# Designing Secure Cryptography

Cornell CS 6831

## Notes

≪**TCR 0.1:** A couple notes for myself, will remove later.≫

- Fix a pseudocode language, including randomness, and an associated model of computation. Treat memory usage as well as run times of algorithms. We will most often treat memory as subservient to run-time, bounding the former by the latter. But special treatments are needed. The goal is to have a simplified abstraction, but which provides good predictions of running code. Probably look at "Careful with Composition" paper as starting point.
- Algorithms are simply pseudocode that take input and produce an output. Define what it means for algorithm to be runnable. Define adversaries as algorithms and require they be runnable. Comment on how this interacts with (and rules out) non-uniform reductions, other such things.
- Algorithms can be parameterized, but this is just notational sugar. The result must still be runnable. They may have oracles, and this defines some interface. Give conventions on resource usage of algorithms: number of oracle queries, run time (with oracle queries unit cost), etc. Discuss how runnable algorithms can then be composed.
- Associate probability space to any algorithm, be pedantic here since we may want to prove some basic stuff via direct manipulation of probability space (i.e., coin-counting arguments).
- Introduce security definitions as special algorithms called games. The game is just another algorithm parameterized by some adversary, scheme, etc. Thus typically what we define as a game is actually a family of algorithms, one for each instantiation of things. (How much notion of a template do we need? Is this confusing?) It is used to measure adversarial advantage, a which is a measure of success for an adversary.
- Discuss the viewpoint underlying all the above. Security models are coarse abstractions of real cryptographic systems. They must be coarse for us to do certain kinds of analysis. This means that what we prove in our formalizations do not typically apply to real systems. But they are a good heuristic: schemes for which we have good analyses tend to provide better security. We will therefore judge the value of models by their utility in helping us build cryptographic schemes that resist attackers in practice. We will include examples and discussions reinforcing this viewpoint along the way. (One example to treat right away is asymptotics. Could be good heuristic, but is often quite poor, and gives little advice about how to set security parameters meaningfully.)
- Discuss assumptions. These are formalized as games. What separates them from security definitions is contextual. They are "lower level" and less related to the goals of cryptographic protocols.

- Reductions are runnable algorithms.
- Using reductions to set security parameters. Maybe interleave some of this discussion with first couple of chapters

# 1 Introduction

The goal of this chapter is to describe the *perspective* taken by these notes. This perspective guides our exploration of cryptography in both overt and subtle ways, and I think it's worth dwelling on for a bit — successful scholars should necessarily be introspective about why they approach problems in certain ways, and me writing this down is such an exercise for myself.

Our focus will be on cryptography as a *tool* for securing contemporary and future information systems. Being in the digital age, this means computers and their associated communication networks. One tension we will face is how much of contemporary technology must we consider to build good cryptography, the details of which change rather rapidly, versus a more mathematical abstraction that has longer staying power. We will endeavor to strike a balance, building lasting abstractions born out of contemporary technology issus.

**Classical and modern cryptography.** A frequently espoused viewpoint is that one can divide between modern cryptography and classical cryptography. The tipping point being perhaps the seminal work of Shannon in the 1940s [6], or that of Goldwasser and Micali [5] (and their contemporaries) or Dolev and Yao [3] in the 1980s. In either case the idea is that these works introduced a key new element into cryptography: proofs of security. Previous to these works, in classical cryptography, one built cryptographic algorithms and protocols and informal arguments of their security, for some often vague notion of security. Indeed early works talk about the inability to recover a message from a ciphertext without the decryption key, without really specifying what are the characteristics of these messages, how keys are chosen, and more.

Shannon in his work provided a beautiful mathematical definition of security for symmetric (or secret key) encryption, called perfect secrecy. Roughly it states that given a ciphertext, the probability, taken over the choice of secret key, that it is an encryption of any message is the same. It's simple to state and intuitive, and provides seemingly the best possible security, hence the name. In the 1980s two different viewpoints emerged. On one hand, following Goldwasser and Micali, arose the view that we should take into account computational complexity. Roughly stated for Shannon's symmetric encryption setting, we should only care that no computationally efficient adversary can learn something about messages given ciphertexts. Computational efficiency was measured in asymptotic terms, though Bellare and Rogaway pioneered a concrete security approach that enabled a more granular accounting of computational resources in a sequence of papers in the 1990s [1]. This line of work built off techniques from computational complexity theory, including the use of manually written reductions relating difficulty of an adversary breaking a scheme to the difficulty of breaking some underlying believed-to-be computationally hard problem. In parallel a body of work built off Dolev and Yao's ideas, in which proofs that dispensed with computational complexity, primarily treated lower level primitives as "perfect" symbolic operations, and utilized techniques from the theory of programming languages. It is of historical and sociological interest that the two threads of research built two largely disjoint academic communities (researchers, conference venues, etc.).

