How to write proofs for cryptographic protocols at IETF

Christopher Patton (chrispatton+ietf@gmail.com)

July 21, 2024

Abstract

The security of many protocols at IETF rests on "pen-and-paper" proofs that are difficult for humans to verify. If the demand for provable security outpaces the capacity of human reviewers, then we run the risk of missing bugs in these proofs that may belie attacks. One way to prevent this is to mechanize portions of the review process. This essay sketches a vision for writing security proofs in which there is a clear division of labor between subject matter experts and experts in formal methods. This, we hope, will lead to more productive interactions across disciplines and to more of our proofs being fully vetted.

1 Introduction

This essay accompanies a talk for UFMRG at IETF 120. It is not a traditional research paper and is based largely on my own observations. All listings in this essay can be reproduced from code in the repository `https://github.com/cjpatton/better_proofs`. For feedback, feel free to file an issue against the repository, or send me an email.

Code-based game-playing proofs [3]. Game playing is a well-known paradigm for establishing computational security properties of cryptographic protocols. In this paradigm, the security goal and execution environment are modeled as a *game* played by the attacker. A *proof* of security relates the attacker's advantage in winning the game to the difficulty of one or more presumed hard problems, like cracking AES or finding short vectors on the lattice used by the protocol.

Games are often expressed in *code* rather than natural language, making them amendable to the following proof strategy. The proof consists of a finite sequence of games (sometimes called *hybrids*) beginning with the game that defines security. Each game is obtained by rewriting the code of the previous one until, in the final game, the attacker interacts with some idealized system that it has no hope of breaking.

The proof establishes that, for each pair of neighboring games, the probability of an attacker in distinguishing between them is negligible. This is done in various ways: sometimes a *rewrite* (also called a *hop* or *transition*) results in a game that is semantically equivalent, in which case no attacker can distinguish them; other rewrites result in distinguishable events that occur only with small probability, such as a collision in the range of a random oracle; still others are bounded computationally, by reducing some assumed hard problem to a distinguisher between the games.

In recent years, provable security in general and game-playing proofs in particular have played an increasingly central role in the design and analysis of protocols specified at IETF. We have seen working groups move from *reacting* to attacks to *proactively* ruling out (classes of) attacks during the design phase [13]. In fact, this process has even been formalized in the TLS working group [7]. While steps like these go a long way towards preventing attacks, the provable security methodology has some well-known limitations [4]. We consider one of them here.

Our crisis of rigor. In their seminal paper on the subject from Eurocrypt 2006 [3], Bellare and Rogaway assessed that many security proofs of the era were "essentially unverifiable". They feared the discipline would eventually reach a "crisis of rigor" in which proofs were believed without being fully vetted, allowing errors in these proofs to go unnoticed. And some of these errors may belie attacks, as has happened from time to time (see Section 2).

Today, carefully checking security proofs remains a significant challenge. There are at least three exacerbating factors:

1. Proofs are most often written in some *pseudocode* language intended for human readers. These are often called *pen-and-paper* proofs. There is no standard pseudocode in use today, and the grammar and semantics of these languages are as varied as the people that use them. Not only does this make mechanized verification nearly impossible, it also makes life difficult for human reviewers, many of whom have a stack of 10 papers or more to get through, each with its own unique semantics.

2. Protocols of interest to IETFers are becoming more complex. First, we are seeing our community do more with cryptography than ever before, often going beyond well-studied goals like key exchange, authentication, and encryption. Second, as attackers get more sophisticated, so must our security definitions capture the real execution environment in greater fidelity. Proofs are getting *longer* as a result, putting pressure on authors to merely "sketch" the proof in order to meet the page limit for the conference submission. Or they simply punt the proof to the paper's appendix, which program committee members are typically not required to review at all.

3. Mechanization of security proofs can help reduce burden on human reviewers. Several tools exist for this, like CryptoVerif, EasyCrypt, and SSProve, but these tools are inaccessible to many folks writing pen-and-paper proofs today. There are several reasons for this. First and foremost, cajoling the theorem prover into generating a proof is a manual process that requires some degree of expertise in the theory underlying the tool. Other problems include: the requirement to learn a new, somewhat esoteric programming language; great support for specific applications or use cases, but not for others; and the maturity of the software itself.

These factors put our community on a trajectory that, in my view, is not sustainable long term. I believe we must find ways to reduce the cost of checking security proofs so that the capacity of reviewers continues to meet the demands of those developing cryptographic protocols, at IETF and beyond.

Proposal. Our proofs ought to meet two basic requirements. First, they should be immediately amendable to mechanized verification so that human verifiers needn't check the full proof. Second, generating proofs should not require significantly more effort or expertise than pen-and-paper so that human provers are more inclined to flesh it out in full detail.

The key idea is simple: games, protocols, and proofs should all be written in a proper programming language rather than pseudocode. The language should have the following properties:

1. It should already be in wide use at IETF.

2. It should be high-level enough to be suitable as a specification language. That is is, the protocol and its intended security properties should be understandable without expertise in the language.

3. It should already be used for production software. This way the effort spent on specifying the protocol can translate more quickly into code that is used in practice.

This is not a particularly novel idea, yet it is not an idea that has taken root among most practitioners of provable security. Part of the problem is the lack of a consensus language and corresponding tools for verifying proofs. However, as I will try to illustrate here, this situation may be changing.

The benefit of writing game-playing proofs in actual code is that it creates a *separation of concerns*. On one end of the assembly line, the *subject matter expert* specifies the security game and protocol and generates the proof, consisting of the game rewrites and reductions. Ideally, this requires no knowledge of formal methods. On the other end of the assembly line, the *formal methods expert* verifies correctness of each of the rewrites. Ideally, this amounts to checking equivalence of programs written in the specification language and requires only limited knowledge of the application. Other benefits to this approach:

- Those of us IETFers who write security proofs as well as software will discover some surprising benefits of this approach. For example, imagine you have written a proof in pseudocode and find you need to go back and tweak the definition: the process of propagating that tweak through the proof (in practice, a pile of hand-written LaTeX) is both time-intensive and error-prone. Having a compiler with a type checker and a set of high-quality unit tests can make this much more sane.

