
  if  $\mathcal{D}[A^e \% n]$  undefined:
     $\mathcal{D}[A^e \% n] \leftarrow \{0, 1\}^\lambda$ 
  return  $\mathcal{D}[A^e \% n]$ 

```

We can remove the unreachable if-statement and also revert the previous change, replacing $\mathcal{D}[A^e]$ with $H[A]$. The result is $\mathcal{L}_{\text{kem-1cca-rand+ro}}$.

<pre> $(n, e, d) := \text{RSA.KeyGen}()$ $PK := (n, e)$ $C^* \leftarrow \mathbb{Z}_n$ $M^* \leftarrow \{0, 1\}^\lambda$ $\frac{1\text{CCA.PK}():}{\text{return } PK}$ $\frac{1\text{CCA.ENC}():}{\text{return } (C^*, M^*)}$ $\frac{1\text{CCA.DEC}(C):}{\text{if } C == C^*: \text{return } M^*}$ if $\mathcal{D}[C]$ undefined: $\mathcal{D}[C] \leftarrow \{0, 1\}^\lambda$ return $\mathcal{D}[C]$ $\mathbb{H}(A):$ if false: return M^* if $\mathcal{D}[A^e \% n]$ undefined: $\mathcal{D}[A^e \% n] \leftarrow \{0, 1\}^\lambda$ return $\mathcal{D}[A^e \% n]$ </pre>	\equiv	<div style="background-color: #f0f0f0; text-align: center; padding: 2px 5px; margin-bottom: 5px;">$\mathcal{L}_{\text{kem-1cca-rand+ro}}$</div> <pre> $(n, e, d) := \text{RSA.KeyGen}()$ $PK := (n, e)$ $C^* \leftarrow \mathbb{Z}_n$ $M^* \leftarrow \{0, 1\}^\lambda$ $\frac{1\text{CCA.PK}():}{\text{return } PK}$ $\frac{1\text{CCA.ENC}():}{\text{return } (C^*, M^*)}$ $\frac{1\text{CCA.DEC}(C):}{\text{if } C == C^*: \text{return } M^*}$ return $\mathbb{H}(C^d \% n)$ $\mathbb{H}(A):$ if $H[A]$ undefined: $H[A] \leftarrow \{0, 1\}^\lambda$ return $H[A]$ </pre>
--	----------	--

□

7.1 Discussion

Defining CCA security. The standard approach for defining CCA security is for the decryption oracle to return \perp if it is queried with a challenge ciphertext (*i.e.*, a ciphertext generated by the library/game itself). My preference for both symmetric-key and public-key encryption, which I have followed in this section's example, is to instead allow the decryption oracle to return the “expected” plaintext, for example (SKE):

$\mathcal{L}_{cca-real}$		$\mathcal{L}_{cca-ideal}$
$K \leftarrow \{0, 1\}^\lambda$		$K \leftarrow \{0, 1\}^\lambda$
$\text{CCA.ENC}(M):$ return $\text{Enc}(K, M)$	\cong	$\text{CCA.ENC}(M):$ $C \leftarrow \mathcal{C}(M)$ $\mathcal{D}[C] := M$ return C
$\text{CCA.DEC}(C):$ return $\text{Dec}(K, C)$		$\text{CCA.DEC}(C):$ if $\mathcal{D}[C]$ defined: return $\mathcal{D}[C]$ return $\text{Dec}(K, C)$

Thus, the “real” library provides completely *unrestricted* access to a decryption oracle, while the “ideal” library remembers the association between challenge ciphertexts and their plaintexts, so that the decryption oracle can “lie” convincingly. This is a completely stylistic choice, and not a mandatory requirement for the library-based approach. Furthermore, both flavors of CCA security definition are equivalent, because the set of challenge ciphertexts is public knowledge (and thus the knowledge of whether the decryption oracle is being called with a challenge ciphertext), as is the value of their corresponding challenge plaintexts.

