Randomness

Why do we need randomness?

Need for Randomness

- stochastic simulations
- randomized algorithms
- distributed algorithms
- statistical sampling
- testing programs
- games (chance and procedural generation)
- generating encryption keys
- generating unpredictable numbers

Fundamental Problem

- computers are very predictable
  - machine operations are deterministic
  - will do exactly same steps at boot
    - a room of identical computers will boot at same time
- without any user interaction, there will be no difference
- random numbers you generate will always be the same
  - unless you introduce something from environment

Key foundation of security is random numbers

Key foundation of security is random numbers

cryptographic key

IVs

authentication cookie

PIN numbers

second factor SMS codes

Types of Randomness

- true randomness
  - randomness from good sources
  - unpredictable from all information both before or after
    - to an information theoretic attacker
  - independent randomness
  - e.g., coin flips
- pseudo randomness
  - numbers that "look" random but may fail some statistical tests
    - i.e., can be predicted with the right information
  - numbers not independent from other generated numbers

If you've used random numbers before
you may have "seeded" it using the "time".

If you've used random numbers before
you may have "seeded" it using the "time".
This provides an initial unique value to create the stream

If you've used random numbers before
you may have "seeded" it using the "time".
This provides an initial unique value to create the stream
Resulting stream looks very different as long as seed is different

Pseudorandom Numbers

- pseudo random number generators (PRNGs) generate a stream of pseudo random using a **seed** and an **algorithm**
  - the **seed** should be unique (nonce) per use of random stream generation
    - stream can be arbitrarily long
  - the same seed and algorithm generates the same stream
  - the algorithm generates a stream of numbers from the seed that are pseudorandom
- two types: **cryptographically suitable** and **not cryptographically suitable**

Cryptographically suitable is a requirement on the **algorithm** that a computationally bounded adversary cannot distinguish a PRNG's random stream and a true random stream.

Cryptographically suitable is a requirement on the **algorithm** that a computationally bounded adversary cannot distinguish a PRNG's random stream and a true random stream. That is, nothing in the PRNG's output stream gives away that it is not true randomness.

Cryptographically suitable is a requirement on the **algorithm** that a computationally bounded adversary cannot distinguish a PRNG's random stream and a true random stream. That is, nothing in the PRNG's output stream gives away that it is not true randomness. Now a small seed can be generated with true randomness, and used to make an unlimited amount of pseudorandomness.

Cryptographically suitable is a requirement on the **algorithm** that a computationally bounded adversary cannot distinguish a PRNG's random stream and a true random stream. That is, nothing in the PRNG's output stream gives away that it is not true randomness. Now a small seed can be generated with true randomness, and used to make an unlimited amount of pseudorandomness. Seed must be unpredictable. (Why?)

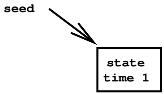
PRNG State

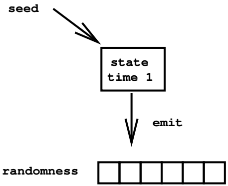
- the seed is used to randomize the state
- the state is then used to generate random numbers
- if you know the state, you can generate the sequence of random numbers
- basic requirement for cryptographically suitable PRNG
  - random numbers do not reveal the state

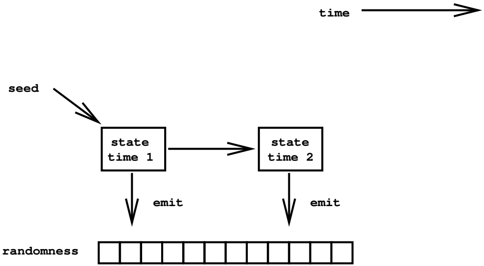
PRNG Cryptographic Properties

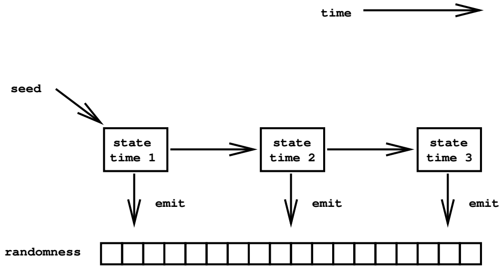
- **prediction** resistant
  - looking at enough random numbers you cannot guess the next ones
  - i.e., even without learning the state
- **rollback** resistant
  - even if you know the current state, you can't learn previous numbers generated by the stream until it reached this state
  - how can this be implemented?