- Cryptographers sometimes spend enormous energy working on definitions. Much of this time is spent probing for "trivial" distinguishers or winning conditions that suggest the attacker is too powerful or not captured in the right way. Writing such definitions in code makes it possible to formalize such conditions symbolically in tools like Tamarin or ProVerif.

Note that this proposal is grounded in extensive experience writing pen-and-paper proofs (sometimes with bugs!), but no professional experience with mechanized verification or other formal methods. It may be that I've massively underestimated how much work is being delegated to formal methods.

Outline. By way of illustrating this proposal, this essay presents a few elementary examples of security proofs from the cryptographic literature, transcribing them into a high-level programming language. These examples illustrate the kind of work carried out by the *subject matter expert*; the task of the *formal methods expert* is to prove each claim (written as "**Claim**") in the body of each proof.

Section 4 recalls the simple hybrid public-key encryption scheme and its security proof from Mike Rosulek's "Joy of Cryptography" [16]. Section 5 considers the PRP/PRF switching lemma, a bit of folklore that is well-known among cryptographers. These examples illustrate three common game rewrites: *equivalent-rewrite*, in which the new code is semantically equivalent (useful for setting up the next step in the proof); *equivalent-by-reduction*, in which an assumed hard problem is reduced to a distinguisher between the games; and *equivalent-until-bad*, in which the new game is equivalent up to some distinguishing, "bad" event.

We begin in Section 2 with a brief (and incomplete) overview of some documented cases of bugs in security proofs. We then describe our notation and conventions (Section 3) before diving into examples.

What about UC? While we focus on game-playing proofs in this essay, the same proposal can be applied to the *simulation paradigm*. Here security is defined relative to some *ideal functionality*, and we say the protocol *realizes* that ideal functionality if, for every attacker, there exists a simulator such that the adversary's interaction with the real system is indistinguishable from the simulator's interaction with the ideal functionality. There is no fundamental difference between simulation- and game-based definitions that makes mechanization of proofs in one paradigm harder than the other. However, the details of popular simulation frameworks, like the UC framework [6], can be rather involved, making mechanized proofs more difficult [5]. Games are a good place to start because the details of the execution model are always explicit.

2 Bugs in pen-and-paper proofs

OCB2. OCB2 ("Offset Code Book 2") is a symmetric, authenticated encryption scheme that was standardized by ISO in 2009 (ISO/IEC 19772:2009). In 2018 [9], an existential forgery attack against OCB2 was demonstrated that exploited a detail of the construction that was overlooked in the original security analysis. The attack is simple and efficient and, with a bit of work, can be improved into a total break of both authenticity and confidentiality. Note that its successor, OCB3 (RFC 7253), is not vulnerable to the same class of attacks.

GCM. GCM ("Galois Counter Mode") is a symmetric, authenticated encryption scheme that has been recommended by NIST since 2007 and is widely used at IETF and beyond. In 2012 [10], an efficient distinguisher was discovered that contradicts the concrete security claimed by the existing security proof. However, the attack does not result in a total break, and in fact the authors describe an alternative proof with slightly looser security. It does, however, impact the safety limits for GCM [8].

HMQV. The HMQV protocol is a well-known milestone in the evolution of authenticated key exchange, being one of the earliest such protocols with a provable security treatment. It is based on the MQV family of protocols (hence "Hashed MQV"), some of which appeared in various standards in the 1990s. Shortly after the proof was published (Crypto 2005 [11])j, the protocol was broken the same year [12]. The attacks exploit HMQV's omission of public-key validation, pointing to at least one flaw in the security proof where a public key controlled by the attacker was assumed to be valid. (See the preface of [11].)

OAEP. OAEP ("Optimzed Assymetric Encryption Padding") is a transformation of a trapdoor permutation into a public-key encryption scheme. The instantiation most widely used today, RSA-OAEP, was standardized in 1998 (RFC 2437). The original security proof (Eurocrypt 1994 [2]) was later shown by to be flawed [17]. The paper goes a bit further, proving there is no *black-box* reduction from a trapdoor permutation to the security of OAEP. However, the paper also shows that the flaw does not translate into an attack on RSA-OAEP in particular: the original proof can be corrected, but the corrected proof makes use of algebraic properties of RSA.

The author's own work. One of my own papers tries to formalize conditions under which key reuse across protocols is safe [14]. For example, consider the problem of securely using an already deployed ECDSA secret key as a static Diffie-Hellman key in another protocol. The main "result" of this paper gives a sufficient condition under which security of a given application is maintained in the presence of key reuse. The published version (Crypto 2019) contains a flaw related to the programmability of the random oracle by the simulator which invalidates the main theorem. This was discovered by an anonymous reviewer of a follow-up paper (Crypto 2020 [15]) in a more general context. Additional issues were pointed out a few years later, but as of this writing, I have not yet updated the paper to address them. (This is work in progress!)

3 Notation

All algorithms, including games and adversaries, are Rust (`https://www.rust-lang.org/`) programs without asynchronous semantics or unsafe code (i.e., with the **`async`**, **`await`**, and **`unsafe`** keywords removed). By convention, an adversary's runtime includes the time it takes to initialize its game and evaluate its oracle queries. Let `block` be a Rust code block whose last statement is an expression that evaluates to `bool`. We denote by $\Pr\left[\texttt{block}\right]$ the probability that execution of `block` halts (without panicking) and outputs `true`. We model `thread_rng()` from the `rand` crate as an ideal source of uniform random coins.

Why Rust? Rust easily meets two of our criteria from Section 1: it is widely used at IETF and in production software systems. Whether it is suitable as a specification language is debatable; I will try to make the case that it is, by conveying to the reader what elements of Rust are important to understanding definitions herein.

Another important reason for choosing Rust is that it is the specification language of the HAX toolchain (`https://cryspen.com/hax/`). HAX is used to transcribe Rust into one of many *backend* languages, such as F* or Coq, where existing tools can be applied for various purposes. HAX is gaining momentum,[1] and while it was not designed with this particular use case in mind, I have not yet found a good reason why it can't be, with some modifications.

