CS 198 Fall 2022

Codebreaking

Ryan Cottone, Imran Khaliq-Baporia

Note 5

Randomness

At the heart of many cryptographic algorithms lies a reliance on a stream of secure random bits – where normal PRNGs fall short. Also known as cryptographically-secure psuedorandom number generators, these systems are extraordinarily complex in nature and are a large attack surface for those looking to break a cryptosystem.

Definition 1: Psuedo-Random Number Generators

A **psuedorandom number generator**, also known as PRNG for short, is a deterministic, software-based random number generator. Given a starting seed x, it is *guaranteed* that all generated numbers will be fixed regardless of any other factors.

Definition 2: Cryptographically-Secure Psuedo-Random Number Generators

A **cryptographically-secure psuedorandom number generator**, also known as CSPRNG, is a PRNG that is random enough to be used in cryptographic applications, alongside some key security properties.

First, how does a PRNG work? Why can't computers just come up with random numbers? The answer lies in a computer's biggest strength – its rigid, logical nature. A computer can only execute preset instructions – there is no way for it to "invent" randomness (entropy). Without outside influence, there is no way to create a non-deterministic RNG, and as such is impossible to use for cryptography.

Such a drawback is why computer-based RNGs are called PRNGs, as they aren't *really* random – they just appear random to the naked eye. However, as we will shortly see, there exist a multitude of hidden patterns within some RNGs.

A key property of CSPRNGs is **rollback resistance**, which is the property of being infeasible to figure out previous RNG states given a state at time t. Informally, if the attacker learns the PRNG state at some time, they still can't figure out what was output before that time.

Uses of RNGs

Why do we need RNGs to begin with? As you've seen from Note 2, a source of truly random (or at least really close to truly random) bits is of utmost important for many algorithms. In fact, you'd be very hard pressed to find a modern cryptosystem that did not use random bits somewhere. Key generation, choosing prime numbers, generating salts, and more rely upon a secure source of

random bits.

A famous example of a PRNG's widespread use (and consequent importance of remaining secure) was the NSA's Dual_EC_DRGB. This was a PRNG that was recommended for use by the National Institute of Standards and Technology (NIST), which is a major authority on the security of modern cryptosystems (potential conflicts of interest within the US government are left as a thought exercise to the reader). As such, many companies such as RSA Security used it to implement their cryptographic algorithms.

Later, it was discovered that the NSA had introduced a backdoor into the system in which a part that was considered computationally infeasible to reverse was actually trivial for a party knowing a certain constant. This vulnerability would have allowed them to break the encryption of millions of users.

Methods of RNG

There are two primary sources of random number generation in modern computers – software RNG and hardware RNG.

Software RNG relies on existing computer hardware and pre-programmed instructions to generate psuedorandom numbers, whereas hardware RNG monitors random physical phenomena and converts it to random bits. This phenomena may include thermal noise, hard drive RPM, and sometimes quantum physics. While these phenomena are technically predictable, in practice it is impossible for an attacker to know the exact physical state down to subatomic particles, and is thus secure in theory (we will later see how this security does not always hold in practice).

Definition 3: Software RNG

Software RNG uses programs and existing hardware to generate psuedorandom numbers. They cannot produce their own entropy.

Definition 4: Hardware RNG

Hardware RNG involves using dedicated hardware to collect entropy from outside physical processes theoretically not related to computer execution.

Hardware RNGs are rare outside of mission-critical cryptographic systems such as those used by cloud providers or government agencies. However, there is growing inclusion of these modules to consumer systems.

Now, how do software RNGs actually generate "random" numbers? The most common method is the linear congruential generator, which is a fancy term for a function like this:

$$f_{n+1} = \alpha f_n + \beta \pmod{n}$$
$$\alpha, \beta, n \in \mathbb{Z}$$

The most notable drawback of such a system is its cap at n, that is, it can never generate a number

equal or greater to n. If an adversary were to discover this n, it leads to an immediately reveal of the state values α, β. We can setup a system of equations like so:

$$s_2 \equiv \alpha s_1 + \beta \mod n$$
$$s_3 \equiv \alpha s_2 + \beta \mod n$$

and solve them to reveal the secret values.

Moreover, this system is trivally weak to rollback attacks. If the attacker compromises our system and learns f_n, α, β, they can always compute $f_{n-1} \equiv \alpha^{-1}(f_n - \beta) \mod n$, eventually constructing all previous outputs.

Relation to Information Theory

Randomness is inherently tied to information theory, like we covered in a previous note. We can measure the "strength" of a CSPRNG by it's level of **seeded entropy**, or how much entropy the original seed had. If we can assume the CSPRNG functions properly, its outputs will be only as random as its initial seeding. This is why having a high-entropy seed is crucial! A low-entropy seed will allow attackers to more easily predict the outputs, or even learn the state altogether.

For example, there are many **password-based key derivation functions** (PBKDFs) that allow a user to generate infinitely many keys based off one original password. The popular PBKDF2 uses HMACs to generate each key. An easy (yet suboptimal) example PBKDF might be defined as such, using a hash function:

PBKDF(password) = H(password)

Since the hash function produces psuedorandom output, this is more or less a PRNG with an initial seeding of the password.

Unfortunately, most passwords have awful entropy. A huge amount of users use passwords known to be common, and more still use words from a dictionary as part of the password. This means attackers can guess passwords extremely fast!

Fortunately, there is a way to make guessing passwords more expensive, and therefore increase the time required to successfully crack a password. We can make our hash function **more expensive**, meaning computing the hash of a guess takes a longer time than usual. Since each guess requires a hash, a expensive-enough function makes attacks impractical.

The base hash function is usually quite cheap, taking microseconds on some architectures. To make it more expensive, we can just *iterate* the function as such:

ExpensiveHash(X) = H(H(H(H(H(...H(x)))))) (1)

usually for upwards of a thousand iterations!

Exploiting Weak Randomness

Let's consider the simplified, but plausible situation in which you are trying to break the PRNG of a computer in order to attack something else in the future. You know that it sets it's state bits as a concatentation of the following:

- CPU temperature at 1 second after boot
- UNIX timestamp at boot
- Process ID of the PRNG process

If one were to reconstruct the initial state, it is possibly to determine the output of the PRNG forever. In this case, the initial state is considerably easy to narrow down to a plausible range: CPU temperatures fluctuate a bit, but stay within a range of a few Celsius at most at boot, the timestamp is easily recovered by a root-level program, and process IDs follow a predictable pattern. Therefore, you might only have to guess a few hundred possible configurations at most due to its low entropy, compared to the "full" initial state of 2^{64} possibilities or so.

Contributors:

- Ryan Cottone