seed

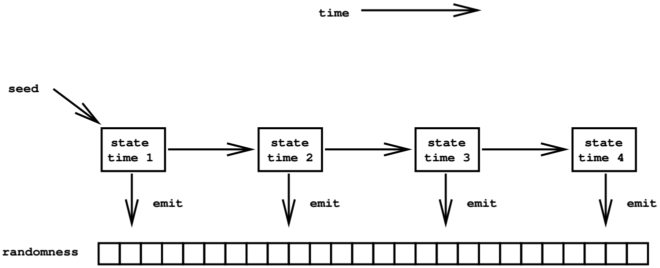

seed
state
time 1

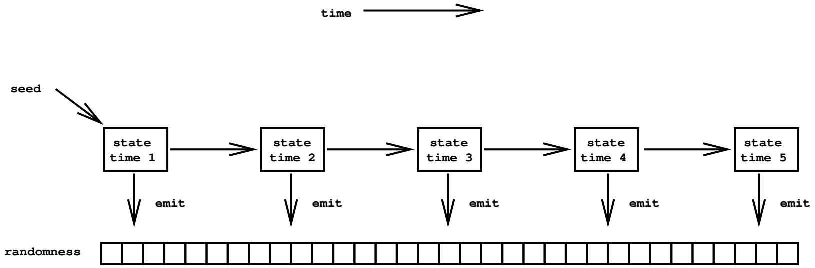

seed
state
time 1
emit
randomness

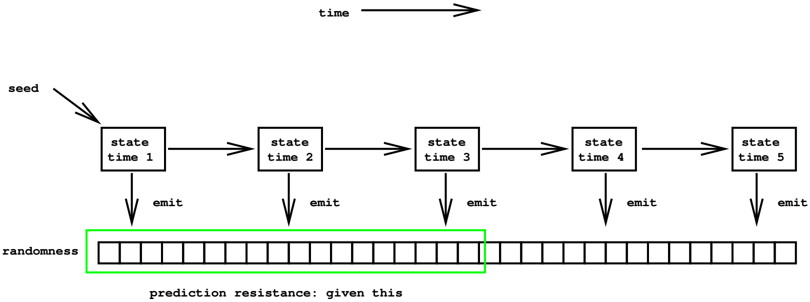

time
seed
state
time 1
state
time 2
emit
emit
randomness

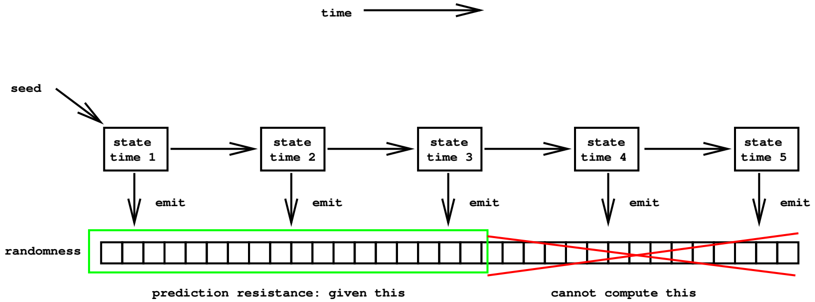


time
seed
state
time 1
state
time 2
state
time 3
emit
emit
emit
randomness

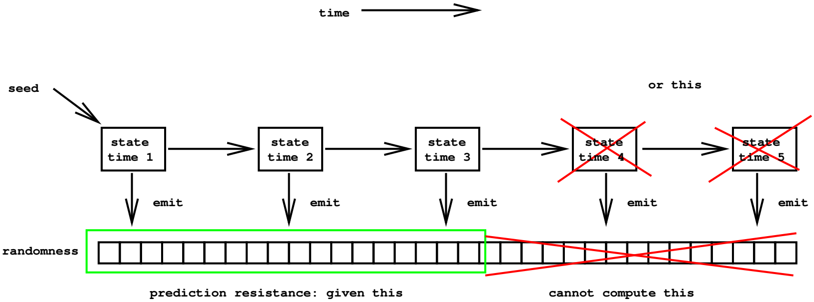

time
seed
state
time 1
state
time 2
state
time 3
state
time 4
emit
emit
emit
emit
randomness

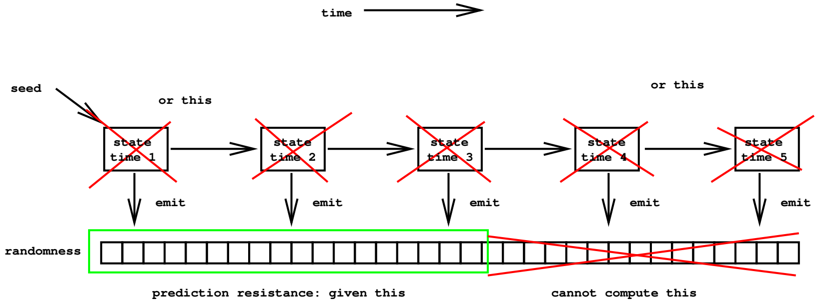

time
seed
state
time 1
state
time 2
state
time 3
state
time 4
state
time 5
emit
emit
emit
emit
emit
randomness