4 Example: hybrid encryption

We first consider the hybrid public-key encryption (PKE) scheme of [16, Chapter 15]. The proof demonstrates two types of game transitions: *equivalent-rewrite* and *equivalent-by-reduction*.

Syntax. A PKE scheme implements the `PubEnc` trait listed in Figure 1. (A *trait* is similar to an interface or abstract base class in other languages that defines an API but not an implementation of the API.) Let `enc: E` implement `PubEnc`. (That is, `enc` is a PKE scheme because its type, denoted `E` implements the `PubEnc` trait. See Figure 2, line 5.) We say `enc` is *correct* if for all `m` $\in$ `E::Plaintext` it holds that

$$\Pr\left[\texttt{let (pk, sk) = enc.key\_gen(); enc.decrypt(sk, enc.encrypt(pk, m)) == Some(m)}\right] = 1\,.$$

[1] It was recently used to produce a verified, fast implementation of ML-KEM: see `https://cryspen.com/post/ml-kem-implementation/`.

```
/// Syntax.
pub trait PubEnc {
    type PublicKey;
    type SecretKey;
    type Plaintext;
    type Ciphertext;
    fn key_gen(&self) -> (Self::PublicKey, Self::SecretKey);
    fn encrypt(&self, pk: &Self::PublicKey, m: &Self::Plaintext) -> Self::Ciphertext;
    fn decrypt(&self, sk: &Self::SecretKey, c: &Self::Ciphertext) -> Option<Self::Plaintext>;
}

/// Security.
pub struct PubCpa<E: PubEnc> {
    enc: E,
    pk: E::PublicKey,
    right: bool,
}
impl<E: PubEnc> PubCpa<E> {
    pub fn init(enc: E, right: bool) -> Self {
        let (pk, _sk) = e.key_gen();
        Self { enc, pk, right }
    }
}
impl<E: PubEnc> GetPublicKey for PubCpa<E> {
    type PublicKey = E::PublicKey;
    fn get_pk(&self) -> &E::PublicKey {
        &self.pk
    }
}
impl<E: SymEnc> LeftOrRight for PubCpa<E> {
    type Plaintext = E::Plaintext;
    type Ciphertext = E::Ciphertext;
    fn left_or_right(&self, m_left: &E::Plaintext, m_right: &E::Plaintext) -> E::Ciphertext {
        if self.right {
            self.enc.encrypt(&self.k, m_right)
        } else {
            self.enc.encrypt(&self.k, m_left)
        }
    }
}
```

Figure 1: Syntax of PKE and game `PubCpa` for defining security under chosen plaintext attack (CPA).

Security. Security under chosen plaintext attack (CPA) is defined by the `PubCpa` game in Figure 1. It captures a left-or-right notion of indistinguishability by giving the attacker an oracle `LeftOrRight` that encrypts one of two messages chosen by the attacker and returns the ciphertext. (Each trait implemented by the game defines an oracle that the attacker has access to. One is `LeftOrRight`; the other is `GetPublicKey`, which grants access to the encryption key.) Which message is encrypted is controlled by a boolean `right` used to initialize the game (line 19).

Definition 1 ([16, Definition 15.1])**.** *Let A implement* `Distinguisher`*. Let* `enc` *implement* `PubEnc`*. Define the advantage of attacker A in breaking the security of* `enc` *under CPA as*

$$\mathrm{Adv}^{\mathrm{cpa}}_{\mathsf{enc}}(A) = \Pr\left[\, A.\texttt{play(PubCpa::init(enc, false))}\,\right] - \Pr\left[\, A.\texttt{play(PubCpa::init(enc, true))}\,\right],$$

where `PubCpa` *is as defined in Figure 1. Informally, we say* `enc` *is secure under CPA if every efficient attacker has small advantage.*

Remark 1. *This definition implicitly requires length hiding, since there is no requirement that the challenge plaintext* `m_left` *and* `m_right` *have the same length. For most PKE schemes, the length of the ciphertext is a function of the length of the plaintext.*

```
pub struct Hybrid<P, S> {
    pub pub_enc: P,
    pub sym_enc: S,
}
impl<P, S> PubEnc for Hybrid<P, S>
where
    S: SymEnc,
    P: PubEnc<Plaintext = S::Key>,
    Standard: Distribution<S::Key>,
{
    type PublicKey = P::PublicKey;
    type SecretKey = P::SecretKey;
    type Plaintext = S::Plaintext;
    type Ciphertext = (P::Ciphertext, S::Ciphertext);
    fn key_gen(&self) -> (P::PublicKey, P::SecretKey) {
        self.pub_enc.key_gen()
    }
    fn encrypt(&self, pk: &P::PublicKey, m: &S::Plaintext) -> (P::Ciphertext, S::Ciphertext) {
        let tk = thread_rng().gen(); // "temporary key"
        let c_pub = self.pub_enc.encrypt(pk, &tk);
        let c_sym = self.sym_enc.encrypt(&tk, m);
        (c_pub, c_sym)
    }
    fn decrypt(
        &self,
        sk: &Self::SecretKey,
        (c_pub, c_sym): &(P::Ciphertext, S::Ciphertext),
    ) -> Option<Self::Plaintext> {
        let tk = self.pub_enc.decrypt(sk, c_pub)?;
        let m = self.sym_enc.decrypt(&tk, c_sym)?;
        Some(m)
    }
}
```

Figure 2: Hybrid PKE scheme of [16, Construction 15.8].

Construction. In practice, PKE schemes are most often constructed by combining symmetric encryption with a mechanism for encapsulating the encryption key to the intended recipient [1] (that is, the holder of the secret key corresponding to the public encapsulation key). By way of gentle introduction to this paradigm, Mike Rosulek gives a construction of CPA-secure, hybrid encryption in "The Joy of Cryptography" [16]. We recall this construction in Figure 2.

