

Application Security (apsi)

Lecture at FHNW

Lecture 4, 2020

Arno Wagner, Michael Schläpfer, Rolf Wagner

<arno@wagner.name>, <{michael.schlaepfer,rolf-wagner}@fort-it.ch>

Agenda

Use of Cryptography

- ▶ Key generation and handling, entropy gathering
- ▶ Password Storage
- ▶ Storing and handling private data
- ▶ Disk encryption: Full-disk encryption, file encryption
- ▶ Encryption in the cloud
- ▶ Encryption in the browser

Note: This is very incomplete...

Generating Cryptographic Keys

Requires high-quality random data (actually: "unpredictable" data)

Usual approach:

- ▶ CPRNG (Cryptographic Pseudo Random Number Generator)
- ▶ Needs seeding, usually 256 bits of entropy or more (generator dependent)
- ▶ After seeding, practically unlimited amount of key-grade data available

Where to get entropy ?

- ▶ Physical processes (disk access, network events, user input, etc.)
- ▶ Store some between reboots (save on shutdown, load on start)
- ▶ Initial seeds from outside (VMs!)

→ Many errors are made here! (looks very easy – is pretty hard)

Lit: "Mining Your Ps and Qs", <https://factorable.net/weakkeys12.extended.pdf>

Information-Theoretical Entropy

Measures the "uncertainty" of a value produced "randomly"

Unit (one symbol): "bit"

Unit (stream of symbols): "bit/symbol"

Unit (Entropy-density, i.e. Entropy per bit): "bit/bit" or none

Definition:
$$E = - \sum_{i=0}^{n-1} p_i * \log_2(p_i) \quad [bit]$$

for n symbols with probabilities $p_0, \dots, p_{(n-1)}$

Example:

A source can emit 128 different symbols, all equally probable

→ Entropy per symbol: 7 bit

→ Stream-Entropy: 7 bit/symbol

Estimating Entropy is Tricky

Yet estimation is needed to make sure a CPRNG seeding is good!

→ The key is to generously over-estimate the seeding entropy

Example: User input timing on the keyboard

Question: How much entropy is there in the time-stamp of one key-press of a sequence of keys? Assume there is a 0.1sec random variation per key.

Answer: ???

Seeding a CPRNG

Example: `/dev/(u)random` (Linux standard kernel CPRNG)

- ▶ `/dev/random`: Only gives data until estimated seeding entropy reached
→ may block a long time (not good)
- ▶ `/dev/urandom`: Gives unlimited CPRNG output (even if badly seeded)
 - Good seeding is critical, and must be done (do not use it before!)
 - After seeding, `/dev/urandom` should be used in all casesSpeed should be > 200MB/s on a desktop CPU (Linux kernel 5.x or later)

Seeding (from Debian `/etc/init.d/urandom`):

```
cat "seedfile" > /dev/urandom
```

Recommendation: Seed at the very least with 256 bits of entropy

Note: There are quite a few myths around about `/dev/(u)random`

Most are wrong, some are dangerous

Unfortunately, that even includes "man random(4)" !

→ If in doubt, ask an expert!

Bad Seeding Examples

- ▶ Seeding `/dev/urandom` from `/dev/random`
 - That does not work, as both are interfaces to the same generator!
- ▶ Seeding with the system time (may/will be predictable!)
- ▶ Seeding with a single entropy source (depends)
- ▶ Just waiting a few seconds after boot and hoping for the best
- ▶ Seeding with a copied seed file (VM or system image clone)
- ▶ Java `SecureRandom()` with a broken implementation (Android...)
 - Security depends on (obscure) details. This is bad design!

Using cryptography with bad keys from badly seeded CPRNGs is worse than not using cryptography, as it gives a false sense of security!

This violates the "Principle of least surprise"

Storing Cryptographic Keys Securely

This is difficult!

Due to time restrictions not treated here.

→ If in doubt, ask an expert

Storing Passwords

Problem: User-supplied passwords need to be stored for verification

- ▶ Solution 1: Store in plain in a database
 - If hacked, attacker has all passwords!
 - Even worse, many users use the same passwords in multiple places!
- ▶ Solution 2: Use a single crypto-hash h and store $h(\text{pwd})$
 - ▶ Verification: Compare $h(\text{pwd})$ and $h(\text{user_supplied_pwd})$
 - ▶ Attacker gets only $h(\text{pwd})$

How much effort is reversing $h(\text{pwd})$?