Taken together, the use of precise security goals stated in mathematical language and frameworks for proving schemes protocols secure represents a big shift in the way cryptography has been analyzed. In the last 40 years especially, we have seen a blossoming of public work in cryptography focused on proving security, with a vibrant academic community focusing on it.

**The ground truth of (in)security.** Security is inherently normative and context-dependent. We will take a pragmatic viewpoint on evaluating security: Are real attacks being foiled? By real attacks, I mean evidenced cases in which actors have sought to, or more likely, achieved harm against some computing system. We will have to use our normative judgement on what is harm,

but the point here is that, for the researcher, we will attempt to steer our design and evaluation of cryptography towards practical security issues. Cryptosystem A will dominate cryposystem B if we can point to real attacks that it foils while cryptosystem B only provides some hypothetical improvements. Evidence of real attacks can be garnered through word-of-mouth discussions with practitioners, or even research into attackers motives and means.

Unfortunately judging efficacy is not yet formulaic, and coming up with rigorous ways to predict whether cryptographic designs prevent attacks represents an opportunity area for much future research.

**Methodoligical regimes.** Towards better predictions, we want principled approaches. One can roughly group together various methodological regimes cryptographers use to evaluate whether a cryptographic design will resist attack. Examples include:

- Cryptanalysis (here I mean the study of blockciphers, hash functions, number-theoretic primitives, and other low-level cryptographic primitives)
- Developing attacks (against algorithms and protocols, or implementations of them)
- Algorithm verification (such as Dolev-Yao type analyses) or even software verification
- Reduction-based analysis (very often called provable security)
- Empiricism (understanding cryptographic system behavior in the real world)
- User studies (to determine if cryptographic systems easy implement and use securely)

Each regime offers instruments for scrutiny of cryptography, and they each have their strengths and weaknesses. If one wants to understand how attackers can exploit an implementation of a cryptosystem, it's probably not best to start with verification or reduction-based security — start by playing the role of an attacker and applying your experience and ingenuity. On the other hand, if you want to rule out attacks that violate the lower-level structure of the cryptographic algorithms, then using reduction-based analysis to show that the design appears to be secure as long as some underlying hard problem remains so. If you want to understand if problems are hard, you're back to the body of cryptanalytic work attempting to build fast algorithms for factoring, discrete log, or finding collisions in hash functions. If you have an interactive protocol for which manual analysis exceeds typical expert human capacity, then you should probably start looking towards symbolic analysis tools.

In line with our utilitarian viewpoint, you picked the right tool not because of success measured by aesthetic concerns such as the cleverness of a demonstrated attack, the elegance of a reduction, or improvements in type theory, but rather because it helped you build a system resisting the types of attacks seen in practice. To really gain confidence should most typically use a combination of methodological regimes to gain confidence in a design, such as was used in a large effort for the recent TLS 1.3 specification, which used a combination of almost all the regimes above. A sophisticated cryptographer will of course, appreciate and try to understand the gaps between different methodological regimes – the attacks that they don't, in aggregate, rule out.

Unfortunately, I don't know much about approaches — there are only so many hours in one's life — and so many will sadly only get cursory discussion. I will focus a lot on reduction-based analysis in the concrete-security tradition, developing attacks, and empiricism, the topics for which I have some experience. They will serve as good examples to understand the rough boundaries between different methodological regimes and the fascinating issues that arises at those boundaries.

**The power and pitfalls of abstraction.** A key element to any security analysis is abstraction, and security analyses use abstraction aggressively to ensure analytical tractability. Abstractions in our context is an exercise in *modeling* of cryptographic settings. For example, we will spend a

lot of time developing security games that capture desirable security properties for cryptographic algorithms.

Modeling is hard. The reason is we must simultaneously balance the need for simplicity of an abstraction with the danger of omitting security-critical details. As a ready example, almost all of our modeling will not countenance side-channel attacks such as those emanating from analyzing power traces or microarchitectural nuances. Would we be better off with models that do try to take these into account? In most cases the seeming answer is 'no', because the models would become too complicated to allow useful analysis.