8 RSA-PSS

8.1 Random Self-Reduction for RSA

Lemma 9. *If a single-challenge RSA assumption holds, then a multi-challenge variant also holds. In other words,*

	<div> $\mathcal{L}_{\text{rsa-real}}$ </div> <div> $(n, e, d) := \text{RSA.KeyGen}()$ $Y \leftarrow \mathbb{Z}_n$ <i>if</i> $\frac{\text{RSA.CHALLENGE}():}{\text{return } (n, e, Y)}$ $\frac{\text{RSA.CHECK}(X):}{\text{return } Y \equiv_n X^e}$ </div>	<div> $\mathcal{L}_{\text{rsa-fake}}$ </div> <div> $(n, e, d) := \text{RSA.KeyGen}()$ $Y \leftarrow \mathbb{Z}_n$ $\cong \frac{\text{RSA.CHALLENGE}():}{\text{return } (n, e, Y)}$ $\frac{\text{RSA.CHECK}(X):}{\text{return false}}$ </div>
<i>then</i>	<div> $\mathcal{L}_{\text{rsa}^*\text{-real}}$ </div> <div> $(n, e, d) := \text{RSA.KeyGen}()$ $\frac{\text{RSA.PK}^*():}{\text{return } (n, e)}$ $\frac{\text{RSA.CHALLENGE}^*():}{Y \leftarrow \mathbb{Z}_n}$ $\mathcal{Y} := \mathcal{Y} \cup \{Y\}$ $\text{return } Y$ <i>// is X the e-th root of any Y ∈ Y?</i> $\frac{\text{RSA.CHECK}^*(X):}{\text{if } X^e \% n \in \mathcal{Y}: \text{return true}}$ $\text{else: return false}$ </div>	<div> $\mathcal{L}_{\text{rsa}^*\text{-fake}}$ </div> <div> $(n, e, d) := \text{RSA.KeyGen}()$ $\frac{\text{RSA.PK}^*():}{\text{return } (n, e)}$ $\cong \frac{\text{RSA.CHALLENGE}^*():}{Y \leftarrow \mathbb{Z}_n}$ $\text{return } Y$ <i>// is X the e-th root of any Y ∈ Y?</i> $\frac{\text{RSA.CHECK}^*(X):}{\text{return false}}$ </div>