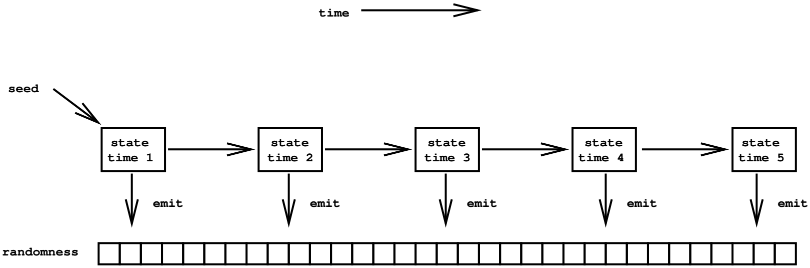

time
seed
state
time 1
state
time 2
state
time 3
state
time 4
state
time 5
emit
emit
emit
emit
emit
randomness
prediction resistance: given this

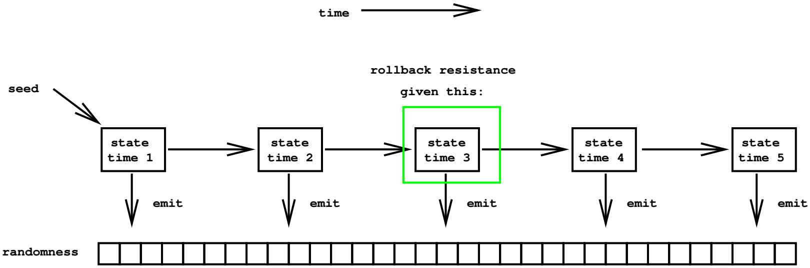


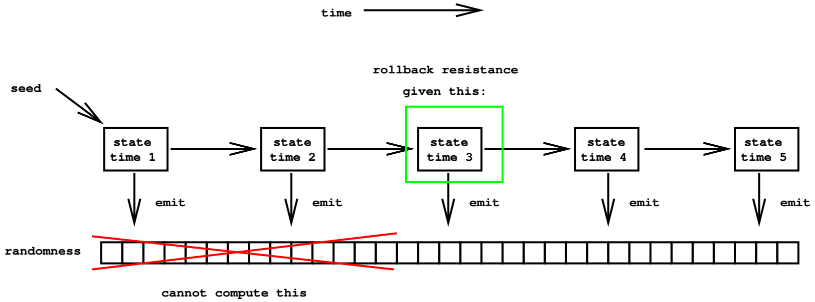
time
seed
state
time 1
state
time 2
state
time 3
state
time 4
state
time 5
emit
emit
emit
emit
emit
randomness
prediction resistance: given this
cannot compute this


time
seed
or this
state time 1
state time 2
state time 3
state time 4
state time 5
emit
emit
emit
emit
emit
randomness
prediction resistance: given this
cannot compute this

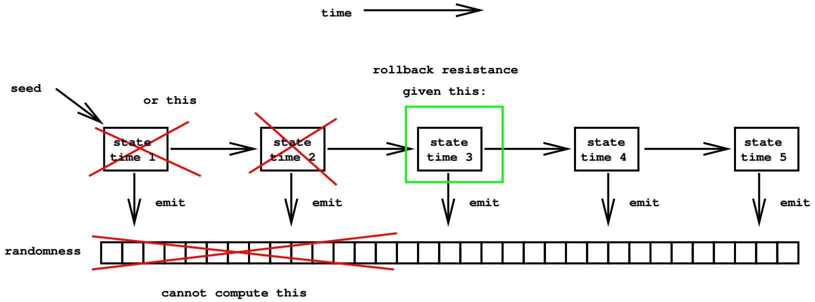

time
seed
or this
or this
state
time 1
state
time 2
state
time 3
state
time 4
state
time 5
emit
emit
emit
emit
emit
randomness
prediction resistance: given this
cannot compute this


time
seed
state
time 1
state
time 2
state
time 3
state
time 4
state
time 5
emit
emit
emit
emit
emit
randomness


time
rollback resistance
given this:
seed
state
time 1
state
time 2
state
time 3
state
time 4
state
time 5
emit
emit
emit
emit
emit
randomness



time
rollback resistance
given this:
seed
state time 1
state time 2
state time 3
state time 4
state time 5
emit
emit
emit
emit
emit
randomness
cannot compute this


time
rollback resistance
given this:
seed
or this
state
time 1
state
time 2
state
time 3
state
time 4
state
time 5
emit
emit
emit
emit
emit
randomness
cannot compute this

Why is Rollback Resistance Useful?

Why is Rollback Resistance Useful?
suppose PRNG is bad, or implementation screws up
a typical use for crypto is generate key, then generate IV

Why is Rollback Resistance Useful?

suppose PRNG is bad, or implementation screws up

a typical use for crypto is generate key, then generate IV

IV is not encrypted when sent!

Non-Cryptographically Suitable PRNG

- functions like `rand()` are not cryptographically secure
  - e.g., linear congruence generators ($y = x \cdot p (\mathrm{mod}\ n)$)
  - given enough samples, you can start predicting the next ones, figure out seed
  - seed is usually earlier timestamp: `srand(time(NULL))`
- if the seed is predictable then the stream is also predictable
  - e.g., using the time as the seed
  - does not mean algorithm isn't cryptographically suitable, only misused

True random seed with a cryptographically suitable PRNG yields a cryptographically suitable pseudo randomness

Recall some **stream ciphers** are simply a cryptographically suitable PRNGs that generates a one-time pad to XOR with the plaintext.

