# Rapport préliminaire – Audit d'OpenSSL

| | |
|---|---|
| **Date** | 5 février 2014 |
| **Rédigé par** | Claire Smets, William Boisseleau, Pascal Edouard, Mathieu Latimier, Julien Legras |
| **À l'attention de** | Ayoub Otmani |

# Table des matières

# Table des figures

# Listings

# Introduction

Ceci est l'introduction

# Chapitre 1

# Entropie

## 1.1 Définitions et contexte

### 1.1.1 Introduction

Cette partie explicite les notions d'entropie nécessaires pour la définition d'aléatoire de certains programmes, mais décrit aussi les sources qui de génération de bits aléatoires et les tests liés.

Trois axes principaux sont nécessaires à la mise en place d'un générateur cryptographique aléatoire de bits :

— Une source de bits aléatoires (source d'entropie)
— Un algorithme pour accumuler ces bits reçus et les faire suivre vers l'application en nécessitant.
— Une méthode appropriée pour combiner ces deux premiers composants

### 1.1.2 Estimation de l'entropie générée par la source

Il est tout d'abord important de vérifier que la source d'entropie choisie produit suffisamment d'entropie, à un taux égalant voire dépassant une borne fixée. Pour ce faire, il faut définir avec précision la quantité d'entropie générée par la source. Il est de plus important de considérer les différents comportements des composants de la source, afin d'éliminer les interactions qu'ils peut y avoir entre les composants. En effet, ceci peut provoquer une redondance dans la génération d'entropie si cela n'est pas considéré. Étant donné une source biaisée, l'entropie générée sera conditionnée et donc plus facilement prévisible/estimable.

La source d'entropie doit donc être minutieusement choisie, sans qu'aucune interaction et conditionnement ne soit possible.

### 1.1.3 Concept d'entropie

**Définition.**
Soit $X$ est une V.A. discrète. On définit l'**entropie** de $X$ comme suit :

$$H(X) = -\sum_{x} P(X = x) * log(P(X = x))$$

Le logarithme est dans notre cas de base 2. L'entropie se mesure en shannons ou en bits.

**Définition.**

On définit le **désordre** (ou incertitude) étant liée à cette expérience aléatoire. Si l'on considère l'ensemble fini des issues possibles d'une expérience $\{v_1, ..., v_n\}$, l'entropie de l'expérience vaudra :

$$H(\epsilon) = -\sum_x P(\{a_i\}) * log(P(\{a_i\}))$$

**Propriété.**

On constate que l'entropie est maximale lorsque X est équi-répartie. En effet, si l'on considère n éléments de X étant équi-répartie, on retrouve notre entropie de $H(X) = log(n)$.

Ainsi, on comprend qu'une variable aléatoire apporte en moyenne un maximum d'entropie lorsqu'elle peut prendre chaque valeur avec une équiprobabilité. D'un point de vue moins théorique, on considère que plus l'entropie sera grande, plus il sera difficile de prévoir la valeur que l'on observe.

**Min-entropy.**

La recommandation du NIST propose le calcul de *Min-entropy* pour mesurer au pire des cas l'entropie d'une observation.

Soit $x_i$ un bruit de la source d'entropie. Soit $p(x_i)$ la probabilité d'obtenir $x_i$. On définit l'entropie au pire des cas telle que :

$$\text{Min-entropy} = -log_2(max(p(x_i))$$

La probabilité d'observer $x_i$ sera donc au minimum $\frac{1}{2^{\text{Min-entropy}}}$.

### 1.1.4 Source d'entropie

**Approche théorique.**

La source d'entropie est composée de 3 éléments principaux :

- le **bruit source**, qui est la voûte de la sécurité du système. Ce bruit doit être non déterministe, il renvoie de façon aléatoire des bits grâce à des processus non déterministes. Le bruit ne vient pas nécessairement directement d'éléments binaires. Si ce bruit est externe, il est alors converti en données binaires. La taille des données binaires générées est fixée, de telle sorte que la sortie du bruit source soit déterminé dans un espace fixe.
- le **composant de conditionnement**, qui permet d'augmenter ou diminuer le taux d'entropie reçu. L'algorithme de conditionnement doit être un algorithme cryptographique approuvé.
- une **batterie de tests**, partie également intégrante du système. Des tests sont réalisés pour déterminer l'état de santé du générateur aléatoire, permettant de s'assurer que la source d'entropie fonctionne comme attendu. On considère 3 catégories de tests :
  - Les tests au démarrage sur tous les composants de la source
  - Les tests lancés de façon continue sur le bruit généré par la source
  - Les tests sur demande (qui peuvent prendre du temps)

  L'objectif principal de ces tests est d'être capable d'identifier rapidement des échecs de génération d'entropie, ceci avec une forte probabilité. Il est donc important de déterminer une bonne stratégie de détermination d'échec pour chacun de ces tests.