We therefore advocate an iterative approach to modeling that roughly goes as follows:

(1) Identify normative security concerns. Or put more simply: what attacks do we believe people care about preventing?

(2) Develop one or more cryptographic security models that (hopefully) speak to these attacks. Analyze different models formally by comparing them. Develop cryptographic systems that we can show analytically are secure in the model.

(3) Return to normative security goals. Can we break our proposed schemes in some normatively damaging way?

I use the term normative here to indicate the fact that, ultimately, security is a social construct and we are driven by people's desires. We won't, in this class, put a significant amount of attention on tools for wading through the ethical and sociological questions of how we should guide our normative concerns — that requires philosophy.[1] That said, we can identify interesting security concerns via intuition, experience, and keeping tabs on socially important topics. My favorite approach is to pay close attention to what problems practitioners are facing — what attacks have arisen in practice that cryptography might solve? Develping the ability to identify such opportunities can be a good way to build a career of impactful research.

Once normative concerns are identified, we can attend to how we can address them as rigorously as possible. Here we come up with security models, often called security definitions. We will build off the tradition of what's called provable security cryptography that provides definitions in the way of probabilistic experiments. We will discuss how to formalize in subsequent chapters. Within these abstractions we will also be able to define absractions of cryptographic tools: these are the pseudocode versions of real software or hardware. These serve as a specification of sorts, though often these are, like security models, woefully lacking in details from the perspective of an implementor. A key issue will be attending to this tension between level of detail and modeling simplicity.

We will then show how to do various formal analyses in these abstract models. The approach will be hand-written proofs, in a mathematical style. They will most often be reduction-based and we will focus on ensuring, whenever possible, that analyses result in actionable information about the security of a proposed cryptographic algorithm.

We will then return to understanding limitations of what we've produced by way of developing attacks against normative security goals. Often we can iteratively develop formal models to attend to these attacks, helping suss out subtleties missed in earlier iterations. However, at some point we expect that more modeling will have diminishing returns: either the normative attacks can't be found or seem more easily dispensed with via informal guidance (e.g., use padding when necessary to obscure lengths, use constant-time code).

Given our utilitarian viewpoint, we will want, throughout, to attend to practicality: Are the cryptographic algorithms we arrive at reasonable for use? Are they fast enough for practice? Will

[1] One might read Rogaway's paper on this topic for further pointers `http://web.cs.ucdavis.edu/~rogaway/papers/moral-fn.pdf`.

they be fragile to implement? If the answers are no, that doesn't necessarily mean that some effort should be discarded — it may be fascinating theory that has value for the future. But we should at least be able to articulate how far from practical some proposal is, and why. This often requires experimentation, measurement, and dealing with often messy issues such as legacy requirements or psychological acceptability by developers.

The viewpoints above may stand in stark contrast to the rhetoric of provable security more often used in modern treatments of cryptography. What people refer to as provable security is just the step (2) above, but I try to avoid this terminology because I think (and experience indicates) that it oversells and may often under-deliver. Proofs can have errors. Models can become so complex they have limited analytical utility and using them leads to mistakes or overly complex protocols that are hard to implement. In any case, while for theoreticians provable security is a meaningful term, for cryptography meant to be secure in practice, it risks being misleading. Nothing is so black-and-white in practice.

This is not to say that theory, modeling, and formal analysis aren't critical to applied cryptography. Most obviously we will be building off the deep literature from theoretical cryptography, and without that theory we would be lost. More directly, theory provides us powerful tools to help us rule out large classes of attacks for deployed schemes, that complement and strengthen the one-at-a-time approach of simple attack-driven investigations. One can visualize this process for a particular scheme as shown in Figure 1, which means to depict the universe of all possible attacks and those attacks covered by different analysis approaches. Each point in this universe represents an implementable attack against an instantiation of the system, which may or may not succeed, and the goal of the analyst is to rule out the efficacy of as many such attacks as possible. The black dots represent one-at-a-time investigation ruling out individual attacks, the green circle might represent formal reduction-based analyses, the blue circle symbolic analyses, and the red circle an analysis of some class of side-channel attacks. Of course this is highly abstracted, but gives a sense of the thinking: formal modeling and analysis can help rule out larger classes of attacks, but certainly will miss things that fall outside the scope of the model. By combining approaches in a thoughtful manner combined with normative intuition, one achieves better coverage and can spot attack strategies that may be more likely to work.

