4 Basing Cryptography on Intractable Computations

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

In 1955, Nash was in correspondence with the United States National Security Agency (NSA),[1] discussing new methods of encryption that he had devised. In these letters, he also proposes some general principles of cryptography (bold highlighting not in the original):

> *. . . in principle the enemy needs very little information to begin to break down the process. Essentially, as soon as λ 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.***

Nash is saying something quite profound: **it doesn't really matter whether attacks are *impossible*, only whether attacks are *computationally infeasible*.** If his letters hadn't been kept classified until 2012, they might have accelerated the development of "modern" cryptography, in which security is based on intractable computations. As it stands, he was decades ahead of his time in identifying one of the most important concepts in modern cryptography.

4.1 What Qualifies as a "Computationally Infeasible" Attack?

Schemes like one-time pad cannot be broken, even by an attacker that performs a **brute-force** attack, trying all possible keys (see Exercise 1.5). However, all future schemes that we will see can indeed be broken by such an attack. Nash is quick to point out that, for a scheme with λ-bit keys:

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

[1] The original letters, handwritten by Nash, are available at: https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/nash-letters/nash_letters1.pdf.

[2] Nash originally used r to denote the length of the key, in bits. In all of the excerpts quoted in this chapter, I have translated his mathematical expressions into our notation (λ).

We call λ the **security parameter** of the scheme. It is like a knob that allows the user to tune the security to any desired level. Increasing λ makes the difficulty of a brute-force attack grow exponentially fast. Ideally, when using λ-bit keys, *every* attack (not just a brute-force attack) will have difficulty roughy 2^λ. However, sometimes faster attacks are inevitable. Later in this chapter, we will see why many schemes with λ-bit keys have attacks that cost only $2^{\lambda/2}$. It is common to see a scheme described as having n**-bit security** if the best known attack requires 2^n steps.

Just how impractical is a brute-force computation on a 64-bit key? A 128-bit key? Huge numbers like 2^{64} and 2^{128} are hard to grasp at an intuitive level.

Example *It can be helpful to think of the cost of a computation in terms of monetary value, and a convenient way to assign such monetary costs is to use the pricing model of a cloud computing provider. Below, I have calculated roughly how much a computation involving 2^λ CPU cycles would cost on Amazon EC2, for various choices of λ.*[3]

clock cycles	*approx cost*	*reference*
2^{50}	*$3.50*	*cup of coffee*
2^{55}	*$100*	*decent tickets to a Portland Trailblazers game*
2^{65}	*$130,000*	*median home price in Oshkosh, WI*
2^{75}	*$130 million*	*budget of one of the Harry Potter movies*
2^{85}	*$140 billion*	*GDP of Hungary*
2^{92}	*$20 trillion*	*GDP of the United States*
2^{99}	*$2 quadrillion*	*all of human economic activity since 300,000 BC*[4]
2^{128}	*really a lot*	*a billion human civilizations' worth of effort*

Remember, this table only shows the cost to perform 2^λ clock cycles. *A brute-force attack checking 2^λ keys would take many more cycles than that! But, as a disclaimer, these numbers reflect only the* retail *cost of performing a computation, on fairly standard* general-purpose hardware. *A government organization would be capable of manufacturing special-purpose hardware that would significantly reduce the computation's cost. The exercises explore some of these issues, as well as non-financial ways of conceptualizing the cost of huge computations.*

Example *In 2017, the first collision in the SHA-1 hash function was found (we will discuss hash functions later in the course). The attack involved evaluating the SHA-1 function 2^{63} times on a cluster of GPUs. An article in* Ars Technica[5] *estimates the monetary cost of the attack as follows:*

> *Had the researchers performed their attack on Amazon's Web Services platform, it would have cost $560,000 at normal pricing. Had the researchers been patient and waited to run their attack during off-peak hours, the same collision would have cost $110,000.*

[3] As of October 2018, the cheapest class of CPU that is suitable for an intensive computation is the m5.large, which is a 2.5 GHz CPU. Such a CPU performs 2^{43} clock cycles per hour. The cheapest rate on EC2 for this CPU is 0.044 USD per hour (3-year reserved instances, all costs paid upfront). All in all, the cost for a single clock cycle (rounding down) is 2^{-48} USD.