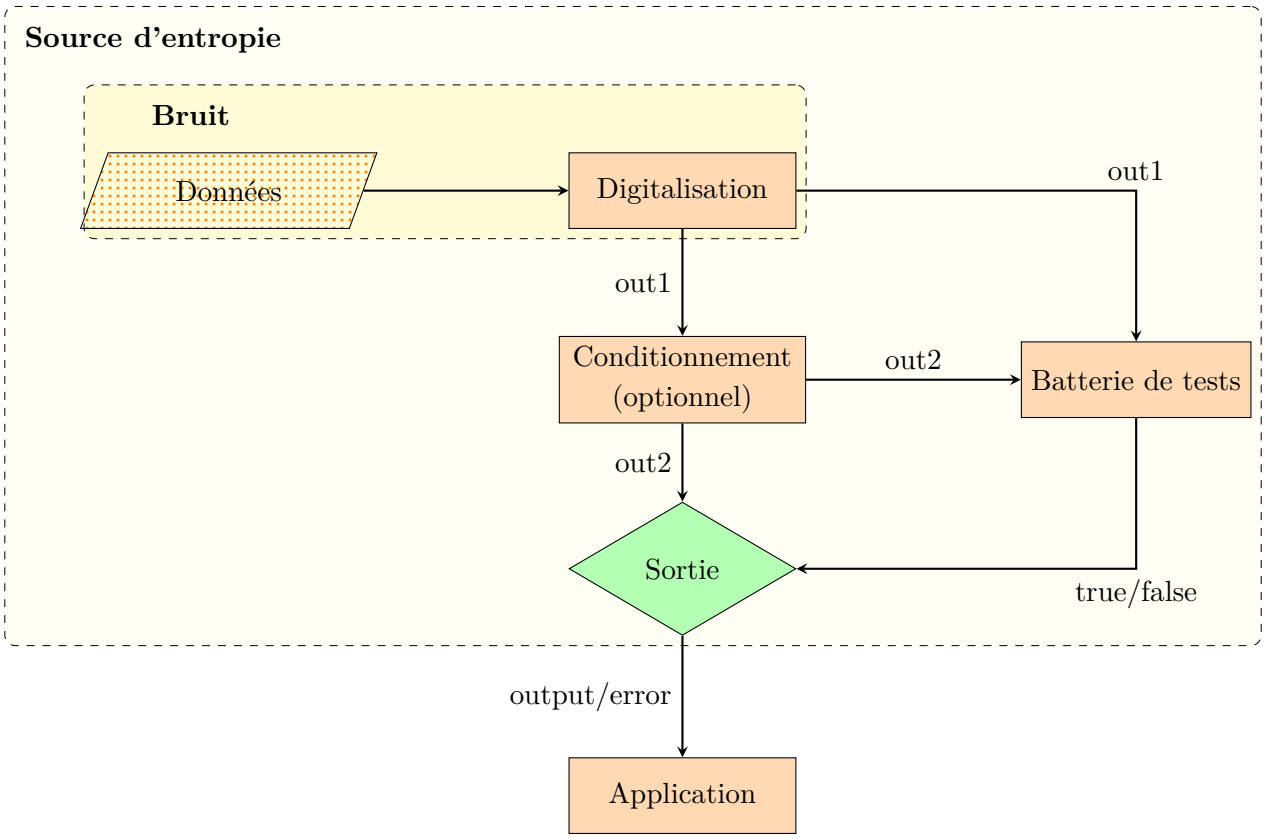


FIGURE 1.1 – Composants d'une source d'entropie. `out1` est une chaîne binaire de taille quelconque et `out2` est une chaîne binaire conditionnée de taille fixe.

**Modèle conceptuel.**

Suivant ces sections précédentes, on peut déterminer 3 interfaces conceptuelle :

- `getEntropy` qui retourne
  - `entropy_bitstring`, une chaîne de bits de l'entropie demandée
  - `assessed_entropy`, entier indiquant le nombre de bits d'entropie de `entropy_bitstring`
  - `status`, booléen renvoyant `true` si la requête est satisfaite, `false` sinon.

- `getNoise` qui prend en entrée :
  - `number_of_sample_requested`, entier indiquant le nombre d'éléments demandés en retour à la source de bruit

  et en sortie :
  - `noise_source_data`, la séquences d'éléments demandée, ayant la taille `number_of_sample_requested`.
  - `status`, booléen renvoyant `true` si la requête est satisfaite, `false` sinon.