Proof. The starting point for the proof is $\mathcal{L}_{\text{rsa}^*-\text{real}}$. First, we rewrite the check for membership in \mathcal{Y} as a for-loop involving explicit comparisons.

$\mathcal{L}_{\text{rsa}^*-\text{real}}$	
$(n, e, d) := \text{RSA.KeyGen}()$ $\text{RSA.PK}^*():$ return (n, e) $\text{RSA.CHALLENGE}^*():$ $Y \leftarrow \mathbb{Z}_n$ $\mathcal{Y} := \mathcal{Y} \cup \{Y\}$ return Y $\text{RSA.CHECK}^*(X):$ if $X^e \% n \in \mathcal{Y}$: return true else: return false	\equiv
	$(n, e, d) := \text{RSA.KeyGen}()$ $\text{RSA.PK}^*():$ return (n, e) $\text{RSA.CHALLENGE}^*():$ $Y \leftarrow \mathbb{Z}_n$ $\mathcal{Y} := \mathcal{Y} \cup \{Y\}$ return Y $\text{RSA.CHECK}^*(X):$ for each $Y \in \mathcal{Y}$: if $X^e \equiv_n Y$: return true return false

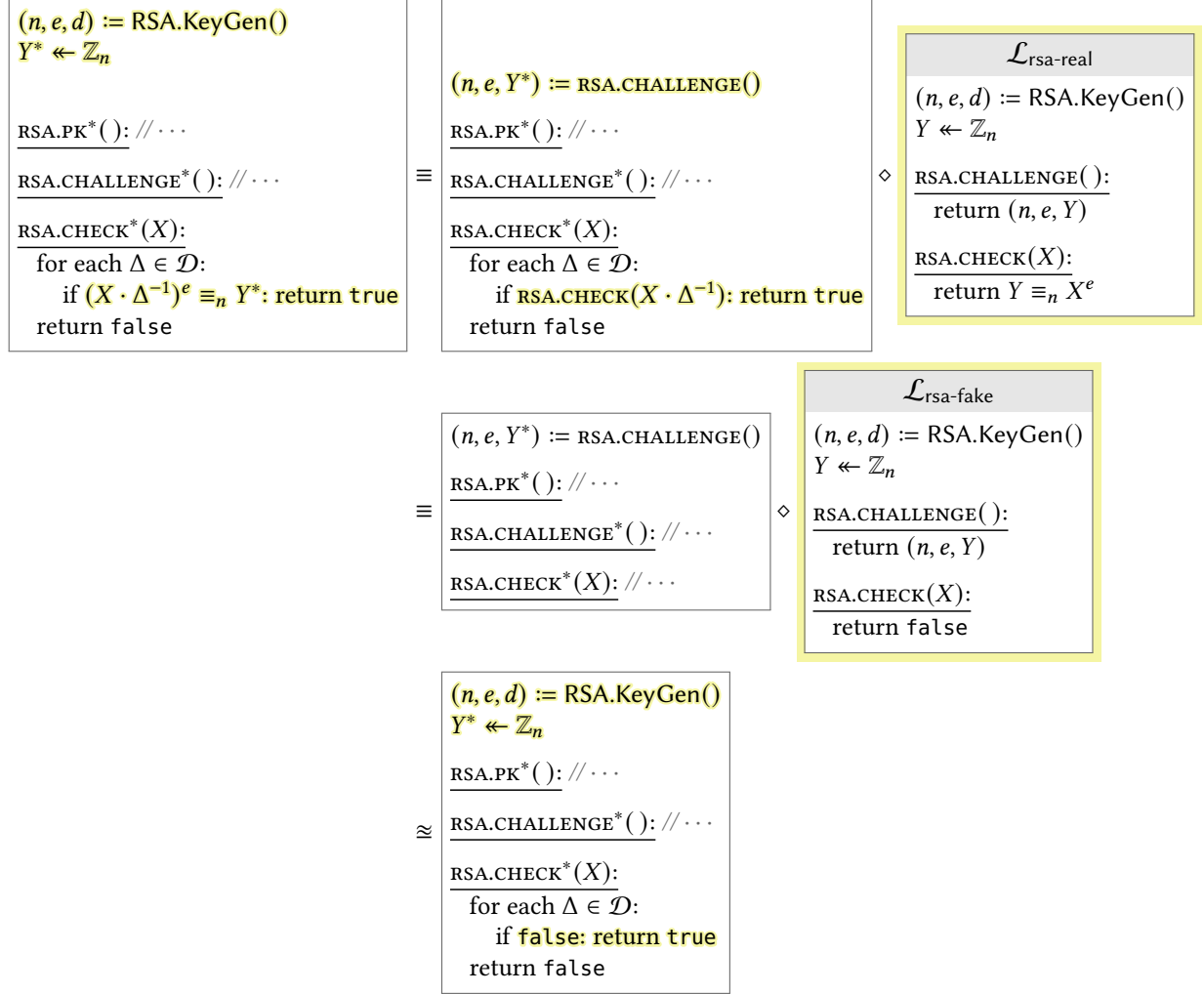
Suppose we initially choose one value Y^* and later define each $Y \in \mathcal{Y}$ as $Y = Y^* \cdot \Delta^e$. If each Δ is uniformly distributed, then the Y 's are also uniformly distributed. In RSA.CHECK^* , we can then replace each Y with its equivalent expression $Y^* \cdot \Delta^e$. Instead of keeping track of a collection of Y -values, it is then enough to keep track of their Δ values.