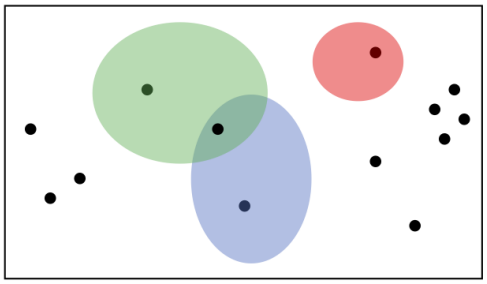

Figure 1: The universe of possible attacks against a cryptographic scheme or protocol.

To make this kind of approach work, though, we must have a mastery of modeling and formal analysis tools. Gaining that mastery is the subject of this course.

## 2 Preliminaries and Notation

We collect here some notation that we will use throughout these notes. ≪**TCR 2.1:** If you introduce new notation in writing up a lecture, please consider adding it here.≫

**Basics.** We most often denote sets by calligraphic, capital letters, such as $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$. A discrete probability distribution is a set $\Omega$, the event space, together with a function $p\colon \Omega \to [0,1]$ for which $\sum_{\omega\in\Omega} p_\Omega(\omega) = 1$. We write $p_\Omega$ when we need to disambiguate the event space, but otherwise simply $p$ when $\Omega$ is clear from context. We will also, following convention, often write $\Pr[\omega]$ instead of $p(\omega)$. The support of $p$ is defined to be the set $\mathrm{Supp}(p) = \{\omega \mid \omega \in \Omega \wedge p(\omega) > 0\}$, i.e., the set of all points in $\Omega$ that have non-zero probability. As a slight abuse of notation, we can write $\Pr[\mathcal{S}] = p(\mathcal{S}) = \sum_{\omega\in\mathcal{S}} p(\omega)$ for any set $\mathcal{S} \subseteq \Omega$.

A random variable $X$ is a map $Y\colon \Omega \to \mathcal{X}$ from some event space $\Omega$ with associated probability distribution $p_\Omega$ over a set $\Omega$. So for some other set $\mathcal{S}$ we let $\Pr[X \in \mathcal{S}] = \Pr[\{\omega \mid X(\omega) \in \mathcal{S}\}]$ to be the probability that the random variable takes on a value in $\mathcal{S}$, over the probability distribution $p_\Omega$.

We use the language of probability as the foundation for formalizing cryptographic algorithms, security, and more. Interestingly the probability spaces involved get complicated quickly, and a common problem is that they end ambiguous. We will therefore rely on a crutch that has proved quite useful for communicating, and reasoning about, probability distributions.

**Pseudocode games.** We fix some pseudocode language to describe security models, cryptographic algorithms, and more. Roughly we will follow the notational tradition established by Bellare and Rogaway in the 2000s [2], but using slightly different syntax/symantecs that are based most closely on a treatment from [4]. Code-based games are convenient for more precisely defining probability spaces, which are what we use to model security and correctness goals, algorithms, and more.

We will use procedures, variables, and typical programming statements (such as operators, loops, procedure calls, etc.). Typical statements are shown in Figure 2. We do not provide a formal specification of the programming language, see [2, Appendix B] for an example of doing so. We will rely on some conventions to help make sense of games. Types should be understable from context. The names of syntactic objects (procedures, variables, etc.) must be distinct. Variables are implicitly initialized to default values: integer variables are set to 0, arrays are everywhere $\perp$, etc. Here $\perp$ is a distinguished symbol by tradition used to denote an error in the cryptographic literature. By distinguished we mean that it assumed to not be used for any other reason, not appearing in alphabets over which strings are taken, etc.