This construction transforms a symmetric encryption scheme `sym_enc: S` into a PKE scheme. The syntax and security of symmetric encryption are defined similarly to PKE, except there is no public key; we defer the formal definition to Appendix A.1. The transformation involves a PKE scheme `pub_enc: E` with a small plaintext space. In particular, we require that plaintext space of `pub_enc` is the same as the key space for `sym_enc`. This is formalized by the trait bound `E: PubEnc<Plaintext = S::Key>` (see line 8).

To encrypt a plaintext, we choose a "temporary key" `tk` uniform randomly from the key space of `sym_enc`. We then encrypt `tk` using the PKE scheme `pub_enc`. Finally, we encrypt the plaintext under `tk` and output both ciphertexts.

Theorem 1 ([16, Claim 15.9])**.** *Suppose that* `pub_enc` *and* `sym_enc` *satisfy the constraints of Figure 2 and let* `hybrid` = `Hybrid{ pub_enc, sym_enc }`*. Then for every* `hybrid`*-attacker* A *there exist a* `sym_enc`*-attacker* B *and a* `pub_enc`*-attacker* C *with the same runtime as* A *for which*

$$\mathrm{Adv}^{\mathrm{cpa}}_{\mathsf{hybrid}}(A) \leq \mathrm{Adv}^{\mathrm{cpa}}_{\mathsf{sym\_enc}}(B) + 2 \cdot \mathrm{Adv}^{\mathrm{cpa}}_{\mathsf{pub\_enc}}(C)\,.$$

Proof. The proof is by a game-playing argument. We begin with game `G1` listed below:

```
pub struct G1<P: PubEnc, S> {
    enc: Hybrid<P, S>,
    pk: P::PublicKey,
}
```

```
impl<P: PubEnc, S> G1<P, S> {
    pub fn init(enc: Hybrid<P, S>) -> Self {
        let (pk, _sk) = enc.pub_enc.key_gen();
        Self { enc, pk }
    }
}
impl<P: PubEnc, S> GetPublicKey for G1<P, S> {
    type PublicKey = P::PublicKey;
    fn get_pk(&self) -> &P::PublicKey {
        &self.pk
    }
}
impl<P, S> LeftOrRight for G1<P, S>
where
    S: SymEnc,
    P: PubEnc<Plaintext = S::Key>,
    Standard: Distribution<S::Key>,
{
    type Plaintext = S::Plaintext;
    type Ciphertext = (P::Ciphertext, S::Ciphertext);
    fn left_or_right(
        &self,
        m_left: &S::Plaintext,
        _m_right: &S::Plaintext,
    ) -> (P::Ciphertext, S::Ciphertext) {
        let mut rng = thread_rng();
        let tk_left = rng.gen();
        let _tk_right = rng.gen(); // new temporary key (unused)
        let c_pub = self.enc.pub_enc.encrypt(&self.pk, &tk_left);
        let c_sym = self.enc.sym_enc.encrypt(&tk_left, m_left);
        (c_pub, c_sym)
    }
}
```

This game was obtained from PubEnc by fixing `right == false`, unrolling the call to `self.enc.encrypt()` in the `LeftOrRight` oracle, and adding code for generating a new temporary key, `_tk_right`. Let us call this new key (unused for the moment) the "right key" and the existing key the "left key".

Claim 1 (Equivalent-rewrite)**.** *For all A,*

$$\Pr\left[A.\texttt{play(PubCpa::init(hybrid, false))}\right] = \Pr\left[A.\texttt{play(G1::init(hybrid))}\right].$$

Next, game `G2` is obtained by applying the following patch to the `LeftOrRight` oracle. By this we mean removing the lines beginning with "-" and and adding the lines beginning with "+":

```
     fn left_or_right(
         &self,
         m_left: &S::Plaintext,
-        _m_right: &S::Plaintext,
+        m_right: &S::Plaintext,
     ) -> (P::Ciphertext, S::Ciphertext) {
          let mut rng = thread_rng();
          let tk_left = rng.gen();
-         let _tk_right = rng.gen(); // new temporary key (unused)
-         let c_pub = self.enc.pub_enc.encrypt(&self.pk, &tk_left);
+         let tk_right = rng.gen(); // new temporary key
+         let c_pub = self.enc.pub_enc.encrypt(&self.pk, &tk_right);
          let c_sym = self.enc.sym_enc.encrypt(&tk_left, m_left);
          (c_pub, c_sym)
      }
```

The patched oracle encrypts the right key rather than the left. The left key is still used to produce `s_sym`, so `c_sym` is now independent of `c_pub`.

We can reduce the attacker's ability to distinguish between `G1` and `G2` to the CPA security of `pub_cpa`. Consider the following reduction, which transforms a `hybrid` attacker into a `pub_enc` attacker. For the moment, fix `sim_right == false`; we will use `sim_right == true` in a later step.

```rust
pub struct FromHybridToPubEnc<P, S>
where
    P: PubEnc,
{
    sym_enc: S,
    game: PubCpa<P>, // Instance of the game for ‘pub_enc‘.
    sim_right: bool,
}
impl<P, S> FromHybridToPubEnc<P, S>
where
    P: PubEnc,
{
    pub fn init(game: PubCpa<P>, sym_enc: S, sim_right: bool) -> Self {
        Self {
            game,
            sym_enc,
            sim_right,
        }
    }
}
impl<P, S> GetPublicKey for FromHybridToPubEnc<P, S>
where
    P: PubEnc,
{
    type PublicKey = P::PublicKey;
    fn get_pk(&self) -> &P::PublicKey {
        self.game.get_pk()
    }
}
impl<P, S> LeftOrRight for FromHybridToPubEnc<P, S>
where
    P: PubEnc,
    S: SymEnc,
    P: PubEnc<Plaintext = S::Key>,
    Standard: Distribution<S::Key>,
{
    type Plaintext = S::Plaintext;
    type Ciphertext = (P::Ciphertext, S::Ciphertext);
    fn left_or_right(
        &self,
        m_left: &S::Plaintext,
        m_right: &S::Plaintext,
    ) -> (P::Ciphertext, S::Ciphertext) {
        let mut rng = thread_rng();
        let tk_left = rng.gen();
        let tk_right = rng.gen();
        // Simulation: If ‘self.left‘, then transition from ‘G1‘ to ‘G2‘;
        // otherwise, transition from ‘G3‘ to ‘G4‘. If ‘game.left‘, then
        // simulate the former; otherwise simulate the latter.
        let c_pub = self.game.left_or_right(&tk_left, &tk_right);
        let c_sym = if self.sim_right {
            self.sym_enc.encrypt(&tk_left, m_right)
    } else {
            self.sym_enc.encrypt(&tk_left, m_left)
        };
        (c_pub, c_sym)
    }
}
```