[4] I found some estimates (https://en.wikipedia.org/wiki/Gross_world_product) of the gross world product (like the GDP but for the entire world) throughout human history, and summed them up for every year.

[5] https://arstechnica.com/information-technology/2017/02/at-deaths-door-for-years-widely-used-sha1-function-is-now-de

Asymptotic Running Time

It is instructive to think about the monetary cost of an enormous computation, but it doesn't necessarily help us draw the line between "feasible" attacks (which we want to protect against) and "infeasible" ones (which we agreed we don't need to care about). We need to be able to draw such a line in order to make security definitions that say "only feasible attacks are ruled out."

Once again, John Nash thought about this question. He suggested to consider the **asymptotic** cost of an attack — how does the cost of a computation scale as the security parameter λ goes to infinity?

> *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 λ, $a \cdot \lambda^2$ or $a \cdot \lambda^3$, as in substitution ciphers.*

Nash highlights the importance of attacks that run in polynomial time:

Definition 4.1 *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 algorithms scale reasonably well (especially when the exponent is small), but exponential-time algorithms don't. It is probably no surprise to modern readers to see "polynomial-time" as a synonym for "efficient." However, it's worth pointing out that, again, Nash is years ahead of his time relative to the field of computer science.

In the context of cryptography, our goal will be to ensure that no polynomial-time attack can successfully break security. We will not worry about attacks like brute-force that require exponential time.

Polynomial time is not a perfect match to what we mean when we informally talk about "efficient" algorithms. Algorithms with running time $\Theta(n^{1000})$ are technically polynomial-time, while those with running time $\Theta(n^{\log\log\log n})$ aren't. Despite that, polynomial-time is extremely useful because of the following **closure property**: repeating a polynomial-time process a polynomial number of times results in a polynomial-time process overall.

Potential Pitfall: Numerical Algorithms

When we study public-key cryptography, we will discuss algorithms that operate on very large numbers (e.g., thousands of bits long). You must remember that representing the number N on a computer requires only $\sim \log_2 N$ bits. This means that $\log_2 N$, rather than N, is our security parameter! We will therefore be interested in whether certain operations on the number N run in polynomial-time as a function of $\log_2 N$, rather than in N. Keep in mind that the difference between running time $O(\log N)$ and $O(N)$ is the difference between writing down a number and counting to the number.

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

Efficient algorithm known:	**No known efficient algorithm:**
Computing GCDs	Factoring integers
Arithmetic mod N	Computing $\phi(N)$ given N
Inverses mod N	Discrete logarithm
Exponentiation mod N	Square roots mod composite N

Again, "efficient" means polynomial-time. Furthermore, we only consider polynomial-time algorithms that run on standard, *classical* computers. In fact, all of the problems in the right-hand column *do* have known polynomial-time algorithms on *quantum* computers.

4.2 What Qualifies as a "Negligible" Success Probability?

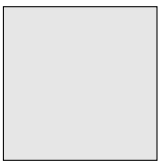
It is not enough to consider only the running time of an attack. For example, consider an attacker who just tries to guess a victim's secret key, making a single guess. This attack is extremely cheap, but it still has a nonzero chance of breaking security!

In addition to an attack's running time, we also need to consider its success probability. We don't want to worry about attacks that are as expensive as a brute-force attack, and we don't want to worry about attacks whose success probability is as low as a blind-guess attack.

An attack with success probability 2^{-128} should not really count as an attack, but an attack with success probability $1/2$ should. Somewhere in between 2^{-128} and 2^{-1} we need to find a reasonable place to draw a line.