A *procedure* is a sequence of statements together with zero or more variable inputs and zero or more outputs. An *unspecified procedure* is a procedure whose pseudocode, inputs, and outputs are understood from context. We will see some examples of unspecified procedures, the most frequent in our security games being the *adversary* which is often left unspecified. A call to a procedure requires providing it with inputs, running its sequence of statements, and returning its output. We will interchangeably use the term call and *query* for procedure invocation. A procedure $P$ can itself query other procedures. The set of procedures $Q_1, \ldots, Q_k$ called by a procedure are statically fixed, and we require that there are no type mismatches in inputs

| | |
|---|---|
| $x \leftarrow y$ | Assignment |
| $x \xleftarrow{\$} \mathcal{X}$ | Uniform sampling from a set |
| $z \xleftarrow{\$} P(x,y)$ | Call a randomized procedure |
| $z \leftarrow P(x,y)$ | Call a deterministic procedure |
| Ret $x$ | Return from a procedure |
| $z \leftarrow x \,\|\, y$ | String concatenation |
| $(x,y) \xleftarrow{n} z$ | Parse string $z$ s.t. $\|x\| = n$ |

Figure 2: Some statements used in our games.

and outputs.

Say that the code of $P$ expects to be able to call $k$ distinct procedures. We will write $P^{Q_1,Q_2,\ldots,Q_k}$ to denote that these calls are handled by $Q_1, Q_2, \ldots, Q_k$ and implicitly assume (for all $i \in [k]$) that there are no syntactic mismatches between the calls that $P$ makes to $Q_i$ and the inputs of $Q_i$, as well as between the return values of $Q_i$ and the return values expected by $P$.

We assume that all procedures eventually halt, regardless of randomness used, returning their outputs, at which point execution returns to the calling procedure. Two procedures $P_1$ and $P_2$ are said to *export the same interface* if their inputs and outputs have the same number and types.

Variables will be local by default, meaning they can only be used within a single procedure. Variables are static, meaning that they retain their state between calls to the procedure. It will be convenient to allow sharing of variables at times, for which we use a *collection of procedures*. This is a set of one or more procedures whose variables have scope covering all of the procedures. We will denote a collection of procedures using a common prefix ending with a period, so for example $(P.x, P.y, \ldots)$, Sometimes we will use the term interfaces for the specific prefixes of the a collection of procedures $P$.

A *main procedure* is a special procedure that takes no inputs and has some output. We will mark it by **main** (though below we'll see some syntactic sugar that provides greater brevity). No procedure may call **main**, it can access all the variables of other specified procedures (though not unspecified procedures).

A game consists of a main procedure together with a set of zero or more specified procedures. We write G for a game. A game may also make use of unspecified procedures (such as adversaries), which we enumerate as superscripts, e.g., $\mathrm{G}^{P_1,P_2,\ldots,P_k}$. In most games used as security definitions, one (or more) of the unspecified procedures will be called the adversary, most often denoted by $\mathcal{A}$. For a given instantiation of the unspecified procedures, one can run a game: execute its statements starting with the designated **main** procedure, and ultimately outputing whatever **main** returns.

**Run times and random variables.** Games can make random choices, due to the supported statements for sampling according to a distribution. We can associate to games a model of computation, which specifies how much running each (type of) statement costs in terms of time, memory, or both. A typical model is to assign to each statement the same abstract unit cost, and the run time then becomes the number of statements executed in the course of the game. When procedures are called, we attribute the unit cost of the call statement to the caller and the remaining cost of executing the procedure's statements to the callee.

By default we will require that games terminate in some finite number of steps $t$, and clarify explicitly when this does not hold. The number of queries made by the main procedure, or any other procedure for that matter including unspecified ones, is therefore also upper bound by $t$. We may often limit the number of queries made by an adversary to some maximum number $q \leq t$.

Given these finiteness conditions, we similarly know that there is a finite limit on the number of random samples made in a game. Since we restricted to random sampling from finite sets, we have that for any game G there is an event space $\Omega_{\mathrm{G}}$ and an associated probability distribution defining the output of the game G. Given our restrictions, $\Omega_{\mathrm{G}}$ is a set of possible values, the cross-product of all the random sampling procedures within G. We sometimes refer to $\Omega_{\mathrm{G}}$ as the *coins* of the game. For some fixed unspecified procedures $P_1, \ldots, P_k$ we denote the event that executing the game $\mathrm{G}^{P_1,\ldots,P_k}$ outputs a particular value $y$ by "$\mathrm{G}^{P_1,\ldots,P_k} \Rightarrow y$" and the associated probability over $\Omega_{\mathrm{G}}$ is denoted $\Pr[\mathrm{G}^{P_1,\ldots,P_k} \Rightarrow y]$. When $y$ is clear from context we will omit it, writing instead $\Pr[\mathrm{G}^{P_1,\ldots,P_k}]$. For example, we will often have games output a boolean and then $y$ will most often be the value $\mathsf{true}$.