The reduction is initialized with an instance of the `PubCpa` game for `pub_enc`, denoted by `game`. It simulates the game for `hybrid`, except that instead of encrypting the left or right key directly, it invokes `game.left_or_right()` and uses the response to produce the ciphertext. (See line 50.) This way the reduction simulaters `G1` when `game.left == false` and `G2` otherwise.

Let $R_{12}^{A}(\texttt{game}) = A.\texttt{play(FromHybridToPubEnc::init(game), sym\_enc, false))}$. We wave the following.

Claim 2 (Equivalent-by-reduction). *For all A,*

$$\Pr\left[\, A.\texttt{play(G1::init(hybrid))} \,\right] = \Pr\left[\, R_{12}^{A}(\texttt{PubEnc::init(pub\_enc, false)}) \,\right]$$

and

$$\Pr\left[\, A.\texttt{play(G2::init(hybrid))} \,\right] = \Pr\left[\, R_{12}^{A}(\texttt{PubEnc::init(pub\_enc, true)}) \,\right].$$

Then by definition,

$$\Pr\left[\, A.\texttt{play(G1::init(hybrid))} \,\right] - \Pr\left[\, A.\texttt{play(G2::init(hybrid))} \,\right] \leq \mathrm{Adv}^{\mathrm{cpa}}_{\texttt{pub\_enc}}(R_{12}^{A}).$$

Next, we obtain `G3` from `G2` by patching the `LeftOrRight` oracle once more:

```
      fn left_or_right(
          &self,
-         m_left: &S::Plaintext,
-         _m_right: &S::Plaintext,
+         _m_left: &S::Plaintext,
+         m_right: &S::Plaintext,
      ) -> (P::Ciphertext, S::Ciphertext) {
          let mut rng = thread_rng();
          let tk_left = rng.gen();
-         let tk_right = rng.gen(); // new temporary key
+         let tk_right = rng.gen();
          let c_pub = self.enc.pub_enc.encrypt(&self.pk, &tk_right);
-         let c_sym = self.enc.sym_enc.encrypt(&tk_left, m_left);
+         let c_sym = self.enc.sym_enc.encrypt(&tk_left, m_right);
          (c_pub, c_sym)
      }
```

Now instead of encrypting the left plaintext, in the new game we encrypt the right plaintext. Because `c_pub` is independent of `c_sym`, we can reduce the adversary's advantage in distinguishing `G2` from `G2` to its advantage in breaking `sym_enc`. Consider the following reduction:

```
pub struct FromHybridToSymEnc<P, S>
where
    P: PubEnc,
    S: SymEnc,
{
    pub_enc: P,
    pk: P::PublicKey,
    game: Cpa<S>, // Instance of the game for ‘sym_enc‘
}
impl<P, S> FromHybridToSymEnc<P, S>
where
    P: PubEnc,
    S: SymEnc,
{
    pub fn init(game: Cpa<S>, pub_enc: P) -> Self {
        let (pk, _sk) = pub_enc.key_gen();
        Self { pub_enc, pk, game }
    }
}
impl<P, S> GetPublicKey for FromHybridToSymEnc<P, S>
where
    P: PubEnc,
    S: SymEnc,
{
    type PublicKey = P::PublicKey;
    fn get_pk(&self) -> &P::PublicKey {
        &self.pk
    }
}
impl<P, S> LeftOrRight for FromHybridToSymEnc<P, S>
where
    P: PubEnc,
    S: SymEnc,
```

```
    P: PubEnc<Plaintext = S::Key>,
    Standard: Distribution<S::Key>,
{
    type Plaintext = S::Plaintext;
    type Ciphertext = (P::Ciphertext, S::Ciphertext);
    fn left_or_right(
        &self,
        m_left: &S::Plaintext,
        m_right: &S::Plaintext,
    ) -> (P::Ciphertext, S::Ciphertext) {
        let tk_right = thread_rng().gen();
        let c_pub = self.pub_enc.encrypt(&self.pk, &tk_right);
        // Simulation: if `game.left`, then this simulates `G2`; otherwise, this
        // simulates `G3`.
        let c_sym = self.game.left_or_right(m_left, m_right);
        (c_pub, c_sym)
    }
}
```

This time the reduction gets an instance of the `Cpa` game for `sym_enc`. It simulates the `PubCpa` game for `hybrid` by generating a dummy temporary key `tk_right` and encrypting it to the public key generated at initialization. The symmetric ciphertext is produced by invoking `game.left_or_right(m_left, m_right)`; by construction, the encryption key is independent of `c_pub`, as it is in both games. Similar to the previous step, when `game.right == false`, the reduction simulates `G2` and simulates `G3` otherwise.

Let $R^A_{23}(\texttt{game}) = A.\texttt{play(FromHybridToSymEnc::init(game), pub\_enc))}$. We have the following.