- `HealthTest`, élément test de la batterie de tests, qui prend en entrée :
  - `type_of_test_requested`, chaine de bits déterminant le type de tests que l'on souhaite effectuer (peut différer suivant le type de source)

  et en sortie :
  - `pass-fail_flag`, booléen qui renvoie `true` si la source d'entropie a réussi le test, `faux` sinon.

### 1.1.5 Forces et faiblesses des différents PRNG

Nos machines utilise ce qu'on appel des PRNG (Pseudo Random Number Generator), ce sont des algorithmes qui génèrent une séquence de nombre s'apparentant à de l'aléatoire. En réalité rien est aléatoire car tout est déterminé par des valeurs initiales (État du PRNG) et des contextes d'utilisation.

Un bon PRNG se doit d'avoir une très forte entropie (proche de un), afin d'éviter de délivrer de l'information.

Comme l'entropie est fourni majoritairement (si ce n'est totalement) par l'OS, il est donc nécessaire de détailler les PRNG les plus utilisés (Surtout par les systèmes Linux et BSD - qui sont les ceux qui génèrent le plus de certificats SSL).

Nous nous basons sur la RFC 4086 [3] : Randomness requirements for security pour le choix des PRNG selon les différents systèmes.

#### 1.1.5.1 /dev/urandom et /dev/random sous Linux

Sous Linux, un pool est initialisé avec 512 octets, auquel on ajoute le temps émis par un évènement et son état parmis :

— Les interruptions clavier - heure et code d'interruption
— Les interruptions de disques - heure de lecture ou écriture
— Les mouvements de souris - heure et position

Quand des octets aléatoires sont demandés, la pool est haché avec SHA-1 (20 octets). S'il est demandé plus que 20 octets, le haché est mélangé dans la pool pour rehacher la pool ensuite etc. À chaque fois que l'on prend des octets dans la pool, l'entropie estimée est décrémentée.

Pour assurer un niveau minimum d'entropie au démarrage, la pool est écrite dans un fichier à l'extinction de la machine.

/dev/urandom fonctionne selon le même principe sauf qu'il n'attend pas qu'il y ait assez d'entropie pour donner de l'aléatoire. Il convient pour une génération de clefs de session.

Pour générer des clefs cryptographiques de longue durée, il est recommandé d'utiliser /dev/random pour assurer un niveau minimum d'entropie.

En effet, sur un serveur sans souris ni clavier, définir l'entropie avec /dev/urandom est très risqué. On recommande donc l'utilisation de /dev/random lors de l'audit OpenSSL sur les versions Linux.

/dev/random utilise une pool d'entropie de 4096 bits (512 octets) génère de l'aléa et s'arrête lorsqu'il n'y a plus assez d'entropie et attend que la pool se remplisse à nouveau.

Si vous souhaitez connaître l'entropie disponible, la commande est :
cat /proc/sys/kernel/random/entropy_avail

Désormais, la taille de la pool est hardcodée dans le noyau Linux (/drivers/char/random.c :275)

Linux offre également la possibilité de récupérer de l'aléa depuis un RNG matériel avec la fonction get_random_bytes_arch [1]

Un patch est également disponible afin de générer de l'aléa avec un débit de 100kB/s [14]. L'entropie est récupérée par le CPU timing jitter.

En conclusion, /dev/random doit être utilisé pour une haute qualité d'entropie (i.e. haute sécurité de chiffrement, one-time pad).
Tandis que /dev/urandom doit être utilisé pour des applications non sensibles à des attaques cryptographiques (i.e. jeu en temps réel), car elle génère plus d'entropie que /dev/random sur un temps donné, mais s'arrêtera même si il n'a pas récolté suffisamment d'entropie.

### 1.1.6 /dev/random sous FreeBSD et /dev/arandom sous OpenBSD

Il faut faire attention au faux ami, le /dev/random du FreeBSD n'est pas le même que celui de Linux. En fait, il est semblable au /dev/urandom de Linux, et est donc tout autant proscrit lors de notre audit.

Même principe avec le /dev/arandom de OpenBSD, qui a également une entropie faible pour du chiffrement cryptographique sûr. Il se base en fait sur un algorithme modifié du RC4 nommé ARC4 (Alleged RC4) pour générer des données aléatoires.
Pour rappel, RC4 était un projet commercial de la RSA Security, et un hacker anonyme a publié un code identique, devenu légitime, identifié par ARC4.
De nos jours, il est fortement conseillé de ne plus utiliser RC4 car le flux de données aléatoire n'est pas vraiment aléatoire et il existe des attaques qui prédisent la sortie de l'algorithme (Attaque de Fluhrer, Mantin et Shamir) [9].
Sur plusieurs de nos sources (plus anciennes), il est recommandé d'utiliser /dev/arandom pour sa rapidité (71 Mb/s) et sa bonne source d'entropie. Ce n'est plus vraiment le cas aujourd'hui.