We can similarly associate to any variable within a game an event within $\Omega_{\mathrm{G}}$. These can also be

equivalently considered to be random variables on domain $\Omega_G$. Our convention will be to overload notation, and define the event that a variable $X$ in a game G takes on a certain value $y$ as simply "$X = y$" with associated probability $\Pr[X = y]$ over the coins of $\Omega_{\mathrm{G}}$.

**Runnable games.** We want to emphasize a point, which is that games are by our conventions above runnable. That means that, once any unspecified procedures are fixed, you could write a program in a conventional programming language, and actually run the game on a real computer. Obviously like with all abstract algorithms, the actual run times will vary, but assuming relative efficiency the game will complete.

It relatively frequently arises in proofs that one deviates from runnable games. A common example is logic of the following form. Let $\mathcal{A}$ be an adversary, and consider a game $\mathrm{G}^{\mathcal{A}}$. Remember we implicitly assume that $\mathcal{A}$ is compatible with G, and we have that $\mathrm{G}^{\mathcal{A}}$ is runnable. Now consider the set of all adversaries compatible with $\mathcal{A}$, and let $\mathcal{A}_{\max}$ be the member of this set that maximizes $\Pr[\mathrm{G}^{\mathcal{A}} \Rightarrow \mathsf{true}]$. But now $\mathrm{G}^{\mathcal{A}_{\max}}$ is no longer runnable, because $\mathcal{A}_{\max}$ is not concretely specified. While mathematically $\mathcal{A}_{\max}$ is well-defined, there is no clear way to write down its code, even given the code defining $\mathcal{A}$. We will try whenever possible to avoid such arguments, as they have various subtle implications that are, in general, not great for making clear claims about security.

# 3 Ciphers and Initial Security Notions

**Ciphers.** We start by defining a cipher. A cipher $\mathcal{E} = (E, D)$ is defined by a a pair of deterministic algorithms $E$ and $D$. To any cipher $\mathcal{E}$ we associate sets called the key space $\mathcal{K}$, message space $\mathcal{M}$, and ciphertext space $\mathcal{C}$. We do not surface in the notation for a cipher these sets, and will require that the association be clear from context.

The algorithms are two-input. Enciphering takes a key $K \in \mathcal{K}$ and message $M \in \mathcal{M}$, and outputs a ciphertext $C \in \mathcal{C}$. Because $E$ is deterministic, we can equally formalize it as a map $E\colon \mathcal{K} \times \mathcal{M} \to \mathcal{C}$. For a given key $K$ we let $E_K\colon \mathcal{M} \to \mathcal{C}$ be defined by $E_K(M) = E(K, M)$ for all $M \in \mathcal{M}$. Deciphering takes a key $K \in \mathcal{K}$ and ciphertext $C \in \mathcal{C}$ and outputs a message $M \in \mathcal{M}$. Again, we can view it as a map $D\colon \mathcal{K} \times \mathcal{C} \to \mathcal{M}$.

Both $E$ and $D$ must be efficiently computable for all $K \in \mathcal{K}$. (We have not defined efficiently computable, and use the term here informally.) We require that a cipher be correct, meaning that for $D_K(E_K(M)) = M$ for all $M$.

**Security notions.** What do we intuitively expect of a cipher? Minimally:

- Secret key should remain secret
- Message should remain secret

Let's try to formalize these notions.

- TKR security
- KR security
- (ot-)IND security

We let $\mathrm{TKR}_{\mathcal{E}}$-advantage of a $\mathrm{TKR}_{\mathcal{E}}$-adversary $\mathcal{A}$ be defined by

$$\mathbf{Adv}^{\mathrm{tkr}}_{\mathcal{E}}(\mathcal{A}) = \Pr\left[\, \mathrm{TKR}^{\mathcal{A}}_{E} \Rightarrow \mathsf{true} \,\right] .$$

We let $\mathrm{KR}_{\mathcal{E}}$-advantage of a $\mathrm{KR}_{\mathcal{E}}$-adversary $\mathcal{A}$ be defined by

$$\mathbf{Adv}^{\mathrm{kr}}_{\mathcal{E}}(\mathcal{A}) = \Pr\left[\, \mathrm{KR}^{\mathcal{A}}_{E} \Rightarrow \mathsf{true} \,\right] .$$