AES in Counter mode is simply using a random key K and a random initial counter x and creating a stream $E\_k(x), E\_k(x+1), E\_k(x+2)\ldots.$

AES in Counter mode is simply using a random key K and a random initial counter x and creating a stream $E\_k(x), E\_k(x+1), E\_k(x+2)\ldots$.
Such PRNG cannot (practically) "run out" of randomness, but are vulnerable if the seed becomes known or AES is broken.

AES in Counter mode is simply using a random key K and a random initial counter x and creating a stream $E\_k(x), E\_k(x+1), E\_k(x+2)\ldots.$

Such PRNG cannot (practically) "run out" of randomness, but are vulnerable if the seed becomes known or AES is broken.

Is it rollback resistent?

What needs Randomness

- random numbers for cryptographic reasons: unpredictable
  - one-time pads, encryption keys, random challenges
  - much of day-to-day Internet security relies on random numbers
  - when it needs to be **unguessable**
- random numbers for super important things: coin flips
  - e.g., long-lived high-stakes keys for banks
- random numbers for other purposes: uniformly distributed
  - salts, challenges, nonces, initialization vectors, identifiers
  - if should be **unique** then cryptographically suitable not needed
    - randomness based on time not guaranteed to be unique!
    - i.e., time is a function every computer is trying to match
- or better yet just always use cryptographically secure randomness
  - LEAST SURPRISE and USABILITY, supports SAFE DEFAULTS

Good Sources of Cryptographically Suitable Randomness

- observations of physical phenomenon
  - dice rolling, coin flipping, radioactive decay
- hardware events
  - time between keystrokes
  - mouse movements
  - I/O events
  - device interrupts
- hard for external observers to also measure

What about using network packet arrival times, like microseconds between packet arrivals or a round-trip-time?

Anything that can be measured, predicted, or known cannot be used for cryptographic purposes.

Anything that can be measured, predicted, or known
cannot be used for cryptographic purposes.
e.g., current time, local port of a socket,
serial number or MAC address.

Anything that can be measured, predicted, or known
cannot be used for cryptographic purposes.
e.g., current time, local port of a socket,
serial number or MAC address.
This is true even if it started
as a good true-random source.

A MILLION Random Digits

WITH

100,000 Normal Deviates

RAND

06750	40432	33289	53985	30223	99287	52542	72912	77573	27159	26506
06751	48600	14984	50460	06979	25631	97125	16982	90361	96453	15722
06752	11189	98129	91287	82537	90248	10023	56746	24623	31181	80059
06753	82316	89002	53626	21874	61847	04496	67587	12189	58463	93421
06754	20997	65043	19874	78083	89039	48212	54892	47055	09491	44986
06755	24868	96203	07440	98795	90568	25508	10526	22537	25312	62816
06756	79646	71527	12128	43803	16018	16469	92482	04983	53133	76609
06757	01562	09355	69618	82265	56912	77086	50164	20086	03033	60328
06758	62042	15129	52659	58010	03278	05591	06380	75136	99776	85437
06759	08952	76364	68661	69705	39860	01057	98245	58755	32656	40209
06760	71237	22621	91084	73853	47087	36389	86074	26221	45873	15734
06761	31028	57629	10485	60319	55288	61759	40195	78713	77972	72633
06762	54547	73653	61245	31210	80410	39461	29668	94090	59069	91895
06763	12422	40249	61006	89635	08355	40869	61353	41854	00699	98145
06764	99540	92572	80528	47006	80136	41888	17425	72036	51519	80601
06765	50085	92052	27539	81547	79065	49154	80614	32645	22853	38408
06766	56766	10191	33487	58048	06931	06446	34136	05200	16627	30567
06767	37492	46786	95895	05658	12147	76745	36801	78810	18451	83331
06768	89205	52190	84478	48627	79878	91016	90237	75868	74138	31045
06769	15562	98178	93768	92288	34055	18545	51795	62616	30585	24928
06770	43381	37908	16691	32867	84331	83465	40413	59342	10225	85335
06771	18129	14799	94983	42371	38489	93004	53677	28898	44325	32571
06772	83082	16929	16647	44438	21222	63172	63264	21951	73674	70956
06773	64213	16717	88091	66883	45369	07242	10120	30736	04284	78621
06774	24810	68101	86650	79566	42871	92969	72307	01124	00641	44763
06775	56980	54463	62382	06004	57984	22092	70100	97659	56582	40347
06776	17696	47872	45494	26901	13451	73439	18105	30875	52358	30045
06777	61877	07645	15919	45203	57170	29080	60885	91882	69552	31836
06778	02520	93145	62292	18862	38012	47524	34204	82442	45104	22418
06779	98342	51123	05400	98351	06674	53367	33763	57348	62806	47774
06780	02652	55557	25685	39051	54410	76284	35735	72568	80939	08575
06781	70215	12348	52486	72305	51980	32868	58705	77197	36140	45481
06782	87602	88456	85683	47416	55353	08875	26927	73700	92852	78477
06783	84685	96266	90199	77595	52574	89909	78356	64473	60334	99544
06784	17039	40915	56856	16023	65355	36957	85772	90294	75213	83497
06785	30412	93464	23975	02631	08840	26510	31129	66076	90114	62945
06786	29260	56176	87737	60667	21401	76766	45862	81831	88846	80776
06787	87517	34165	00825	43652	17719	64302	66857	01991	35811	45889
06788	57405	27107	48438	59909	99800	22244	14783	08766	49465	05350
06789	27585	04368	67020	26268	36506	98213	72454	55329	78967	56724
06790	11997	06214	93328	99860	62997	97363	80919	76764	69087	93475
06791	30213	55553	74372	48850	86751	02979	29212	62519	51702	50298
06792	50739	96291	57363	46271	84866	60839	15590	13072	42023	64080
06793	47922	56709	38179	28103	66983	10368	41073	74074	13511	37505
06794	72146	89466	66108	25880	34574	39968	56710	48873	16957	96974
06795	49415	90116	26527	99006	35727	76348	09585	28933	19189	75033
06796	23904	19350	96318	85926	51104	62840	46496	69171	31203	67161
06797	08807	57755	65595	37637	71377	67331	37900	87251	18500	05818
06798	42442	89072	07213	25761	15609	47073	20593	44571	92787	88223
06799	89954	69860	21229	60015	24505	21852	75919	05531	72493	00954

