6.1 Historical Perspective: Nash’s Letter to the NSA

John Nash is a mathematician who earned the 1994 Nobel Prize in Economics for his work in game theory. His life was made into a popular Hollywood movie, *A Beautiful Mind*.

In 1955, Nash was in correspondence with the United States National Security Agency (NSA), discussing new methods of encryption that he had devised. In these letters, he also proposes some general principles of cryptography. The bold highlighting was added by me:

To consider the resistance of an enciphering process to being broken we should assume that at some times the enemy knows everything but the key being used and to break it need only discover the key from this information.

We see immediately that in principle the enemy needs very little information to begin to break down the process. Essentially, as soon as \( r \) bits\(^2\) of enciphered message have been transmitted the key is about determined. This is no security, for a practical key should not be too long. But this does not consider how easy or difficult it is for the enemy to make the computation determining the key. If this computation, although possible in principle, were sufficiently long at best then the process could still be secure in a practical sense.

The most direct computation procedure would be for the enemy to try all \( 2^r \) possible keys, one by one. Obviously this is easily made impractical for the enemy by simply choosing \( r \) large enough.

In many cruder types of enciphering, particularly those which are not auto-coding, such as substitution ciphers [letter for letter, letter pair for letter pair, triple for triple...] shorter means for computing the key are feasible, essentially because the key can be determined piece meal, one substitution at a time.

So a logical way to classify enciphering processes is by the way in which the computation length for the computation of the key increases with increasing length of the key. This is at best exponential and at worst probably a relatively small power of \( r \), \( ar^2 \) or \( ar^3 \), as in substitution ciphers.

---


2Nash is using \( r \) to denote the length of the key, in bits.
Now my general conjecture is as follows: For almost all sufficiently complex types of enciphering, especially where the instructions given by different portions of the key interact complexly with each other in the determination of their ultimate effects on the enciphering, the mean key computation length increases exponentially with the length of the key, or in other words, with the information content of the key.

The significance of this general conjecture, assuming its truth, is easy to see. It means that it is quite feasible to design ciphers that are effectively unbreakable. As ciphers become more sophisticated the game of cipher breaking by skilled teams, etc. should become a thing of the past.

These letters from Nash remained classified until 2012, so they did not influence the development of modern cryptography. Still, his letters illustrate some important concepts that are now fundamental in the field:

- **The idea that security can be based on the computational difficulty of an attack rather than on the impossibility of an attack.** In other words, breaking a cryptographic scheme may be possible in principle, but require too much computation time to be feasible.

  In the previous parts of this course, we have already discussed one-time private communication and secret-sharing. Both of these tools have security guarantees which essentially say that attacks are impossible in principle. Only a handful of other cryptographic tools have such strong guarantees.

  By contrast, nearly all of the well-known products of cryptography (public-key encryption, hash functions, etc.) have weaker security guarantees, which say only that attacks are merely computationally infeasible, not impossible.

- **The importance of distinguishing between polynomial and exponential time.** Throughout computer science, polynomial-time is used as a formal definition of “efficient,” and exponential-time (and above) is synonymous with “intractible.” In modern cryptography we (more or less) define security following Nash’s suggestion here, demanding that breaking a scheme requires exponential time.

  Nash’s comments were surprisingly ahead of their time. Polynomial-time wasn’t formally proposed as a natural definition for “efficiency” in computation until Alan Cobham, 10 years after Nash’s letter was written.³ Before that, a famous 1956 letter from Kurt Gödel to John von Neumann hints at the importance of polynomial time.⁴ But that letter is not as explicit as Nash’s, besides being written a year later.

### 6.2 What Makes Cryptography “Modern?”

When we refer to “modern” cryptography (in contrast to “classical” cryptography) we are talking about cryptography that incorporates the following aspects:

- Emphasis on formal security definitions and formal proofs of security.

- Basing security on the hardness, rather than impossibility, of computational tasks.


• Considering computationally-bounded adversaries.

Cryptographic security definitions in the “modern crypto paradigm” generally all follow the following template:

An adversary that runs in a reasonable amount of time won’t break the security of the scheme, except with very small probability.

Often the most interesting part of these definitions is deciding what it means to “break the security” of a scheme. This, of course, depends on the task we are trying to accomplish with cryptography. In the upcoming lectures, we’ll soon see some examples.

Before that, we can get more specific about the other parts of this template, since they are common to all the security definitions we’ll see in the course.