### 1.1.7 CryptGenRandom sous Windows

Du coté de Microsoft, il recommande aux utilisateurs de Windows d'utiliser CryptGenRandom, qui est un appel système de génération d'un nombre pseudo-aléatoire. La génération est réalisée par une librairie cryptographique (Cryptographic service provider library). Celui ci gère un pointeur vers un buffer en lui fournissant de l'entropie afin de générer un nombre pseudo aléatoire en retour avec en plus, le nombre d'octet d'aléatoire désiré.

```
BOOL WINAPI CryptGenRandom(
  _In_    HCRYPTPROV hProv,
  _In_    DWORD dwLen,
  _Inout_ BYTE *pbBuffer
);
```

Le service provider sauvegarde une variable d'état d'un sel pour chaque utilisateur. Lorsque CryptGenRandom est appelé, celui ci est combiné avec un nombre aléatoire généré par la librairie en plus de différentes données systèmes et utilisateurs telles que :

— l'ID du processus
— l'ID du thread
— l'horloge système
— l'heure système
— l'état de la mémoire
— l'espace de disque disponible du cluster
— le haché du block d'environnement mémoire de l'utilisateur

Le tout est envoyé à la fonction de hachage SHA-1 et le nombre en sortie est utiliser comme sel pour une clef RC4.

Cette clef est enfin utilisé pour produire des données pseudo aléatoire et mettre à jour la variable d'état du sel de l'utilisateur.

### 1.1.8 Autres systèmes

Nous avons également d'autres RNG comme srandom, prandom ou encore wrandom [2]

/dev/srandom est simple et lent, il n'est pas recommandé de l'utilisé.

Certains systèmes ne disposant pas de /dev/*random, il est alors possible d'utiliser l'EGD (Entropy Gathering Daemon) [15].

Il faut pour cela utiliser les fonctions OpenSSL RAND_egd, RAND_egd_bytes et RAND_query_egd_bytes.

L'EGD est également utilisé par GPG, et peut être utilisé comme seed.

### 1.1.9 Standards

#### 1.1.9.1 RFC 4086

##### 1.1.9.1.1 Sous Linux

Suivant la RFC 4086 intitulée *Randomness requirements for security* , Il existe plusieurs niveaux de récupération d'aléatoire sous linux :

— Point primaire (*Primary pool*) :
512o (128 mots de 4o) + ajout d'entropie
— Point secondaire (*Secondary pool*) :
128o pour générer le fichier `/dev/random`. Un autre point secondaire existe : `/dev/urandom`

L'entropie est récupérée par exemple lorsqu'un événement apparaît (telle qu'une interruption du disque dur), la date et l'heure de l'événement est récupéré et XORée dans le *pool*, puis est "mélangée" avec une primitive polynomiale possédant un degré de 128. Le *pool* devient ensuite une boucle où de nouvelles données

sont XORées ("mélangées" encore par la primitive polynomiale) tout le long du *pool*.

A chaque appel qui rajoute de l'entropie dans le *pool*, celui-ci calcule une estimation de la probabilité d'une réelle entropie des données. Le pool contient alors l'accumulation des estimations de l'entropie totale contenue dans le pool.

Les sources d'entropie sont les suivantes :
— Interruption Clavier
— Interruption des complétions du disque
— Mouvements de la souris
Quand des octets aléatoires sont demandés, la *pool* est haché avec SHA-1 (20 octets). Si plus de 20 octets est demandé plus que 20 octets, le haché est mélangé dans la pool pour rehacher la pool ensuite etc. À chaque fois que l'on prend des octets dans la pool, l'entropie estimée est décrémentée. Pour assurer un niveau minimum d'entropie au démarrage, la *pool* est écrite dans un fichier à l'extinction de la machine.

`/dev/urandom` fonctionne selon le même principe sauf qu'il n'attend pas qu'il y ait assez d'entropie pour donner de l'aléatoire. Il convient pour une génération de clefs de session. Pour générer des clefs cryptographiques de longue durée, il est recommandé d'utiliser `/dev/random` pour assurer un niveau minimum d'entropie.