Claim 3 (Equivalent-by-reduction)**.** *For all A,*

$$\Pr\left[A.\texttt{play(G2::init(hybrid))}\right] = \Pr\left[R^A_{23}(\texttt{Cpa::init(sym\_enc, false)})\right]$$

and

$$\Pr\left[A.\texttt{play(G3::init(hybrid))}\right] = \Pr\left[R^A_{23}(\texttt{Cpa::init(sym\_enc, true)})\right].$$

Then by definition,

$$\Pr\left[A.\texttt{play(G2::init(hybrid))}\right] - \Pr\left[A.\texttt{play(G3::init(hybrid))}\right] \leq \mathrm{Adv}^{\mathrm{cpa}}_{\texttt{sym\_enc}}(R^A_{23}).$$

Finally, we obtain `G4` from `G3` by applying the following patch:

```
    fn left_or_right(
         &self,
         _m_left: &S::Plaintext,
         m_right: &S::Plaintext,
     ) -> (P::Ciphertext, S::Ciphertext) {
         let mut rng = thread_rng();
-        let tk_left = rng.gen();
+        let _tk_left = rng.gen(); // no longer used
         let tk_right = rng.gen();
         let c_pub = self.enc.pub_enc.encrypt(&self.pk, &tk_right);
-        let c_sym = self.enc.sym_enc.encrypt(&tk_left, m_right);
+        let c_sym = self.enc.sym_enc.encrypt(&tk_right, m_right); // use right temporary key
         (c_pub, c_sym)
     }
```

In the new game, we encrypt the right temporary key instead of the left. Here again we bound the attacker's distinguishing advantage by reducing to CPA security of `pub_enc`. We can reuse `FromHybridToPubEnc`, except we fix `sim_right == true`. Let $R^A_{34}(\texttt{game}) = A.\texttt{play(FromHybridToPubEnc::init(game), sym\_enc, true))}$. We Have the following.

Claim 4 (Equivalent-by-reduction)**.** *For all A,*

$$\Pr\left[A.\texttt{play(G3::init(hybrid))}\right] = \Pr\left[R^A_{34}(\texttt{PubEnc::init(pub\_enc, false)})\right]$$

and

$$\Pr\left[A.\texttt{play(G4::init(hybrid))}\right] = \Pr\left[R^A_{34}(\texttt{PubEnc::init(pub\_enc, true)})\right].$$

```rust
/// Syntax.
pub trait Func {
    type Key;
    type Domain: ?Sized;
    type Range;
    fn eval(&self, k: &Self::Key, x: &Self::Domain) -> Self::Range;
}

/// Real game for defining PRP and PRF security.
pub struct Real<F: Func> {
    f: F,
    k: F::Key,
}
impl<F: Func> Real<F>
where
    Standard: Distribution<F::Key>,
{
    pub fn with(f: F) -> Self {
        let k = thread_rng().gen();
        Self { f, k }
    }
}
impl<F: Func> Eval<F> for Real<F> {
    fn eval(&mut self, x: &F::Domain) -> F::Range {
        self.f.eval(&self.k, x)
    }
}
```

Figure 3: Syntax and real security game for PRPs and PRFs. The ideal games are listed in Appendix A.2.

Then by definition,

$$\Pr\left[\, A.\texttt{play(G3::init(hybrid))} \,\right] - \Pr\left[\, A.\texttt{play(G4::init(hybrid))} \,\right] \leq \mathrm{Adv}^{\mathrm{cpa}}_{\texttt{pub\_enc}}(R^A_{34})\,.$$

Finally, game `G4` is equivalent to `PubCpa` with `right == true`.

Claim 5 (Equivalent-rewrite)**.** *For all A,*

$$\Pr\left[\, A.\texttt{play(G1::init(hybrid))} \,\right] = \Pr\left[\, A.\texttt{play(PubCpa::init(hybrid, true))} \,\right].$$

So far we have that

$$\mathrm{Adv}^{\mathrm{cpa}}_{\texttt{hybrid}}(A) \leq \mathrm{Adv}^{\mathrm{cpa}}_{\texttt{pub\_cpa}}(R^A_{12}) + \mathrm{Adv}^{\mathrm{cpa}}_{\texttt{sym\_cpa}}(R^A_{23}) + \mathrm{Adv}^{\mathrm{cpa}}_{\texttt{pub\_cpa}}(R^A_{34})\,.$$

Let $B = R^A_{23}$. Define C as follows. Flip a coin: if the outcome is `false`, then run R^A_{12}; if the outcome is `true`, then run R^A_{34}. Applying the law of total probability, we obtain

$$\mathrm{Adv}^{\mathrm{cpa}}_{\texttt{pub\_cpa}}(R^A_{12}) + \mathrm{Adv}^{\mathrm{cpa}}_{\texttt{pub\_cpa}}(R^A_{34}) = 2 \cdot \mathrm{Adv}^{\mathrm{cpa}}_{\texttt{pub\_cpa}}(C)\,.$$

And that's the way it is! □

5 Example: the PRP/PRF Switching Lemma

In this section we fix the PRP/PRF switching lemma [3], which states that a good pseudorandom permutation (PRP) is a good pseudorandom function (PRF) up to birthday attacks. The proof demonstrates the *equivalent-until-bad* transition.

Syntactically, both a PRP and PRF implement the `Eval` trait listed in Figure 3.[2] For example, the AES-128 blockipher would set `Key`, `Domain`, and `Range` to `[u8; 16]`. Security is defined by the `Real` game in

[2] A PRP also implements the inverse of `Eval::eval()`. We do not list this syntax because it is not needed here.

Figure 3, which gives the attacker an oracle for the function with a randomly sampled key. In the ideal PRP and PRF games, the oracle lazy evaluates, respectively, a random permutation and a random function. The ideal games, denoted `RandPerm` and `RandFunc`, are listed for completeness in Appendix A.2.

Let `f: F` implement `Func` and let A implement `Distinguisher` for `Eval<F>`.

Definition 2 (PRP). *Define the advantage of A in distinguishing `f` from a random permutation as*

$$\mathrm{Adv}^{\mathrm{prp}}_{\mathsf{f}}(A) = \Pr\left[\, A\texttt{.play(Real::with(f))} \,\right] - \Pr\left[\, A\texttt{.play(RandPerm::default())} \,\right].$$

(The `default()` method is provided by the `Default` trait, which the `RandPerm` trait derives automatically. The hash map and hash set are initially empty.) Informally, we say `f` is a PRP if every efficient attacker gets small advantage.

Definition 3 (PRF). *Define the advantage of A in distinguishing `f` from a random function as*

$$\mathrm{Adv}^{\mathrm{prf}}_{\mathsf{f}}(A) = \Pr\left[\, A\texttt{.play(Real::with(f))} \,\right] - \Pr\left[\, A\texttt{.play(RandFunc::default())} \,\right].$$

Informally, we say `f` is a PRF if every efficient attacker gets small advantage.