6.3 Adversaries are Arbitrary Algorithms, Actually

In our security definitions we model adversaries as arbitrary algorithms/computations. We can specify the input/output “interface” of the adversary to model the capabilities of an adversary. In some security definitions, we also provide additional interfaces to the adversary, which allow her to take various kinds of actions.

For instance, in a security definition for encryption, an adversary might be a program that receives a ciphertext as input. In a different security definition, we might also provide the adversary program a special interface where it can generate ciphertexts on its own and ask them to be decrypted.

6.3.1 Kerckhoffs’ Principle (What Does the Adversary “Know?”)

In 1883, Auguste Kerckhoffs laid out several rules of thumb for cryptography. One of those rules is now known as Kerckhoffs’ principle, and it basically says “assume that the adversary has full knowledge of the system.” It is very difficult (and often unrealistic) to model a situation in which the adversary does not have full knowledge of cryptographic algorithms she is attacking.

Kerckhoffs’ principle shows up in our security definitions in the following way. Suppose we come up with a cryptographic scheme that we want to show is secure. The security definition requires that we consider all adversaries, and that includes adversaries who “know” (or, whose code is allowed to depend on) arbitrary facts about our cryptographic scheme.

There is a subtlety here that deserves some careful attention, though. Cryptographic algorithms include steps where the users generate randomness. Kerckhoffs’ principle says that adversaries will know how the users choose randomness (i.e., from which distribution); it doesn’t say that adversaries will know the result of the users’ choice of randomness. It’s the difference between knowing that you will choose a random card from a deck (i.e., the uniform distribution over a set of 52 items) versus knowing exactly what card you chose. This subtlety shows up in our definitions in that we fix a particular adversary, and then we consider a “security game” in which users pick fresh randomness, etc. The adversary cannot depend on the random choices made by the users, since the choice of randomness “happens after” the adversary is fixed.

Using Kerckhoffs’ principle gives us strong security guarantees. If we can show a system to be secure in this standard way, then we can be sure that all of the security comes from the operation of the cryptographic algorithms, and not from any secrecy of the algorithms themselves.
6.4 Bounds on Computation

I don’t lay awake at night worried about hackers who are willing to build supercomputers that might be able to decrypt my TLS traffic in 100 years. Likewise, our security template only concerns itself with adversaries who spend a “reasonable amount” of computational resources. Put differently, we don’t bother putting in the extra effort to secure ourselves against unrealistically super-powerful adversaries (more precisely, it is often not even possible to be secure against them).

Yet another way to look at things is that we are placing limits on what adversaries are “allowed to do.” In a typical security definition, adversaries are “not allowed” to spend too much computation time. Isn’t that a little odd? We are setting up the rules of the game and fixing them in our favor!

Indeed, we have to carefully strike a balance between two factors. On the one hand, if we let adversaries be unlimited in their power, then most cryptographic tasks become impossible. On the other hand, we want security definitions that consider all realistic adversaries, and that don’t artificially limit them in ways that trivialize the problem.

In computer science, “polynomial-time” is the standard way to formalize the informal idea of a “reasonable amount of computation time.”

Definition: A program runs in polynomial time if there exists a constant $c > 0$ such that for all sufficiently long input strings $x$, the program stops after no more than $O(|x|^c)$ steps.

Polynomial time is not a perfect match to what we mean by “efficient.” It includes algorithms with running time $\Theta(n^{1000})$, while excluding those with running time $\Theta(n^{\log^* n})$. Still, it enjoys an important closure property: repeating a polynomial-time process a polynomial number of times results in a polynomial-time process overall.

6.4.1 Security Parameter

The definition of polynomial-time is asymptotic. That is, it considers the behavior of a computer program as the size of the inputs grows to infinity, which is standard fare in the study of algorithms.

But cryptographic algorithms often take multiple different inputs to serve various purposes. To be absolutely clear about our “measuring stick” for polynomial time, we measure the efficiency of cryptographic algorithms (and adversaries!) against something called the security parameter, which is the number of bits needed to represent secret keys and/or randomly chosen elements used in the scheme.

It’s helpful to think of the security parameter as a tuning knob for the cryptographic system. When we dial up this knob, we increase the size of the keys in the system, the size of all associated things like ciphertexts, and the required computation time of the associated algorithms. Importantly, the amount of effort required by the honest parties grows reasonably (as a fixed polynomial function of the security parameter) while the effort required to violate security increases faster than any (polynomial-time) adversary can keep up.