#### 1.1.9.1.2 Sous Windows

Du coté de Microsoft, il est recommandé aux utilisateurs d'utiliser `CryptGenRandom`, qui est un appel système de génération d'un nombre pseudo-aléatoire. La génération est réalisée par une librairie cryptographique (*Cryptographic service provider library*). Celle-ci gère un pointeur vers un *buffer* en lui fournissant de l'entropie afin de générer un nombre pseudo aléatoire en retour avec en plus, le nombre d'octets d'aléatoires désirés.

```
BOOL WINAPI CryptGenRandom(
  _In_     HCRYPTPROV hProv,
  _In_     DWORD dwLen,
  _Inout_  BYTE *pbBuffer
);
```

Le service *provider* sauvegarde une variable d'état d'un sel pour chaque utilisateur. Lorsque `CryptGenRandom` est appelé, celui-ci est combiné avec un nombre aléatoire généré par la librairie en plus de différentes données systèmes mais aussi de l'utilisateur tels que l'ID du processus du *thread*, l'horloge système, l'heure système, l'état de la mémoire, l'espace de disque disponible du *cluster* et le haché du block d'environnement mémoire de l'utilisateur.

Le tout est envoyé à la fonction de hachage SHA-1 et le nombre en sortie est utilisé comme sel de clef RC4. Cette clef est enfin utilisée pour produire des données pseudo-aléatoires et mettre à jour la variable d'état du sel de l'utilisateur.

#### 1.1.9.1.3 OpenBSD

Dans OpenBSD, on trouve des sources d'aléatoire supplémentaires par rapport à Linux. On trouve notamment `/dev/arandom` qui génère de l'aléatoire selon une version leakée de RC4 : `ARC4` (Alleged RC4). Pour rappel, RC4 était un projet commercial de RSA Security et un hacker anonyme a publié un code qui faisait la même chose, code légitime identifié par ARC4. De nos jours, il est fortement conseillé de ne plus utiliser RC4 car le flux de données aléatoires n'est pas vraiment aléatoire et il existe des attaques qui prédisent la sortie de l'algorithme (Attaque de Fluhrer, Mantin et Shamir) .

### 1.1.10 FIPS 140

Le FIPS 140 (*Ferderal Information Processing Standards* est un standard du gouvernement américain spécifique aux modules cryptographiques déployés par des éléments du gouvernement. Il inclue notamment des standards de tests qui permettent de qualifier la qualité et/ou validation des générateurs d'entropie ou générateurs pseudos aléatoires. La dernière version en présence est le FIPS 140-2, qui décrit plusieurs niveaux de tests.

Les test du FIPS 140-1 permet de s'assurer que les sources d'entropies produisent suffisamment de "bonnes" données, pourvu que les sources d'entropies n'utilisent pas quelques opérations cryptographiques internes. Si elle en utilise, la source d'entropie réussira les tests avec quasi- certitude, même si la source d'entropie est faible. Pour la même raison, ces jeux de tests ne sont pas bon pour tester des des générateurs de nombres pseudo-aléatoires cryptographiques, car ceux-ci passeront facilement les tests même si les générateurs d'entropie sont faibles. Par exemple, si l'on hache une suite d'entiers (par pas de 1), les tests seront tous validés bien qu'ils n'auront pas été tirés aléatoirement, ceci en vertu de la fonction de hachage. Pourtant, la donnée est hautement prédictible.

Les tests du FIPS 140-2 ne sont également pas très efficaces, excepté si un matériel commence à produire un motif répété. Ils consistent à comparer différentes sorties consécutives d'un générateur. De telle sorte, si le "générateur aléatoire" consiste à produire une simple incrémentation, les tests passeront sans problème.

Il est donc conseillé d'utiliser les tests du FIPS 140-1 pour vérifier uniquement si la source d'entropie produit de bonnes données, au démarrage et périodiquement lorsque c'est possible.

Le FIPS 140-1 définit 4 tests statistiques à lancer sur 20000 bits consécutifs, tests au démarrage ou à la demande :

1. Le **test Monobit**, où le nombre de bits à 1 sont comptés. Le test est considéré comme réussi si le nombre de bits à 1 est raisonnablement proche de 10000.
2. Le **test Poker**, pour lequel les données sont séparées en une suite consécutive de 4bits pour déterminer combien de fois les 16 configurations de 4 bits apparaissent. Les carrés du résultats sont sommés et permettent de définir si le test passe ou non.
3. Le **test runs**, que nous développerons dans les recommandations du NIST.
4. Le test **long runs**, qui effectue le test de runs sur 34 bits ou davantage.

Le FIPS 140-2 définit enfin des tests continuels de sortie (*continuous output tests*). Les données de sorties sont découpés en blocs de 16 octets (ou davantage). Le premier block est stocké et comparé second. S'ils sont identiques, alors le test est échoué. On passe à la paire suivante, le 2e bloc est comparé avec le 3e, etc.