Lemma 1 ([3, Lemma 1]). *Suppose A makes at most q queries to its oracle. Then*

$$\mathrm{Adv}^{\mathrm{prp}}_{\mathsf{f}}(A) \le \mathrm{Adv}^{\mathrm{prf}}_{\mathsf{f}}(A) + \frac{q(q-1)}{2|\texttt{F::Range}|}.$$

Proof. We first observe that

$$\mathrm{Adv}^{\mathrm{prp}}_{\mathsf{f}}(A) = \mathrm{Adv}^{\mathrm{prf}}_{\mathsf{f}}(A) + \Pr\left[\, A\texttt{.play(RandFunc::default())} \,\right] - \Pr\left[\, A\texttt{.play(RandPerm::default())} \,\right].$$

We consider the difference $\Pr\left[\, A\texttt{.play(RandFunc::default())} \,\right] - \Pr\left[\, A\texttt{.play(RandPerm::default())} \,\right]$ in the remainder. We begin with the following game:

```rust
#[derive(Default)]
pub struct G1<F: Func>
where
    F::Domain: Sized,
{
    table: HashMap<F::Domain, F::Range>,
    range: HashSet<F::Range>,
    bad: bool,
}
impl<F: Func> Eval<F> for G1<F>
where
    Standard: Distribution<F::Range>,
    F::Domain: Clone + std::hash::Hash + Eq,
    F::Range: Clone + std::hash::Hash + Eq,
{
    fn eval(&mut self, x: &F::Domain) -> F::Range {
        let mut rng = thread_rng();
        self.table
            .entry(x.clone())
            .or_insert(loop {
                let y = rng.gen();
                if self.range.contains(&y) {
                    self.bad = true;
                }
                self.range.insert(y.clone());
                break y;
            })
            .clone()
    }
}
```

This game was obtained from `RandFunc` by modifying the closure passed to `.or_insert()`. The new code samples the range value as usual, but does some extra book-keeping that sets us up for the next step. In particular, the sampled point, denoted `y`, is stored in a set `range`: if ever `y` is already in `range`, then the game sets a flag called `bad`, but otherwise proceeds as usual.

Claim 6 (Equivalent-rewrite)**.** *For all A,*

$$\Pr\left[\, A.\texttt{play(RandFunc::default())} \,\right] = \Pr\left[\, A.\texttt{play(G1::default())} \,\right].$$

Next, game `G2` is obtained from `G2` by applying the following patch:

```
    fn eval(&mut self, x: &F::Domain) -> F::Range {
        let mut rng = thread_rng();
        self.table
            .entry(x.clone())
            .or_insert(loop {
                let y = rng.gen();
                if self.range.contains(&y) {
                    self.bad = true;
+                   continue; // reject and try again
                }
                self.range.insert(y.clone());
                break y;
            })
            .clone()
    }
```

This modifies the code after `bad` gets set. In the new game, if there is a collision in the range, then `y` is rejected and we try again. Let $bad = A.\texttt{play\_then\_return(G2::default()).bad}$.

Claim 7 (Equivalent-until-bad)**.** *For all A,*

$$\Pr\left[\, A.\texttt{play(G1::default())} \,\right] - \Pr\left[\, A.\texttt{play(G2::default())} \,\right] \leq \Pr\left[\, bad \,\right].$$

Finally, game `G2` is equivalent to `RandPerm`. Namely, the latter is obtained from the former by negating the condition and re-arranging the code.

Claim 8 (Equivalent-rewrite)**.** *For all A,*

$$\Pr\left[\, A.\texttt{play(G2::default())} \,\right] = \Pr\left[\, A.\texttt{play(RandPerm::default())} \,\right].$$

To obtain the final bound, observe that the *bad* event occurs if, at any point in the game's execution, it holds that `y` is contained in `range`. This can occur on any `Eval` query, of which there at at most q. Since each range point is sampled uniform randomly, we have that

$$\Pr\left[\, bad \,\right] \leq \binom{q}{2} \cdot \frac{1}{|\texttt{F::Range}|} = \frac{q(q-1)}{2|\texttt{F::Range}|}.$$

And so it was. □

References

[1] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. RFC 9180, February 2022.

[2] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, pages 92–111, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[3] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Paper 2004/331, 2004. `https://eprint.iacr.org/2004/331`.

[4] Daniel J. Bernstein. Comparing proofs of security for lattice-based encryption. Cryptology ePrint Archive, Paper 2019/691, 2019. `https://eprint.iacr.org/2019/691`.

[5] R. Canetti, A. Stoughton, and M. Varia. Easyuc: Using easycrypt to mechanize proofs of universally composable security. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 167–16716, Los Alamitos, CA, USA, jun 2019. IEEE Computer Society.

[6] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Paper 2000/067, 2000. https://eprint.iacr.org/2000/067.

[7] Deirdre Connolly. Kicking off the TLS 1.3 formal analysis triage panel. TLS mailing list: https://mailarchive.ietf.org/arch/msg/tls/FhhICSR3qlLHjFcf1cA7Iry11CA/, 2024.

[8] Felix Günther, Martin Thomson, and Christopher A. Wood. Usage Limits on AEAD Algorithms. Internet-Draft draft-irtf-cfrg-aead-limits-08, Internet Engineering Task Force, April 2024. Work in Progress.

[9] Akiko Inoue, Tetsu Iwata, Kazuhiko Minematsu, and Bertram Poettering. Cryptanalysis of OCB2: Attacks on authenticity and confidentiality. Cryptology ePrint Archive, Paper 2019/311, 2019. https://eprint.iacr.org/2019/311.

[10] Tetsu Iwata, Keisuke Ohashi, and Kazuhiko Minematsu. Breaking and repairing GCM security proofs. Cryptology ePrint Archive, Paper 2012/438, 2012. https://eprint.iacr.org/2012/438.