Example *Now we are dealing with extremely tiny probabilities that can be hard to visualize. Again, it can be helpful to conceptualize these probabilities with a more familiar reference:*

probability	*equivalent*
2^{-10}	*full house in 5-card poker*
2^{-20}	*royal flush in 5-card poker*
2^{-28}	*you win this week's Powerball jackpot*
2^{-40}	*royal flush in 2 consecutive poker games*
2^{-60}	*the next meteorite that hits Earth lands in this square* →



As before, it is not clear exactly where to draw the line between "reasonable" and "unreasonable" success probability for an attack. Just like we did with polynomial running time, we can also use an **asymptotic** approach to define when a probability is negligibly small. Just as "polynomial time" considers how fast an algorithm's running time approaches infinity as its input grows, we can also consider how fast a success probability approaches zero as the security parameter grows.

In a scheme with λ-bit keys, a blind-guessing attack succeeds with probability $1/2^\lambda$. Now what about an attacker who makes 2 blind guesses, or λ guesses, or λ^{42} guesses? Such an attacker would still run in polynomial time, and has success probability $2/2^\lambda$, $\lambda/2^\lambda$, or $\lambda^{42}/2^\lambda$. However, no matter what polynomial you put in the numerator, the probability still goes to zero. Indeed, $1/2^\lambda$ **approaches zero so fast that no polynomial can "rescue" it**; or, in other words, it approaches zero faster than 1 over any polynomial. This idea leads to our formal definition:

Definition 4.2 (Negligible) *A function f is **negligible** if, for every polynomial p, we have $\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = 0$.*

In other words, a negligible function approaches zero so fast that you can never catch up when multiplying by a polynomial. This is exactly the property we want from a security guarantee that is supposed to hold against all polynomial-time adversaries. If a polynomial-time attacker succeeds with probability f, then repeating the same attack p independent times would still be an overall polynomial-time attack (if p is a polynomial), and its success probability would be $p \cdot f$.

When you want to check whether a function is negligible, you only have to consider polynomials p of the form $p(\lambda) = \lambda^c$ for some constant c:

Claim 4.3 *If for every integer c, $\lim_{\lambda\to\infty} \lambda^c f(\lambda) = 0$, then f is negligible.*

Proof Suppose f has this property, and take an arbitrary polynomial p. We want to show that $\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = 0$.

If d is the degree of p, then $\lim_{\lambda\to\infty} \frac{p(\lambda)}{\lambda^{d+1}} = 0$. Therefore,

$$\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = \lim_{\lambda\to\infty}\left[\frac{p(\lambda)}{\lambda^{d+1}}\left(\lambda^{d+1}\cdot f(\lambda)\right)\right] = \left(\lim_{\lambda\to\infty}\frac{p(\lambda)}{\lambda^{d+1}}\right)\left(\lim_{\lambda\to\infty}\lambda^{d+1}\cdot f(\lambda)\right) = 0\cdot 0.$$

The second equality is a valid law for limits since the two limits on the right exist and are not an indeterminate expression like $0 \cdot \infty$. The final equality follows from the hypothesis on f. ■

Example *The function $f(\lambda) = 1/2^\lambda$ is negligible, since for any integer c, we have:*

$$\lim_{\lambda\to\infty} \lambda^c/2^\lambda = \lim_{\lambda\to\infty} 2^{c\log(\lambda)}/2^\lambda = \lim_{\lambda\to\infty} 2^{c\log(\lambda)-\lambda} = 0,$$

since $c\log(\lambda) - \lambda$ approaches $-\infty$ in the limit, for any constant c. Using similar reasoning, one can show that the following functions are also negligible:

$$\frac{1}{2^{\lambda/2}}, \qquad \frac{1}{2^{\sqrt{\lambda}}}, \qquad \frac{1}{2^{\log^2\lambda}}, \qquad \frac{1}{\lambda^{\log\lambda}}.$$