06800	31827	80191	43585	20270	74558	45961	90052	02750	82718	27982
06801	92204	68347	84735	32061	47876	42152	89344	83827	44440	61944
06802	72608	47319	85449	66261	38104	76120	66100	86843	17497	79969
06803	71181	34112	21904	22894	46802	68260	67676	37401	50290	46941
06804	30238	58381	06203	10840	07664	84061	78870	19046	94038	74214
06805	97806	63153	46986	88540	26772	51091	60132	13542	29098	02527
06806	68901	15231	70325	54459	74210	33350	67053	03497	00764	59007
06807	51517	35148	82482	85693	34742	79244	54316	59097	05238	71302
06808	86035	69002	34342	01936	91700	87950	36445	27181	94249	35572
06809	40704	12550	78982	10013	72214	98454	63763	75478	24327	74597
06810	99130	52082	16513	04318	44844	62677	52651	92644	60732	82781
06811	71335	76694	81253	49676	62672	77020	33251	77045	66312	20038
06812	13116	26616	14165	91983	19943	51088	33249	54613	76240	99180
06813	97727	69794	70411	30598	83133	74095	05019	92651	23968	39257
06814	55499	59891	93900	73882	25113	59288	43088	23301	32577	32791
06815	68114	62784	03503	02342	33585	79067	62339	67327	50998	48054
06816	10644	70253	87979	40870	51988	92913	41660	58484	48654	81809
06817	63563	42705	55463	28808	32894	93355	85519	85878	05904	85119
06818	50696	67283	43473	16233	06090	37324	02533	41551	86849	63729
06819	38518	61790	07851	50846	59824	61794	38329	18693	74317	87486
06820	29835	05742	96097	41131	44163	56513	17119	69246	05420	06509
06821	81722	66318	35983	03825	65327	00154	32181	50676	88628	92081
06822	76493	58045	96750	07129	28694	33174	95029	09974	53959	79355
06823	49335	20556	69838	18227	50454	68776	00591	81478	95160	32618
06824	32626	25525	16717	87974	58254	09435	16945	70276	45279	49740
06825	31413	49624	17412	92485	88605	17066	49553	43131	83541	54640
06826	30882	36088	10376	15157	23479	92796	08852	98101	43943	44458
06827	41294	09786	32189	23357	72569	43449	42922	91977	57528	49302
06828	17888	24568	43374	48671	62219	17537	23896	10865	64795	21522
06829	84534	85828	24040	62091	52814	00627	38812	37041	53031	62063
06830	84770	38718	43464	28531	51519	98086	26105	98067	75509	03821
06831	57412	03967	67914	47176	77597	98060	53675	83472	08001	75477
06832	64826	46172	01491	06483	17601	86795	48441	79485	38804	89016
06833	70411	41221	57763	52366	06071	32907	65560	31382	38259	13439
06834	52345	55303	85463	56129	92052	58633	91461	13604	36921	23004
06835	89904	07019	11723	27044	91405	04809	58411	56670	09970	31461
06836	79283	35627	79392	14301	64037	26769	21626	82401	36774	88633
06837	48682	88664	43008	32795	31584	98842	23352	88054	24483	93679
06838	76037	32852	87414	96027	98954	42626	80580	93418	71767	88077
06839	17517	46860	09293	41303	06117	13912	46878	38007	08537	27833
06840	83388	12208	91115	21707	13677	90780	32243	09065	21672	39205
06841	55719	99726	72750	18190	51008	70429	34917	50515	86410	87268
06842	24435	18058	05772	72162	34926	62984	78068	00540	12552	72151
06843	54699	57233	62385	34763	55021	47298	60832	32583	42662	00153
06844	10678	53085	81841	14499	40856	34563	60072	29819	05728	72342
06845	59680	53378	61676	67807	03084	19757	93934	80627	44152	21253
06846	44014	55930	28617	75065	82315	92855	00405	22571	77823	38423
06847	33995	38895	35776	76418	62458	17011	44858	36430	38343	31087
06848	75524	91815	79153	32915	41471	14944	69944	17231	15907	48228
06849	68239	39437	42908	78396	31568	38097	86315	14236	46656	90679