[11] Hugo Krawczyk. HMQV: A high-performance secure diffie-hellman protocol. Cryptology ePrint Archive, Paper 2005/176, 2005. https://eprint.iacr.org/2005/176.

[12] Alfred Menezes. Another look at HMQV. Cryptology ePrint Archive, Paper 2005/205, 2005. https://eprint.iacr.org/2005/205.

[13] Kenneth G. Paterson and Thyla van der Merwe. Reactive and proactive standardisation of tls. In Lidong Chen, David McGrew, and Chris Mitchell, editors, *Security Standardisation Research*, pages 160–186, Cham, 2016. Springer International Publishing.

[14] Christopher Patton and Thomas Shrimpton. Security in the presence of key reuse: Context-separable interfaces and their applications. Cryptology ePrint Archive, Paper 2019/519, 2019. https://eprint.iacr.org/2019/519.

[15] Christopher Patton and Thomas Shrimpton. Quantifying the security cost of migrating protocols to practice. Cryptology ePrint Archive, Paper 2020/573, 2020. https://eprint.iacr.org/2020/573.

[16] Mike Rosulek. The joy of cryptography. https://joyofcryptography.com.

[17] Victor Shoup. OAEP reconsidered. Cryptology ePrint Archive, Paper 2000/060, 2000. https://eprint.iacr.org/2000/060.

A Deferred definitions

A.1 Symmetric encryption

A symmetric encryption scheme implements the `SymEnc` trait listed in Figure 4. Let `enc: E` implement `SymEnc`. We say `enc` is *correct* if for all `m` $\in$ `E::Plaintext` it holds that

$$\Pr\left[\texttt{let k = e.key\_gen(); enc.decrypt(k, enc.encrypt(k, m)) == Some(m)}\right] = 1\,.$$

Security under chosen plaintext attack (CPA) is defined by the `Cpa` game in Figure 4.

Definition 4 ([16, Definition 7.1])**.** *Let A implement* `Distinguisher`*. Let* `enc` *implement* `SymEnc`*. Define the advantage of A in breaking the security of* `enc` *under CPA as*

$$\mathrm{Adv}^{\mathrm{cpa}}_{\texttt{enc}}(A) = \Pr\left[A.\texttt{play(Cpa::init(enc, false))}\right] - \Pr\left[A.\texttt{play(Cpa::init(enc, true))}\right].$$

Informally, we say `enc` *is secure under CPA if every efficient adversary has small advantage.*

```rust
/// Syntax.
pub trait SymEnc {
    type Key;
    type Plaintext;
    type Ciphertext;
    fn key_gen(&self) -> Self::Key;
    fn encrypt(&self, k: &Self::Key, m: &Self::Plaintext) -> Self::Ciphertext;
    fn decrypt(&self, k: &Self::Key, c: &Self::Ciphertext) -> Option<Self::Plaintext>;
}

/// Security.
pub struct Cpa<E: SymEnc> {
    e: E,
    k: E::Key,
    right: bool,
}
impl<E: SymEnc> Cpa<E> {
    pub fn init(e: E, right: bool) -> Self {
        let k = e.key_gen();
        Self { e, k, right }
    }
}
impl<E: SymEnc> LeftOrRight for Cpa<E> {
    type Plaintext = E::Plaintext;
    type Ciphertext = E::Ciphertext;
    fn left_or_right(&self, m_left: &E::Plaintext, m_right: &E::Plaintext) -> E::Ciphertext {
        if self.right {
            self.enc.encrypt(&self.k, m_right)
        } else {
            self.enc.encrypt(&self.k, m_left)
        }
    }
}
```

Figure 4: Syntax of symmetric encryption and game `Cpa` for defining security under CPA.

A.2 Ideal games for defining PRPs and PRFs

```rust
/// Ideal game for defining PRP security.
#[derive(Default)]
pub struct RandPerm<F: Func>
where
    F::Domain: Sized,
{
    table: HashMap<F::Domain, F::Range>,
    range: HashSet<F::Range>,
}
impl<F: Func> Eval<F> for RandPerm<F>
where
    Standard: Distribution<F::Range>,
    F::Domain: Clone + std::hash::Hash + Eq,
    F::Range: Clone + std::hash::Hash + Eq,
{
    fn eval(&mut self, x: &F::Domain) -> F::Range {
        let mut rng = thread_rng();
        self.table
            .entry(x.clone())
            .or_insert(loop {
                let y = rng.gen();
                if !self.range.contains(&y) {
                    self.range.insert(y.clone());
                    break y;
                }
            })
            .clone()
```

```rust
    }
}

/// Ideal game for defining PRF security.
#[derive(Default)]
pub struct RandFunc<F: Func>
where
    F::Domain: Sized,
{
    table: HashMap<F::Domain, F::Range>,
}
impl<F: Func> Eval<F> for RandFunc<F>
where
    Standard: Distribution<F::Range>,
    F::Domain: Clone + std::hash::Hash + Eq,
    F::Range: Clone + std::hash::Hash + Eq,
{
    fn eval(&mut self, x: &F::Domain) -> F::Range {
        self.table
            .entry(x.clone())
            .or_insert(thread_rng().gen())
            .clone()
    }
}
```

A.3 Traits

```rust
/// A generic distinguishing attacker.
pub trait Distinguisher<G> {
    fn play_then_return(&self, game: G) -> (G, bool);
    fn play(&self, game: G) -> bool {
        // Return the `bool` only.
        self.play_then_return(game).1
    }
}

/// Left-or-right encryption oracle.
pub trait LeftOrRight {
    type Plaintext;
    type Ciphertext;
    fn left_or_right(
        &self,
        m_left: &Self::Plaintext,
        m_right: &Self::Plaintext,
    ) -> Self::Ciphertext;
}

/// Oracle for obtaining the public key in a public-key cryptosystem.
pub trait GetPublicKey {
    type PublicKey;
    fn get_pk(&self) -> &Self::PublicKey;
}

/// Oracle for evaluating a keyed function.
pub trait Eval<F: Func> {
    fn eval(&mut self, x: &F::Domain) -> F::Range;
}
```