$(n, e, d) := \text{RSA.KeyGen}()$ $\text{RSA.PK}^*(): // \dots$ $\text{RSA.CHALLENGE}^*():$ $Y \leftarrow \mathbb{Z}_n$ $\mathcal{Y} := \mathcal{Y} \cup \{Y\}$ return Y $\text{RSA.CHECK}^*(X):$ for each $Y \in \mathcal{Y}$: if $X^e \equiv_n Y$: return true return false	\equiv	$(n, e, d) := \text{RSA.KeyGen}()$ $Y^* \leftarrow \mathbb{Z}_n$ $\text{RSA.PK}^*(): // \dots$ $\text{RSA.CHALLENGE}^*():$ $\Delta \leftarrow \mathbb{Z}_n$ $Y := Y^* \cdot \Delta^e \% n$ $\mathcal{D} := \mathcal{D} \cup \{\Delta\}$ return Y $\text{RSA.CHECK}^*(X):$ for each $\Delta \in \mathcal{D}$: if $X^e \equiv_n Y^* \cdot \Delta^e$: return true return false
---	----------	--

Now we can rearrange the equation $X^e \equiv_n Y^* \cdot \Delta^e$ to obtain the equivalent equation $(X\Delta^{-1})^e \equiv_n Y^*$.

$(n, e, d) := \text{RSA.KeyGen}()$ $Y^* \leftarrow \mathbb{Z}_n$ $\text{RSA.PK}^*(): // \dots$ $\text{RSA.CHALLENGE}^*(): // \dots$ $\text{RSA.CHECK}^*(X):$ for each $\Delta \in \mathcal{D}$: if $X^e \equiv_n Y^* \cdot \Delta^e$: return true return false	\equiv	$(n, e, d) := \text{RSA.KeyGen}()$ $Y^* \leftarrow \mathbb{Z}_n$ $\text{RSA.PK}^*(): // \dots$ $\text{RSA.CHALLENGE}^*(): // \dots$ $\text{RSA.CHECK}^*(X):$ for each $\Delta \in \mathcal{D}$: if $(X \cdot \Delta^{-1})^e \equiv_n Y^*$: return true return false
---	----------	--

Now the library has a single target value Y^* and repeatedly checks whether a certain expression $(X \cdot \Delta^{-1})$ is its e -th root. Additionally, the private exponent d is never used. We can therefore apply the RSA assumption, with Y^* playing the role of the single challenge.



Now RSA.CHECK^* always returns false, so its code can be greatly simplified. As we argued above, each Y value is sampled uniformly. Now that the Δ values are not needed, we can simply sample the Y values

uniformly in a more direct way, as they were previously. The result is $\mathcal{L}_{\text{rsa}^* \text{-fake}}$, which completes the proof.

$(n, e, d) := \text{RSA.KeyGen}()$ $Y^* \leftarrow \mathbb{Z}_n$ $\text{RSA.PK}^*():$ $\quad \text{return } (n, e)$ $\text{RSA.CHALLENGE}^*():$ $\quad \Delta \leftarrow \mathbb{Z}_n$ $\quad Y := Y^* \cdot \Delta^e \% n$ $\quad \mathcal{D} := \mathcal{D} \cup \{\Delta\}$ $\quad \text{return } Y$ $\text{RSA.CHECK}^*(X):$ $\quad \text{for each } \Delta \in \mathcal{D}:$ $\quad \quad \text{if false: return true}$ $\quad \text{return false}$	\equiv	$\mathcal{L}_{\text{rsa}^* \text{-fake}}$ $(n, e, d) := \text{RSA.KeyGen}()$ $\text{RSA.PK}^*():$ $\quad \text{return } (n, e)$ $\text{RSA.CHALLENGE}^*():$ $\quad Y \leftarrow \mathbb{Z}_n$ $\quad \text{return } Y$ $\text{RSA.CHECK}^*(X):$ $\quad \text{return false}$
--	----------	---

□

8.2 RSA-PSS

Claim 10. *If the RSA assumption is true, then RSA-PSS (below) is a secure digital signature scheme in the random oracle model.*

$$\begin{array}{l}
 \text{Sign}(SK = (n, d), M): \\
 \quad R \leftarrow \{0, 1\}^\lambda \\
 \quad S := \mathbb{H}(M \| R)^d \% n \\
 \quad \text{return } (R, S)
 \end{array}
 \quad
 \begin{array}{l}
 \text{Verify}(PK = (n, e), M, (R, S)): \\
 \quad \text{return } \mathbb{H}(M \| R) == S^e \% n
 \end{array}$$

In other words,

	$\mathcal{L}_{\text{rsa-real}}$	\cong	$\mathcal{L}_{\text{rsa-fake}}$