## 1.2 Audits

### 1.2.1 Audit 1 : Le cas Debian 4.0 et OpenSSL 0.9.8

#### 1.2.1.1 Norme visée

#### 1.2.1.2 Faille

##### 1.2.1.2.4 Description

Le 13 Mai 2008, Luciano Bello découvert une faille critique du paquet d'OpenSSL sur les systèmes Debian [4]. Un mainteneur Debian souhaitant corriger quelques bugs aurait malencontreusement supprimé une grosse source d'entropie lors de la génération des clés.
Il ne restait plus que le PID comme source d'entropie !
Comme celui-ci ne pouvait dépasser 32.768 (qui le PID maximal atteignable), l'espace des clés a été restreint à 264.148 clés distinctes.

Analysons plus en détail cette faille. Elle se situe au niveau de la fonction **md_rand.c**. La ligne **MD_Update(&m, buf, j);** a été commentée. La conséquence est le blocage de la graine (seed) que l'on passe ensuite au PRNG.

Cette ligne a été commentée par erreur en voulant corriger un avertissement soulevé par le compilateur Valgrind sur une valeur non initialisé.

Le 14 Mai 2008, Steinar H. Gunderson démontre simplement comment en connaissant le secret $k$ d'une signature, on peut retrouver la clé privée d'un certificat immédiatement [11].

Ce secret k étant généré avec un PRNG prévisible, on peut stocker deux signatures utilisant le même $k$, où le prédire directement.

Une signature DSA consiste en deux nombres $r$ et $s$ tels que :
$r = (g^k[p])[q]$
$s = (k^{-1} * (H(m) + x * r))[q]$

La clé publique $= (p, q, g)$.
Le message en clair $= m$, et $H(m)$ est le fingerprint de $m$ connu.

Attaque n° 1 : En connaissant $k$
$s * k[q] = (H(m) + x * r)[q]$
$\iff (s * k - H(m))[q] = x * r[q]$
$\iff ((s * k - H(m)) * r^{-1})[q] = x$
$\iff (s * k - H(m)) * r^{-1} = x$

Attaque n° 2 : Deux messages possèdent le même $k$
$s_1 = (k^{-1}(H(m_1) + x * r))[q]$
$s_2 = (k^{-1}(H(m_2) + x * r))[q]$
$\iff s_1 - s_2 = (k^{-1}(H(m_1) - H(m_2))[q]$

$\iff (s_1 - s_2) * (H(m_1) - H(m_2))^{-1} = k^{-1}[q]$
$\iff$ On connaît $k \implies$ Attaque 1

Pour savoir si une clé SSL, SSH, DNSSEC ou OpenVPN est affectée, plusieurs détecteur de données [7] [8] de clés faibles sont fournis par l'équipe Security de Debian, en même temps que l'avertissement de sécurité [6].

#### 1.2.1.2.5 Tests

Nous avons décidé de tester le nombre de certificats vulnérables causés par le bug OpenSSL de Debian (qui reste le plus populaire), et connaissant la blacklist des clés privés.

Les résultats nous montrent que sur 500.000 certificats récupérés, au moins [1] 769 sont vulnérables.

Vous pouvez trouver nos scripts parcourant un fichier contenant un certificat sur chaque ligne, ou un dossier contenant des certificats sous forme de fichiers PEM et nos résultats, dans le dossier consacré à l'audit des clés cryptographiques.

Le format de nos résultats est :

**COMPROMISED :** *<haché_du_certificat> <nom _fichier_corrompu (sous forme d'adresse IP)*

Évidemment, nous ne mettons pas ces résultats sur le net puisqu'il indique très clairement les adresses IP contenant le certificat friable, et sa clé privée (que l'on peut facilement retrouvé parmi la courte blacklist).

Pour information, parmi les entreprises vulnérables nous trouvons les géants IBM et CISCO.

### 1.2.1.3 Implémentation

**Version OpenSSL.**

**Fonction.**

La fonction liée à cette norme est accessible sous le paquetage `bla/bla/bla`, dont les composantes principales sont listées en *listing* 3.1.

```
#include <stdio.h>
#define N 10
/* Block
 * comment */

int main()
{
    int i;

    // Line comment.
    puts("Hello world!");
```

1. Le logiciel ne prend pas en compte les clés $\leq$ à 512 bits et celles $\geq$ à 4096 bits, et ne prend en compte que les certificats RSA

```
13      for (i = 0; i < N; i++)
        {
15          puts("LaTeX is also great for programmers!");
        }
17
        return 0;
19  }
```

Listing 1.1 – codeAleatoire.c

**Audit.**

### 1.2.2 Audit 2 : Le cas LinuxMintDebianEdition sous Android

#### 1.2.2.1 Norme visée

#### 1.2.2.2 Faille

Récemment, en Août 2013 précisément, un patch de sécurité pour les systèmes Android utilisant la version LinuxMintDebianEdition/OpenSSL, dévoile une réparation du générateur de nombres pseudo-aléatoire (PRNG) qui ne donnait pas suffisamment d'entropie [12] [13].