We let $\mathrm{otIND}_{\mathcal{E}}$-advantage of a $\mathrm{otIND}_{\mathcal{E}}$-adversary $\mathcal{A}$ be defined by

$$\mathbf{Adv}^{\text{ot-ind}}_{\mathcal{E}}(\mathcal{A}) = 2 \cdot \Pr\left[\, \mathrm{otIND}^{\mathcal{A}}_{E} \Rightarrow \mathsf{true} \,\right] - 1 .$$

- First example: Our simple OTP cipher is not TKR secure? Go over example: $\mathcal{A}$ queries once on arbitrary message, recovers $K$ by computing $M \oplus C$. This is guaranteed to succeed because $M \oplus C$ uniquely defines $K$. What does this mean? Isn't OTP considered secure? Shannon said so!
- Second example: Give toy cipher $\mathcal{E}$ for which $\mathrm{TKR}_{\mathcal{E}}$ has $\mathbf{Adv}^{\mathrm{tkr}}_{\mathcal{E}}(\mathcal{A}) = 0$ for any adversary $\mathcal{A}$. What is it? $E(K, M) = M$. It is correct
- Third example: Exhaustive key search attack against generic cipher. Emphasize that lower-bounding the efficacy of this is not possible in general. Why? Consider toy identity cipher!
- Discuss the KR definition. Rules out the identity map as being relevant. Lower-bounding security is

$\underline{\mathrm{TKR}^{\mathcal{A}}_{\mathcal{E}}}$
$K \leftarrow\!\!\$\ \mathcal{K}$
$K^* \leftarrow\!\!\$\ \mathcal{A}^{\mathrm{Fn}}$
Ret $(K = K^*)$

$\underline{\mathrm{Fn}(M)}$
$C \leftarrow E_K(M)$
Ret $C$

$\underline{\mathrm{KR}^{\mathcal{A}}_{\mathcal{E}}}$
$K \leftarrow\!\!\$\ \mathcal{K}$
$K^* \leftarrow\!\!\$\ \mathcal{A}^{\mathrm{Fn}}$
win $\leftarrow$ true
For $M \in \mathcal{X}$:
    If $E_{K^*}(M) \neq E_K(M)$ then
        win $\leftarrow$ false
Ret win

$\underline{\mathrm{Fn}(M)}$
$\mathcal{X} \leftarrow \mathcal{X} \cup \{M\}$
$C \leftarrow E_K(M)$
Ret $C$

$\underline{\mathrm{otIND}^{\mathcal{A}}_{\mathcal{E}}}$
$K \leftarrow\!\!\$\ \mathcal{K}$
$b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathrm{Fn}}$
Ret $(b = b')$

$\underline{\mathrm{Fn}(M_0, M_1)}$
$C \leftarrow E_K(M_b)$
Ret $C$

- Shannon's perfect secrecy (one-time left-or-right indistinguishability).

**Theorem 1** *Let $\mathcal{E}$ be a cipher. For any* $\mathrm{TKR}_{\mathcal{E}}$*-adversary $\mathcal{A}$, we give a* $\mathrm{KR}_{\mathcal{E}}$*-adversary $\mathcal{B}$ such that* $\mathbf{Adv}^{\mathrm{kr}}_{\mathcal{E}}(\mathcal{A}) = \mathbf{Adv}^{\mathrm{tkr}}_{\mathcal{E}}(\mathcal{B})$.

**Theorem 2** *Let $\mathcal{E}$ be the OTP cipher. Then for any single-query* $\mathrm{otIND}_{\mathcal{E}}$*-adversary $\mathcal{A}$ it holds that* $\mathbf{Adv}^{\mathrm{ot\text{-}ind}}_{\mathcal{E}}(\mathcal{A}) = 0$.

**Theorem 3** *Let $\mathcal{E}$ be a cipher defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ such that for any* $\mathrm{otIND}_{E}$*-adversary $\mathcal{A}$ it holds that* $\mathbf{Adv}^{\mathrm{ot\text{-}ind}}_{\mathcal{E}}(\mathcal{A}) = 0$. *Then* $|\mathcal{K}| \geq |\mathcal{M}|$.

## References


[1] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In *CRYPTO*, 1994.

[2] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT*, 2006.

[3] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[4] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In *EUROCRYPT*, 2011.

[5] Goldwasser Shafi and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[6] Claude E Shannon. Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715, 1949.