Example: A modern PC does about 5M SHA1 hashes per second per CPU

Passwords: letters + digits + 10 special symbols → 6.2 bit entropy/char

Breaking effort (1 CPU): 4 chars → 6 sec, 6 char → 9h, 8 char → 5.4 years

But: 1. Rainbow-tables → Compute once, break many passwords fast

2. Graphics cards → may be > 1000x as fast (5.4 years → 2 days)

Rainbow Table

A table that on lookup of $h(\text{pwd})$ delivers pwd

→ Break a larger set of password-hashes (think the 500M from Yahoo)

Note: Base form, optimizations to save space are possible

Creation for a given set of passwords $\{\text{pwd}\}$:

- 1) For all elements p of $\{\text{pwd}\}$: calculate $h(p)$
- 2) Create a hash-table: $h(p) \rightarrow p$
Traditionally: Sort $\{h(p)\}$, and do binary search.
But: sorting $O(n \log n)$, bin-search $O(\log n)$, while hashing $O(n)/O(1)$ (usually)

Effort (single hash, random 6.2bit/char entropy passwords, estimates):

- ▶ 6 char pwd: CPU: a few days, space: ~ 2TB
- ▶ 8 char pwd: Graphics card: a few days, space: ~10EB (~1000 disks)
- ▶ 10 char pwd: May be infeasible today

Storing Passwords

- ▶ Refinement: Salted hashing to make Rainbow-Tables unusable:

Store salt, $h(\text{pwd} + \text{salt})$. Salt: Random, non-secret, 64bit or more

- One Rainbow table per salt value needed
- Storage space for Rainbow table becomes prohibitive

- ▶ Refinement: Iterated hashing, to slow-down brute-force reversal:

Instead of $h(\text{pwd})$, store $h(\dots(h(\text{pwd})\dots))$, for, say, 1 sec of hashing time

- Attacker gets massively slowed down
- Around 1M iterations for 1 sec on modern CPUs

Combination of both is the old (!) state-of the art

- ▶ PBKDF2() implements this with some refinements

Reference: RFC 2898

$\text{DK} = \text{PBKDF2}(\text{hash}, \text{password}, \text{salt}, \text{iteration_count}, \text{result_bits})$

So What is Wrong with PBKDF2?

It does not need much memory to compute it!

- ▶ Graphics cards (~64kB RAM per core for example) still work well
- ▶ ASICs and FPGAs work well

Needed is hashing + iteration + "large memory property"

- ▶ Older partial and obsolete solutions: bcrypt(), scrypt()

Current state-of-the-art: Argon2:

- ▶ Use this whenever possible
- ▶ Time and memory parameter
=> Parameters tunable to target situation
- ▶ Computing with less memory makes things massively slower
- ▶ Recommendations: CPU: ~ 1 sec, RAM: 100M or more if possible

Hardware RNGs

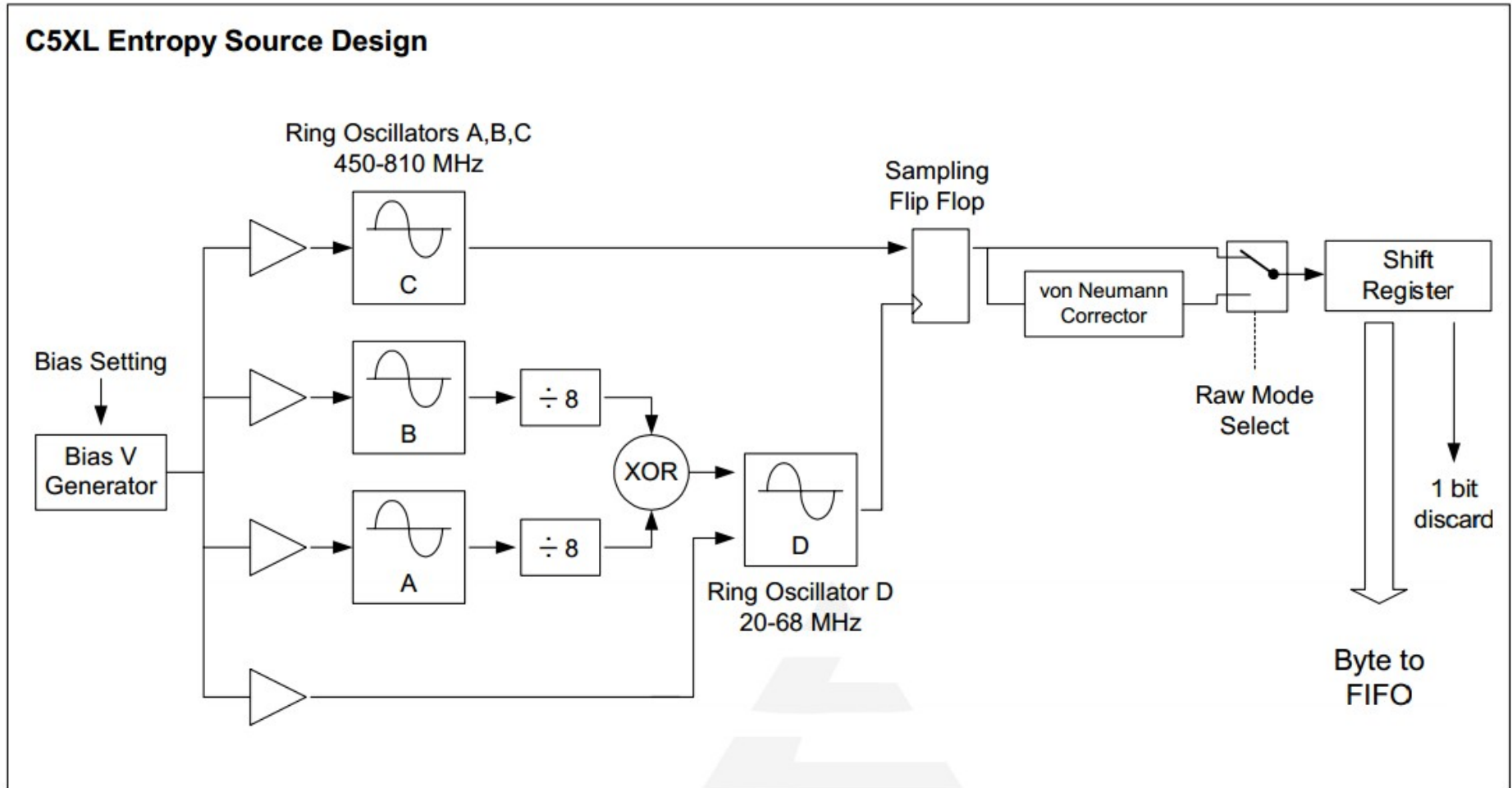
True random number generators found (for example) in some CPUs