Le patch indique que le PRNG de cette version d'OpenSSL utilise dorénavant une combinaison de données plus ou moins prévisibles associées à l'entropie générées par /dev/urandom.
Mais sachant que le PRNG d'OpenSSL utilise lui-même /dev/urandom, on a du mal à comprendre pourquoi en rajouter davantage.

Eric Wong et Martin Boßlet apporte la solution sur leur site [5], l'erreur provient d'un bug "à la Debian", une simple ligne diffère de la version officielle d'OpenSSL (utilisant SecureRandom) à celle de OpenSSL : :Random ce situant dans la fonction ssleay_rand_bytes.

La cause est là même que celle de Debian, un patch de sécurité atteint la source d'entropie du PRNG. Alors que tout semblait être rentré dans l'ordre, un résidu de cette erreur reste dans cette version. Les développeurs d'OpenSSL assure que ça n'a pas d'impact (ou alors très peu) sur la sécurité globale. Mais à cause de la mémoire non initialisé des systèmes Android, la source d'entropie ne nous permet pas de générer des nombres non-prédictibles.

La conséquence n'est pas aussi lourde que celle de Debian, tout d'abord parce que le système Android est rarement utilisé pour du chiffrement de données sensible, et une attaque par prédiction bien que plus rapide qu'une attaque par brute-force, reste infaisable. Mais l'erreur est quand même là.

#### 1.2.2.3 Implémentation

**Version OpenSSL.**

**Fonction.**

La fonction liée à cette norme est accessible sous le paquetage `bla/bla/bla`, dont les composantes principales sont listées en *listing* 3.1.

```c
#include <stdio.h>
#define N 10
/* Block
 * comment */

int main()
{
    int i;

    // Line comment.
    puts("Hello world!");

    for (i = 0; i < N; i++)
    {
        puts("LaTeX is also great for programmers!");
    }

    return 0;
}
```

Listing 1.2 – codeAleatoire.c

**Audit.**

### 1.2.3 Audit 3 : Le cas LinuxMintDebianEdition sous Android

#### 1.2.3.1 Norme visée

#### 1.2.3.2 Faille

C'est le syndrome OpenSSH de Debian qui frappe une nouvelle fois.
Du fait d'une parenthèse mal placée dans le code du fichier /src/sys/kern/subr_cprng.c, il s'avère que le générateur pseudo-aléatoire de NetBSD 6.0 est bien moins solide que ce qui était attendu.
Sa sortie est prévisible et il faut d'urgence changer les clés SSH qui ont été générées avec ce noyau.

#### 1.2.3.3 Implémentation

**Version OpenSSL.**

**Fonction.**

La fonction liée à cette norme est accessible sous le paquetage `bla/bla/bla`, dont les composantes principales sont listées en *listing* 3.1.

```c
#include <stdio.h>
#define N 10
/* Block
 * comment */

int main()
{
    int i;

    // Line comment.
    puts("Hello world!");

    for (i = 0; i < N; i++)
    {
        puts("LaTeX is also great for programmers!");
    }

    return 0;
}
```

Listing 1.3 – codeAleatoire.c

**Audit.**

## 1.3 Recommandations sur la conception

Le NIST propose des recommandations à plusieurs niveaux concernant la conception de la source d'entropie. Nous n'en reporterons ici que les recommandations génériques. Les éléments ci-dessous sont basés

### 1.3.1 Modélisation et validation

La source d'entropie doit suivre les exigences suivantes

1. Le développeur doit documenter complètement la modélisation de la source d'entropie, incluant toutes les interactions entre les composants. De ce fait, la documentation doit pouvoir justifier en quoi la source d'entropie est de confiance.
2. Il doit définir de façon précise les limites conceptuelles de sécurité de la source d'entropie, qui doivent elles mêmes être équivalentes à un module cryptographique délimitant un périmètre de sécurité (cf. *Cryptographic module boundary* du FIPS 140).
3. Le développeur doit définir le champ des conditions optimales de fonctionnement du générateur d'entropie.
4. La source source d'entropie doit être possiblement validée suivant les recommandations du FIPS 140, ainsi que les tests s'y référant.