If Eve can predict the next random bit that Alice chooses even **slightly** better than random guess, it is a bad source of randomness.

Randomness for Linux

- Linux effectively has one random device: /dev/urandom
  - /dev/random deprecated in 2020, retained for compatibility
  - /dev/random was true randomness
  - /dev/urandom was pseudorandomness
  - both now work the same
  - uses ChaCha20 stream cipher as a PRNG
- randomness from kernel's entropy pool
  - timings between interrupts
  - user input such as keyboard and mouse
  - hardware random number generators if available
  - disk access timing
  - kernel jitter

Accessing Randomness

- formerly one would open /dev/random pseudofile and read it
  - this runs a program whose output appeared as it were contents of file
  - standard concet in UNIX systems where "everything is a file"
- security issue: files can change
  - user has no proof /dev/random is doing what it should
    - in containers or chroot environments, may not exist or be configured
    - file was removed and replaced with something else
    - file is not "ready" early in the boot process
    - program may "run out" of file descriptors to open
- preferred method: getrandom system call
  - ssize_t getrandom(void buf, size_t buflen, unsigned int flags)
  - blocks until random device is ready

Sourcing Randomness

- mixing multiple sources reduces attacker knowledge or control
  - randomness from keyboard and mouse
    - i.e., human activity measured at very fine time scales
  - also events in the operating system
    - e.g., network and disk activity that is human driven
- servers may not get useful HID events for randomness
  - racks of identical servers running identical workloads won't differ

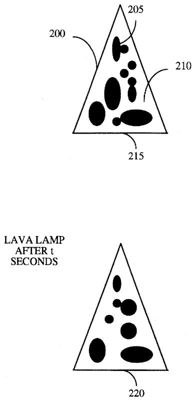


FIG. 2

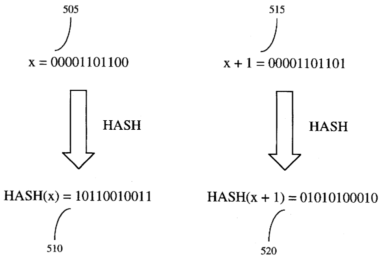

505
x = 00001101100
HASH
HASH(x) = 10110010011
510
515
x + 1 = 00001101101
HASH
HASH(x + 1) = 01010100010
520

FIG. 5

At Cloudflare each server on boot obtains fresh entropy from a LavaRand service.

At Cloudflare each server on boot obtains
fresh entropy from a LavaRand service.
This is done over TLS (we'll cover later) but importantly...

At Cloudflare each server on boot obtains
fresh entropy from a LavaRand service.
This is done over TLS (we'll cover later) but importantly...
TLS needs randomness to work in initial key exchange.

At Cloudflare each server on boot obtains
fresh entropy from a LavaRand service.
This is done over TLS (we'll cover later) but importantly...
TLS needs randomness to work in initial key exchange.
Each machine configured with its own set of secrets
one is used as an HMAC key to generate tags
on the current timestamp in nanoseconds
for TLS randomness in order to fill the entropy pool

Denoising Bias

Suppose you had a coin that flips heads two thirds odds and tails one third.

Suppose you had a coin that flips heads two thirds odds and tails one third.
How do you get random numbers without bias?

Suppose you had a coin that flips heads two thirds odds and tails one third.

How do you get random numbers without bias?

flip twice, two heads is $\frac{4}{9}$, two tails is $\frac{1}{9}$ and one of each is $\frac{2}{9}$ both ways.

Suppose you had a coin that flips heads two thirds odds and tails one third.

How do you get random numbers without bias?

flip twice, two heads is $\frac{4}{9}$, two tails is $\frac{1}{9}$

and one of each is $\frac{2}{9}$ both ways.

say two heads is heads,

one of each is tails,

and two tails is a fail (repeat).