	$(n, e, d) := \text{RSA.KeyGen}()$ $Y \leftarrow \mathbb{Z}_n$ $\text{RSA.CHALLENGE}():$ $\quad \text{return } (n, e, Y)$ $\text{RSA.CHECK}(X):$ $\quad \text{return } Y \equiv_n X^e$		$(n, e, d) := \text{RSA.KeyGen}()$ $Y \leftarrow \mathbb{Z}_n$ $\text{RSA.CHALLENGE}():$ $\quad \text{return } (n, e, Y)$ $\text{RSA.CHECK}(X):$ $\quad \text{return false}$
if			

<i>then</i>	$\mathcal{L}_{\text{sig-real+ro}}$	\approx	$\mathcal{L}_{\text{sig-fake}}$
	<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): return (n, e) SIG.SIGN(M): // Sign(SK, M): R ← {0, 1}^λ S := H(M R)^d % n return (R, S) SIG.VER(M, (R, S)): // Verify(PK, M, (R, S)): Y := H(M R) return Y == S^e % n H(A): if H[A] undefined: H[A] ← Z_n return H[A] </pre>		<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): return (n, e) SIG.SIGN(M): R ← {0, 1}^λ S := H(M R)^d % n S := S ∪ {(M, R, S)} return (R, S) SIG.VER(M, (R, S)): if (M, R, S) ∈ S: return true return false H(A): if H[A] undefined: H[A] ← Z_n return H[A] </pre>

Proof. The starting point is $\mathcal{L}_{\text{sig-real+ro}}$: The library can remember signatures produced by SIG.SIGN; they don't need to be verified later in SIG.VER.

$\mathcal{L}_{\text{sig-real+ro}}$	\equiv	<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): return (n, e) SIG.SIGN(M): R ← {0, 1}^λ S := H(M R)^d % n S := S ∪ {(M, R, S)} return (R, S) SIG.VER(M, (R, S)): if (M, R, S) ∈ S: return true Y := H(M R) return Y == S^e % n H(A): if H[A] undefined: H[A] ← Z_n return H[A] </pre>
------------------------------------	----------	---

Inline the call made to \mathbb{H} inside SIG.SIGN.

<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): // ... SIG.SIGN(M): R ← {0, 1}^λ S := H(M R)^d % n S := S ∪ {(M, R, S)} return (R, S) SIG.VER(M, (R, S)): // ... H(A): if H[A] undefined: H[A] ← Z_n return H[A] </pre>	≡	<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): // ... SIG.SIGN(M): R ← {0, 1}^λ if H[M R] undefined: H[M R] ← Z_n S := H[M R]^d % n S := S ∪ {(M, R, S)} return (R, S) SIG.VER(M, (R, S)): // ... H(A): if H[A] undefined: H[A] ← Z_n return H[A] </pre>
---	---	--

Make the if-statement in SIG.SIGN unconditional. This changes the library's behavior only in the bad event that $H[M||R]$ is already defined. However, R is sampled uniformly each time. If the adversary makes at most q calls to the library's subroutines, then there are most q entries in $H[\cdot]$, so at most q "bad" values of R that can be chosen. Overall, across at most q calls to SIG.SIGN, the bad event has negligible probability at most $q^2/2^\lambda$.

<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): // ... SIG.SIGN(M): R ← {0, 1}^λ if H[M R] undefined: H[M R] ← Z_n S := H[M R]^d % n S := S ∪ {(M, R, S)} return (R, S) SIG.VER(M, (R, S)): // ... H(A): // ... </pre>	≈	<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): // ... SIG.SIGN(M): R ← {0, 1}^λ H[M R] ← Z_n S := H[M R]^d % n S := S ∪ {(M, R, S)} return (R, S) SIG.VER(M, (R, S)): // ... H(A): // ... </pre>
---	---	---

Instead of uniformly sampling $H[M||R]$ and setting $S := H[M||R]^d$, we can uniformly sample S and set

$H[M||R] := S^e$. The two methods induce the same joint distribution over $(S, H[M||R])$.