6.4.2 Potential Pitfall: Numerical Algorithms

Many public-key cryptographic algorithms are based on problems from abstract algebra and number theory. As such, we’re often asking users to perform operations on very large numbers. We must remember that in a computer, a number is represented as a sequence of bits! To represent a number

---

5I wrote that sentence before the NSA was revealed to be poisoning crypto standards and mounting man-in-the-middle attacks on TLS traffic with forged certificates. Now that’s something to be concerned about.
in the range from \{0, \ldots, N - 1\} requires $\lceil \log_2 N \rceil$ bits. This means that $\lceil \log_2 N \rceil$, rather than $N$, is our security parameter. We should therefore check whether our algorithms have running times which are polynomial in $\lceil \log_2 N \rceil$, rather than polynomial in $N$.

For reference, here are some numerical operations that we will be using later in the class, and their known efficiencies:

<table>
<thead>
<tr>
<th>Efficient algorithms exist</th>
<th>No efficient algorithm (as far as we know)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computing GCD (Euclid’s algorithm)</td>
<td>Factoring integers</td>
</tr>
<tr>
<td>Computing modular inverses (Bezout)</td>
<td>Determining $\phi(n)$ given $n$</td>
</tr>
<tr>
<td>Modular arithmetic ($\times, \div, +, -$)</td>
<td>Discrete logarithm</td>
</tr>
<tr>
<td>Modular exponentiation (repeated squaring)</td>
<td>Square roots modulo a composite</td>
</tr>
</tbody>
</table>

### 6.5 Very Small Probabilities

In most of our applications, an adversary can always violate security simply by guessing some value that was chosen at random, like the secret key. However, imagine a scheme that has 1024-bit secret keys chosen uniformly at random. The probability of correctly guessing the key is $2^{-1024}$, which is the probability of being dealt a royal flush in each of 53 consecutive hands of 5-card stud poker.

So it makes sense that we wouldn’t consider “attacks” with such ridiculously small success probability as problematic to security. More often, we’d be quite happy to settle for a security guarantee like the following:

> “Even a sophisticated adversary can’t do much better than a dumb adversary that just attacks by blindly guessing the key.”

Recall the concept of a security parameter from above. If the security parameter of a scheme is $n$, then a blind guess is correct with probability $1/2^n$. Now what about an adversary who makes 2 guesses, or $n$ guesses, or $n^{12}$ guesses? Such an adversary might then succeed in her attack with probability $2/2^n$, $n/2^n$, or $n^{12}/2^n$. In what sense are these probabilities “not much better than” the $1/2^n$ probability of blind guessing?

The common feature of all of these probabilities is in how fast they approach 0, as functions of $n$. This property forms the basis for a cryptographically useful definition of “small enough” probabilities:

**Definition:** A function $f : \mathbb{N} \to \mathbb{R}$ is negligible if, for all constants $c > 0$, we have $f(n) < 1/n^c$ for all but finitely many $n$. That is, $f$ asymptotically approaches zero faster than any inverse polynomial.

To see why this is the “right” definition for cryptography, let’s consider a definition that is equivalent:

**Definition:** A function $f$ is negligible if, for every polynomial $p$, we have $\lim_{n \to \infty} p(n)f(n) = 0$.

In fact, negligible functions satisfy a kind of closure property:

If $p(n)$ is a polynomial, and $f(n)$ is negligible, then $p(n)f(n)$ is also negligible.

This property means we can use the following kind of reasoning: suppose there is an efficient adversary $A$ which breaks our scheme with negligible probability $f(n)$. Repeating this attack $A$ a polynomial $p(n)$ number of times results in another efficient adversary (because the product of
two polynomials is polynomial). Depending on the kind of attack we are talking about, this new adversary’s success probability could be \( p(n) f(n) \), which is negligible thanks to our convenient closure property.

### 6.5.1 Shortcuts

In general, we will try to avoid mucking around too much with negligible functions (see the notation below, which nicely glosses over these things). You will usually be safe to simply associate negligible probabilities with some concrete function like \( \frac{1}{2^n} \). In the end, what’s important is that you cannot “amplify” a negligible probability into a non-negligible probability by repeating the attack a polynomial number of times.