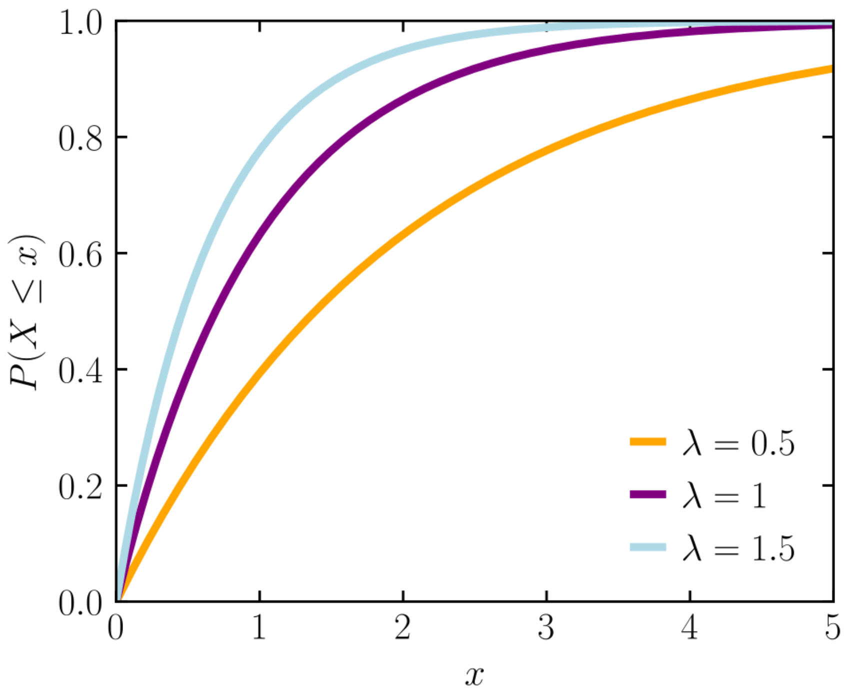
What about a coin with unknown static bias?
that is, a X:1-X bias.

What about a coin with unknown static bias?
that is, a X:1-X bias.
Can you get 50:50 random numbers without learning X?

What about a coin with unknown static bias?
that is, a X:1-X bias.
Can you get 50:50 random numbers without learning X?
probability of one of each is the same for both configurations regardless of X

Suppose you get random numbers from
a known continuous distribution,
e.g., exponential distribution.

Suppose you get random numbers from
a known continuous distribution,
e.g., exponential distribution.
How can you extract uniform randomness from it?



$P(X \leq x)$
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5
x
$\lambda = 0.5$
$\lambda = 1$
$\lambda = 1.5$

UUID: Universally Unique Identifier (rfc 4122)

- when you need a “name” for something
- and this name must be unique
- formatted as hyphen separated chunks of hexadecimal
  - 00112233-4455-6677-8899-aabbccddeeff
- three ways to generate them

Random UUID

Random UUID

use a random value (called version 4)

is 122 bits of randomness and some version information

```java
/**
 * Static factory to retrieve a type 4 (pseudo randomly generated) UUID.
 *
 * The {@code UUID} is generated using a cryptographically strong pseudo
 * random number generator.
 *
 * @return  A randomly generated {@code UUID}
 */
public static UUID randomUUID() {
    SecureRandom ng = Holder.numberGenerator;

    byte[] randomBytes = new byte[16];
    ng.nextBytes(randomBytes);
    randomBytes[6]  &= 0x0f;  /* clear version        */
    randomBytes[6]  |= 0x40;  /* set to version 4     */
    randomBytes[8]  &= 0x3f;  /* clear variant        */
    randomBytes[8]  |= 0x80;  /* set to IETF variant  */
    return new UUID(randomBytes);
}
```

Name-derived UUID

- hash a string to generate it
- use MD5 (version 3) or SHA-1 (version 5)
- deterministic but unique for a particular value
    - e.g., file name is name UUID based on its contents

```java
/**
 * Static factory to retrieve a type 3 (name based) {@code UUID} based on
 * the specified byte array.
 *
 * @param  name
 *         A byte array to be used to construct a {@code UUID}
 *
 * @return  A {@code UUID} generated from the specified array
 */
public static UUID nameUUIDFromBytes(byte[] name) {
    MessageDigest md;
    try {
        md = MessageDigest.getInstance("MD5");
    } catch (NoSuchAlgorithmException nsae) {
        throw new InternalError("MD5 not supported", nsae);
    }
    byte[] md5Bytes = md.digest(name);
    md5Bytes[6]  &= 0x0f;  /* clear version        */
    md5Bytes[6]  |= 0x30;  /* set to version 3     */
    md5Bytes[8]  &= 0x3f;  /* clear variant        */
    md5Bytes[8]  |= 0x80;  /* set to IETF variant  */
    return new UUID(md5Bytes);
}
```

Time-based UUID:

- the reason for the strange shape of the UUID
- uses a timestamp defined as follows:
  - 60-bit count of 100-nanosecond intervals since 15 Oct 1582"
- XXXXXXXX-YYYY-ZZZZ-VVVV-WWWWWWWWWWWW
  - X is low value of timestamp
  - Y is middle value of timestamp
  - Z is high value of timestamp and version
  - V is clock sequence number
    - incremented each time the system clock changes
  - W is node id (MAC address of the system that generated)
    - 48 bit value
    - reveals the host that generates it, but "guaranteed" to be unique