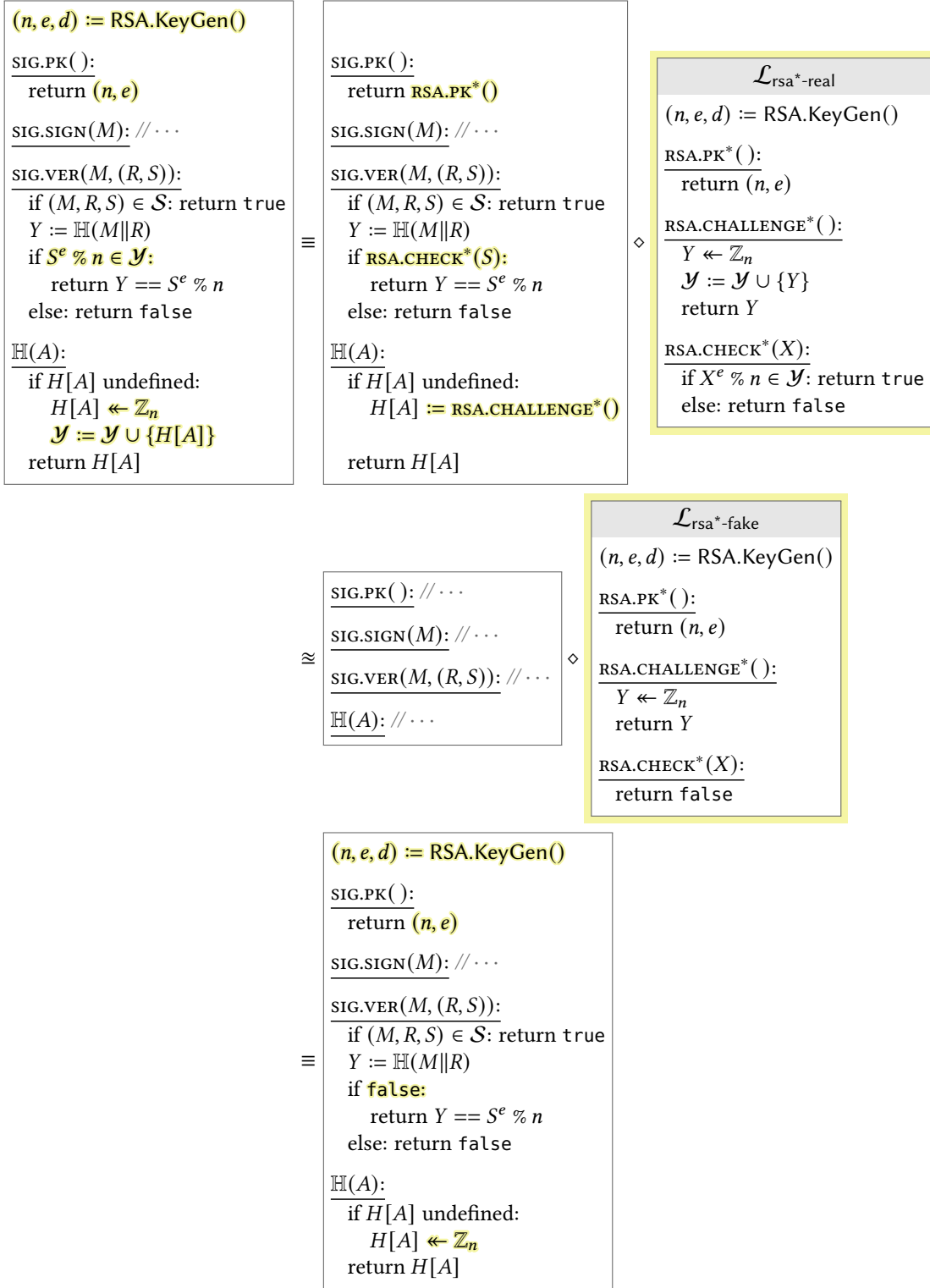
<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): // ... SIG.SIGN(M): R ← {0, 1}^λ H[M R] ← Z_n S := H[M R]^d % n S := S ∪ {(M, R, S)} return (R, S) SIG.VER(M, (R, S)): // ... H(A): // ... </pre>	≡	<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): // ... SIG.SIGN(M): R ← {0, 1}^λ S ← Z_n H[M R] := S^e % n S := S ∪ {(M, R, S)} return (R, S) SIG.VER(M, (R, S)): // ... H(A): // ... </pre>
--	---	--

When SIG.VER checks whether $Y = S^e$, it always uses a Y that was chosen uniformly in \mathbb{H} , not chosen with a known e -th root in SIG.SIGN (the if-statement in SIG.VER catches these cases). If we let \mathcal{Y} denote the set of all \mathbb{H} -outputs, then the condition $Y = S^e$ *implies* $S^e \in \mathcal{Y}$. Hence, it has no effect on the library to add the additional check $S^e \in \mathcal{Y}$ in SIG.VER.

<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): // ... SIG.SIGN(M): // ... SIG.VER(M, (R, S)): if (M, R, S) ∈ S: return true Y := H(M R) return Y == S^e % n H(A): if H[A] undefined: H[A] ← Z_n return H[A] </pre>	≡	<pre> (n, e, d) := RSA.KeyGen() SIG.PK(): // ... SIG.SIGN(M): // ... SIG.VER(M, (R, S)): if (M, R, S) ∈ S: return true Y := H(M R) if S^e % n ∈ Y: return Y == S^e % n else: return false H(A): if H[A] undefined: H[A] ← Z_n Y := Y ∪ {H[A]} return H[A] </pre>
---	---	--