### 6.5.2 Notation

Because of the convenient properties of negligible and polynomial functions, we can use convenient notation like this:

\[
\Pr[X] \approx \Pr[Y] \iff \left| \Pr[X] - \Pr[Y] \right| \text{ is a negligible function (in the security parameter)}
\]

The \( \approx \) symbol is then transitive: if \( \Pr[X] \approx \Pr[Y] \) and \( \Pr[Y] \approx \Pr[Z] \), then \( \Pr[X] \approx \Pr[Z] \) (perhaps with a slightly larger, but still negligible, difference).

Then we can interpret statements like the following:

\[
\begin{align*}
\Pr[X] &\approx 0 \iff \text{“event } X \text{ almost never happens”} \\
\Pr[Y] &\approx 1 \iff \text{“event } Y \text{ almost always happens”} \\
\Pr[A] &\approx \Pr[B] \iff \text{“events } A \text{ and } B \text{ happen with essentially the same probability”}
\end{align*}
\]

In this last example \( \Pr[A] \approx \Pr[B] \), this doesn’t mean that events \( A \) and \( B \) almost always happen together — it might be that \( A \) and \( B \) are mutually exclusive events. To say that “\( A \) and \( B \) almost always happen together,” you’d have to say something like \( \Pr[A \oplus B] \approx 0 \), where \( A \oplus B \) denotes the event that \textit{exactly one} of \( A \) and \( B \) happens.

### Summary:

- The security parameter is the \textbf{number of bits} needed to write down a secret key.
- The honest parties must be able to carry out their prescribed operations efficiently — \textbf{polynomial time} (measured as a function of the security parameter).
- The adversary is allowed to be an arbitrary computer program that \textbf{runs in polynomial time} (measured as a function of the security parameter).
- The probability that an adversary succeeds in breaking the security condition must be \textbf{a negligible function of the security parameter}.

\[\text{It's only transitive when applied a polynomial number of times. So you can't define a whole slug of events } X_i, \text{ show that } \Pr[X_i] \approx \Pr[X_{i+1}], \text{ and conclude that } \Pr[X_i] \approx \Pr[X_t], \text{ if } t \text{ is, say, exponential in the security parameter. It's rare that we'll encounter this subtlety in this course.}\]
6.6 Reductions, and Interpreting “Security Proofs”

Recall that our security conditions refer to whether it’s possible for a polynomial-time algorithm to accomplish a particular task. Maybe this reminds you of the famous P = NP problem. Indeed, if P = NP, then every problem that can possibly appear in a cryptographic definition can be solved in polynomial time, and all of what I’ve called “modern” cryptography is impossible. Put another way, the kind of cryptography we’ll discuss from here on depends on the unproven assumption that P ≠ NP. A full discussion of why P ≠ NP is plausible is best left for a course on computational complexity.

Given that our basis for modern cryptography rests on unproven assumptions, can we really hope to prove that our systems are secure? In some sense, no, because an outright proof would imply that P ≠ NP, which seems well beyond our reach. On the other hand, we can get as much confidence as possible (given that P vs NP is unresolved) in the security of our systems by using the idea of a reduction, a concept that pervades all of computer science.

The “provable” part of modern cryptography typically follows the following structure:

1. First, a well-known, well-studied problem is identified, which has no known polynomial-time solution. For instance, the problem of factoring large numbers has been studied for thousands of years, and no known polynomial-time (classical) algorithm is yet known.

2. We design a system so that:

   If you give me an efficient adversary that breaks the system, then I can use that adversary to come up with an efficient algorithm that solves this very well-known problem (like factoring).

   A proof of this form — “if some kind of algorithm exists for problem A, then a certain kind of algorithm exists for problem B” — is called a reduction.

   Admittedly, reduction proofs are a little bit strange at first. We are proving an “if A then B” statement, where we’re pretty sure that A is in fact impossible. Just think of this process as saying: Look, we think condition A is implausible. In fact, it’s so implausible, that if it were true, then this other even more implausible thing would happen!

3. Finally, we typically state the security of the system in the contrapositive:

   If factoring is intractable (i.e., there is no efficient algorithm for factoring large numbers), then my scheme is secure (i.e., no efficient adversary breaks the security property).

6.7 Problems

1. In Section 6.5, we said:

   In most of our applications, an adversary can always violate security simply by guessing some value that was chosen at random, like the secret key.

---

7Other properties are also useful too, but these are the important ones when considering how to interpret “provable security.”

8"If pigs could fly, then ..."
Is this statement true for schemes with perfect (information-theoretic) security like one-time pad, and secret sharing?