Functions like $1/\lambda^5$ approach zero but not fast enough to be negligible. To see why, we can take polynomial $p(\lambda) = \lambda^6$ and see that the resulting limit does not satisfy the requirement from Definition 4.2:

$$\lim_{\lambda\to\infty} p(\lambda)\frac{1}{\lambda^5} = \lim_{\lambda\to\infty}\lambda = \infty \neq 0$$

In this class, when we see a negligible function, it will typically always be one that is easy to recognize as negligible (just as in an undergraduate algorithms course, you won't really encounter algorithms where it's hard to tell whether the running time is polynomial).

Definition 4.4 ($f \approx g$) *If $f, g : \mathbb{N} \to \mathbb{R}$ are two functions, we write $f \approx g$ to mean that $\left|f(\lambda) - g(\lambda)\right|$ is a negligible function.*

(c) Money is not the only way to measure the energy cost of a huge computation. Search online to find out how much carbon dioxide (CO_2) is placed into the atmosphere per unit of electrical energy produced, under a typical distribution of power production methods. Estimate how many tons of CO_2 are produced as a side-effect of computing $2^{40}, 2^{50}, \ldots, 2^{120}$ SHA-256 hashes.

★ (d) Estimate the corresponding CO_2 concentration (parts per million) in the atmosphere as a result of computing $2^{40}, 2^{50}, \ldots, 2^{120}$ SHA-256 hashes. If it is possible without a PhD in climate science, try to estimate the increase in average global temperature caused by these computations.

4.2. Which of the following are negligible functions in λ? Justify your answers.

$$\frac{1}{2^{\lambda/2}} \quad \frac{1}{2^{\log(\lambda^2)}} \quad \frac{1}{\lambda^{\log(\lambda)}} \quad \frac{1}{\lambda^2} \quad \frac{1}{2^{(\log\lambda)^2}} \quad \frac{1}{(\log\lambda)^2} \quad \frac{1}{\lambda^{1/\lambda}} \quad \frac{1}{\sqrt{\lambda}} \quad \frac{1}{2^{\sqrt{\lambda}}}$$

4.3. Suppose f and g are negligible.

(a) Show that $f + g$ is negligible.

(b) Show that $f \cdot g$ is negligible.

(c) Give an example f and g which are both negligible, but where $f(\lambda)/g(\lambda)$ is not negligible.

4.4. Show that when f is negligible, then for every polynomial p, the function $p(\lambda)f(\lambda)$ not only approaches 0, but it is also negligible itself.

Hint: Use the contrapositive. Suppose that $p(\lambda)f(\lambda)$ is non-negligible, where p is a polynomial. Conclude that f must also be non-negligible.

4.5. Prove that the $\approx$ relation is transitive. Let $f, g, h : \mathbb{N} \to \mathbb{R}$ be functions. Using the definition of the $\approx$ relation, prove that if $f \approx g$ and $g \approx h$ then $f \approx h$. You may find it useful to invoke the *triangle inequality*: $|a - c| \leqslant |a - b| + |b - c|$.

4.6. Prove Lemma 4.6.

4.7. Prove Lemma 4.7.

★ 4.8. A *deterministic* program is one that uses no random choices. Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are two *deterministic* libraries with a common interface. Show that either $\mathcal{L}_1 \equiv \mathcal{L}_2$, or else $\mathcal{L}_1$ & $\mathcal{L}_2$ can be distinguished with advantage 1.

4.9. Algorithm $\mathcal{B}$ in Section 4.4 has worst-case running time $O(q^2)$. Can you suggest a way to make it run in $O(q \log q)$ time? What about $O(q)$ time?

4.10. Assume that the last 4 digits of student ID numbers are assigned uniformly at this university. In a class of 46 students, what is the **exact** probability that two students have ID numbers with the same last 4 digits?

Compare this exact answer to the upper and lower bounds given by Lemma 4.10.