The library no longer uses the private RSA exponent d ; it maintains a set \mathcal{Y} of uniformly sampled values; it later checks whether the adversary has provided an e -th root of *any* of these values. Hence, we can apply

Lemma 9:



After removing the now-unreachable if-statement, and reversing several earlier changes, the result is

$\mathcal{L}_{\text{sig-fake}}$. This completes the proof.

$\mathcal{L}_{\text{sig-fake}}$	
$(n, e, d) := \text{RSA.KeyGen}()$ $\text{SIG.PK}():$ return (n, e) $\text{SIG.SIGN}(M):$ $R \leftarrow \{0, 1\}^\lambda$ $S \leftarrow \mathbb{Z}_n$ $H[M\ R] := S^e \% n$ $\mathcal{S} := \mathcal{S} \cup \{(M, R, S)\}$ return (R, S) $\text{SIG.VER}(M, (R, S)):$ if $(M, R, S) \in \mathcal{S}$: return true $Y := H(M\ R)$ if false: return $Y == S^e \% n$ else: return false $H(A):$ if $H[A]$ undefined: $H[A] \leftarrow \mathbb{Z}_n$ return $H[A]$	$(n, e, d) := \text{RSA.KeyGen}()$ $\text{SIG.PK}():$ return (n, e) $\text{SIG.SIGN}(M):$ $R \leftarrow \{0, 1\}^\lambda$ $S := H(M\ R)^d \% n$ $\mathcal{S} := \mathcal{S} \cup \{(M, R, S)\}$ return (R, S) $\text{SIG.VER}(M, (R, S)):$ if $(M, R, S) \in \mathcal{S}$: return true return false $H(A):$ if $H[A]$ undefined: $H[A] \leftarrow \mathbb{Z}_n$ return $H[A]$

□

References

- [BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, Heidelberg, May / June 2006.
- [Ros] Mike Rosulek. The joy of cryptography. <https://joyofcryptography.com>.
- [Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. <https://eprint.iacr.org/2004/332>.