- ▶ Produces "true" randomness (usually thermal noise and/or quantum effects)
Note: Quantum effects often called "true random", but reality is we simply do not know how they work...
- ▶ Usually have high bandwidth (for example $> 1\text{MB/s}$ random data)
- ▶ Not dependent on any external interactions or software (main advantage)
- ▶ Usually easy to use on assembler-level

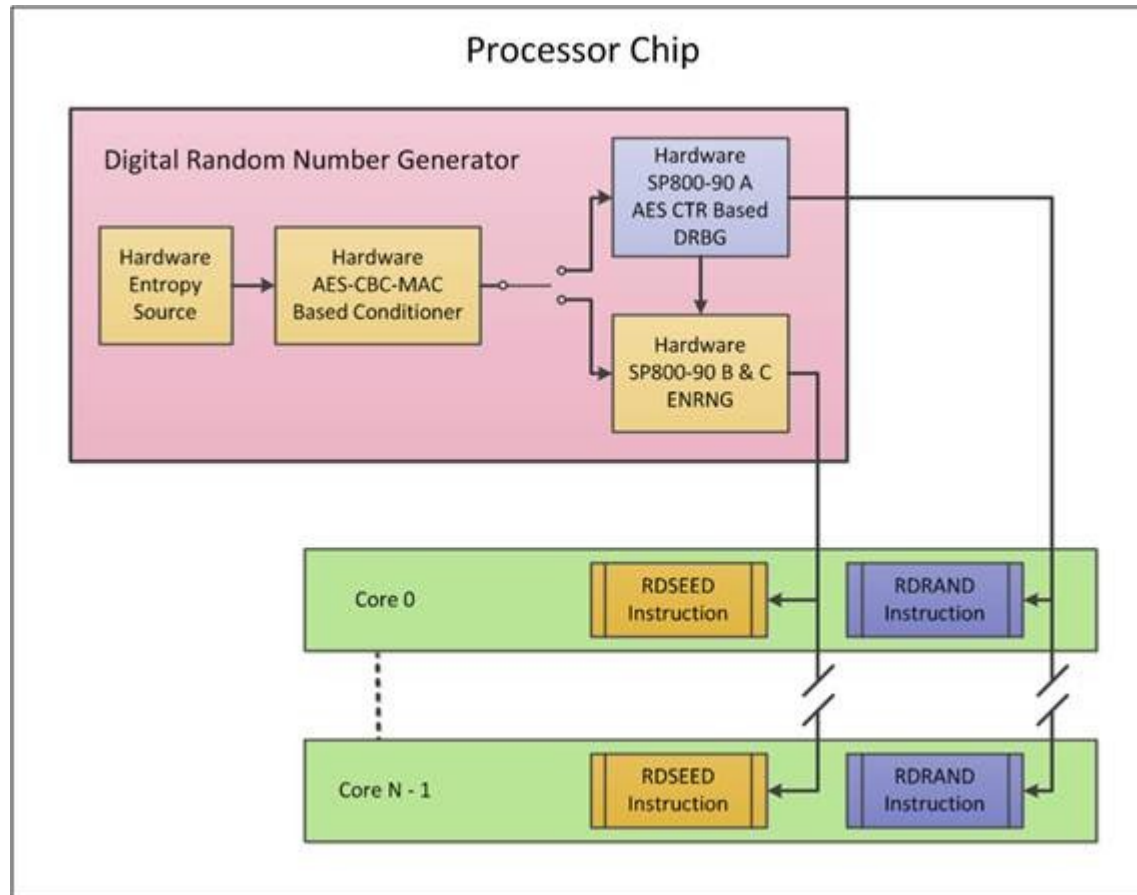
But:

- ▶ Implementation is more or less "trusted"
Note: "trusted" = "can break your security" = "you have to trust it..."
- ▶ Problematic implementations exist and are widespread

Example 1: Via C3 Hardware CPRNG Schematic



Example 2: Intel RDRAND Schematic



The Conditioner is a problem....

Compromise on Design-Level

The C3 CPRNG is a non-compromised design:

- ▶ Raw signal can be verified
- ▶ May still be implemented in a compromised fashion!
- ▶ May still be implemented as something different!
- ▶ Note: Delivers around 0.75 bit/bit in entropy

Intel RDRAND is a compromised design:

- ▶ Impossible to read and verify raw randomness
 - ▶ Impossible to distinguish compromised implementation from good one
- Do not depend on this alone!
- Linux kernel does not trust it and only uses it as additional conditioner, not as Entropy-source (update: Kernel may now be set to trust it)

Reference (archived copy): <https://archive.fo/Md6TM>

Storing Private Data (as a Service Provider)

Aim: Preserve privacy

How to do that?

- ▶ Encrypt everything!
But: Data needs to be used at some time and then it needs to be decrypted
Made worse by desires of marketing...
- ▶ Do not store private data in the first place!
What you do not have cannot be stolen or misused...
- ▶ Abstract or anonymize private data
→ May be enough for the intended use (and for marketing)
Warning: This is really difficult to get right!

Disk/Storage Encryption

Refers to encryption done by the OS (or storage device)

Example full-disk encryption (FDE):

- ▶ LUKS (Linux)

Example File-level disk encryption:

- ▶ EncFS (Linux)

Note: File-level encryption is less secure, but more flexible

- ▶ Leaks file-names, time-stamps, length, etc.
- ▶ May allow overwrite-attacks

Disk Encryption Protects Against What?

Basically only:

- ▶ Your disks being stolen
- ▶ Your non-running (!) computer being stolen
=> That does not sound so great for a server or laptop...

It does not help for:

- ▶ Being hacked while the disk is decrypted (mapped)
- ▶ The running machine with decrypted (mapped) disks is stolen

Privacy benefits are limited!

One important case though: Disks/tapes/etc. being decommissioned insecurely

Decryption Only With User Logged In

Idea: Disk encryption limited to user-presence

- ▶ Use user-password as (part/basis of) the encryption key
Wipe it on log-out

Problems:

- ▶ A patient hacker can wait for the user to log in
- ▶ User-passwords are not very secure
- ▶ If done wrongly may expose user passwords

Better approach:

- ▶ Use this, but decrypt only on user-side (but browser-crypto is still a problem)
→ This is how cloud-storage should be done (but usually is not)

Encryption in the Cloud

Usually, this means disk encryption

Protects:

- ▶ A non-running VM disk image (if done right)

Problems:

- ▶ Same problems as disk encryption
- ▶ No protection against the cloud provider
- ▶ Keys may leak to other VMs on the same hardware (numerous problems)
- ▶ Marketing, the copyright industry and the "Four Horsemen of the Infocalypse" will ensure your data does get scanned, analyzed, aggregated, etc.

→ Encryption in the cloud is basically marketing. Assume data in the cloud is only secure if it never reaches the cloud non-encrypted.

JavaScript Client-Side Encryption in Web-Apps

"JavaScript Crypto" (basically applies to all browser-executed code)

- ▶ Widely used to achieve compatibility, but pretty bad security level

Common problems:

- ▶ Browser may be arbitrarily old and/or insecure
- ▶ You must trust the browser (it can always attack you successfully)
- ▶ How do you deliver the code securely? If you can, you may not need it...
- ▶ The code is not under user-control, so hacking the server hacks it as well
- ▶ ...

Reference:

<https://web.archive.org/web/20110925120934/http://www.matasano.com/articles/javascript-cryptography/>

→ Do not rely on it. (Note: Some people strongly disagree)

- ▶ Will "WebCrypto" fix this? I do not think so: <https://tonyarcieri.com/whats-wrong-with-webcrypto>

→ This may still need a long time to become secure, if it ever does...