Example of Bad Randomness

Dual_EC_PRNG

- Dual_EC_DRBG is a PRNG based on elliptic curves
- relied on two parameters, P and Q
- if chosen in a particular way one can predict random numbers
  - basic attack: 30 output bytes revealed PRNG's internal state

Dual_EC_DRBG

- it was criticized by experts for its poor design shortly after publication
- it was thousands of times slower than existing simpler secure alternatives
- there was bias in the output bytes
  - i.e., it failed the most basic test of a useful PRNG
- it was known that for any P, there would be a specific Q backdoor
  - impossible for anyone to prove they didn't know backdoor

It was clear this is a terrible PRNG.

It was clear this is a terrible PRNG.
And no one wanted to use it.

It was clear this is a terrible PRNG.
And no one wanted to use it.
No one did. Case Closed.

It was clear this is a terrible PRNG.
And no one wanted to use it.
No one did. Case Closed.
Except...

Exclusive: Secret contract tied NSA and security industry pioneer

By Joseph Menn

9 MIN READ

SAN FRANCISCO (Reuters) - As a key part of a campaign to embed encryption software that it could crack into widely used computer products, the U.S. National Security Agency arranged a secret $10 million contract with RSA, one of the most influential firms in the computer security industry, Reuters has learned.

Dual_EC_DRBG

- RSA accepted 10 million dollars from NSA in a secret deal
  - used a P and Q that the NSA recommended
  - implemented it and made it the default PRNG
- RSA then pushed it as a NIST standard and it was put into other products
- NSA could then break security since a backdoor PRNG is now widespread

Broken Dual_EC_DRBG

- if you had backdoor, looking at 40 bytes of randomness would reveal state
- it was also not rollback resistant!
  - once you know state, you can go backwards and forwards

Versions: 00 01 02

Network Working Group   E. Rescorla
Internet-Draft   RTFM, Inc.
Intended status: Informational   M. Salter
Expires: September 3, 2009   National Security Agency
                                                            March 02, 2009

Extended Random Values for TLS
draft-rescorla-tls-extended-random-02.txt

If you generate encryption keys and then generate "benign" random values later that you send in plaintext, NSA can determine your state and rollback to the key.

Sabotage Magic Numbers

- many implementations of crypto rely on specific "magic" numbers to work
- if these are chosen without clear justification it leaves room for such attacks
- "nothing-up-my-sleeve" numbers are one chosen to be by construction above suspicious of hidden properties
  - should have a low Kolmogorov complexity
  - should be hard to "tweak" into any other number using the same approach

```
// Initialize variables:
var int a0 := 0x67452301   // A
var int b0 := 0xefcdab89   // B
var int c0 := 0x98badcfe   // C
var int d0 := 0x10325476   // D
```

AES S-box

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

The column is determined by the least significant nibble, and the row by the most significant nibble. For example, the value $9a_{16}$ is converted into $b8_{16}$.

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

where $[s_7, \ldots, s_0]$ is the S-box output and $[b_7, \ldots, b_0]$ is the multiplicative inverse as a vector.

This affine transformation is the sum of multiple rotations of the byte as a vector, where addition is the XOR operation:

$$s = b \oplus (b \lll 1) \oplus (b \lll 2) \oplus (b \lll 3) \oplus (b \lll 4) \oplus 63_{16}$$

where b represents the multiplicative inverse, $\oplus$ is the bitwise XOR operator, $\lll$ is a left bitwise circular shift, and the constant $63_{16} = 01100011_2$ is given in hexadecimal.

An equivalent formulation of the affine transformation is

$$s_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$$

where s, b, and c are 8 bit arrays, c is 01100011_2, and subscripts indicate a reference to the indexed bit.[3]

Another equivalent is:

$$s = (b \times 31_{10} \mod 257_{10}) \oplus 99_{10}$$ [4][5]

where $\times$ is polynomial multiplication of b and 31_{10} taken as bit arrays.

Best PRNG to use: HMAC_DRBG

Best PRNG to use: HMAC_DRBG
state is a key k from seed and v of data

Best PRNG to use: HMAC_DRBG
state is a key k from seed and v of data
uses HMAC three times per round
one for randomness, two to update state

```
function hmac_drbg_generate (state, n) {
  tmp = ""
  while(len(tmp) < N){
     state.v = hmac(state.k,state.v)
     tmp = tmp || state.v
  }
  // Update state with no input
  state.k = hmac(state.k, state.v || 0x00)
  state.v = hmac(state.k, state.v)
  // Return the first N bits of tmp
  return tmp[0:N]
}
```

HMAC_DRBG

- prediction resistance
  - HMAC prevents generating tags without key
- rollback resistance
  - current key is an HMAC tag on previous state
  - HMAC prevents key revelation