5. Le comportement du bruit de la source doit être documenté, validant en quoi le taux d'entropie ne fluctue pas lors d'une utilisation normale.
6. Dès lors qu'un test de validation n'est pas réussi, la source d'entropie doit immédiatement cesser d'envoyer des données de sortie, et doit notifier à l'application l'erreur rencontrée.

   Enfin, une recommandation optionnelle :
7. La source d'entropie doit contenir différent types de bruits pour améliorer le comportement global de la source, et prévenir des tentatives de contrôle externe. Chaque bruit doit vérifier les spécifications du 3.1.

### 1.3.2 Source d'entropie absolue

Certains générateurs de bits aléatoire demandent une source d'entropie absolue, à savoir qu'elle approxime une sortie qui est uniformément distribuée et indépendante des autres sorties.

1. La chaine de bit générée doit fournir au moins $(1 - \epsilon)n$ bits d'entropie, où :
   — $n$ est la taille de chaque sortie
   — $\epsilon$ tel que : $0 \leqslant \epsilon \leqslant 2^{-64}$

### 1.3.3 Bruit de la source

Le bruit fournit par la source doit suivre les recommandations suivantes :

1. Le bruit doit avoir un comportement probabiliste et ne doit en aucun cas être définissable par quelconque algorithme ou règle.
2. Le développeur doit pouvoir documenter l'opération sur le fonctionnement du bruit de la source, en montrant en quoi le choix de ce bruit fournit une sortie d'entropie acceptable. Ceci comprend le référencement d'articles de recherche et autre littérature pertinente.
3. Le bruit de la source doit pouvoir être testable, de telle sorte que l'on puisse s'assurer qu'il effectue l'opération attendue. Il doit donc être possible de récupérer des données sur la source de bruit sur lesquels on puisse lancer la batterie de tests. La récupération des données sur la source de bruit ne doit en aucun cas altérer le comportement du bruit, ou de la sortie.
4. Toute défaillance du bruit doit être rapidement détectable. Les méthodes de détection doivent être documentées.
5. La documentation de la source du bruit doit également décrire sous quelles conditions le bruit est connu pour mal fonctionner. Ceci consiste ainsi à répertorier les environnements dans lesquels la source peut fonctionner correctement.
6. La source de bruit doit être protégée au maximum contre toute attaque/tentative de conditionnement ou simple connaissance de fonctionnement de la part d'un adversaire.

### 1.3.4 Composant de conditionnement

Le composant de conditionnement doit suivre les recommandations suivantes :

1. Le développeur doit documenter si la source d'entropie nécessite ou non un conditionnement.
2. La méthode de conditionnement doit être décrite et argumentée. Elle doit en effet expliciter en quoi elle permet de diminuer l'alignement d'une source de bruit, ou en quoi la sortie créée correspond à l'entropie attendue

3. La méthode de conditionnement doit pouvoir être validée par des tests
4. Le développeur doit pouvoir estimer l'alignement en sortie de conditionnement.
5. Le développeur doit prévoir une documentation expliquant le comportement de la méthode de conditionnement en cas de variation de comportement de la part de la source de bruit.

### 1.3.5 Batterie de tests

Globalement, les tests doivent suivre les recommandations suivantes :

1. Les tests doivent être effectués au démarrage puis de façon continuelle pour s'assurer que les composants du générateur d'entropie fonctionnent correctement.
2. Tous les tests doivent être documentés, en particulier sur les conditions sous lesquels ils doivent être exécutés, les résultats attendus pour chacun de ceux-ci, et une explication rationnelle indiquant en quoi chaque test est approprié pour la détection de dysfonctionnement de la part de la source d'entropie.

#### 1.3.5.1 Tests sur le bruit

Les tests sur le bruit sont pour la plupart du temps dépendants de la technologie utilisée. Dans la majorité des cas, il faut tester via des procédures traditionnelles de tests (type test monobit, test du $\chi_2$, et tests d'exécution) si celui-ci est bien non biaisé et produit des données indépendantes.

1. Au minimum, des tests continus doivent être implémentés, et ceci de façon indépendante. Le développeur doit de plus documenter toutes les sources de défaillance.
2. Les tests sont implémentés sur les données binaires récupérées via le bruit source.
3. La source de bruit doit être testé dans son ensemble (suivant la variation du bruit).
4. Le bruit généré durant la démarrage ayant passé avec succès les tests de démarrage peut être utilisé pour produire de l'entropie.
5. Lorsqu'un test est échoué, le générateur doit en être notifié.

   Optionnel :

6. Une étude peut être requise sur les bords du bruit bruit généré, dans les cas ou le comportement du générateur est altéré.

#### 1.3.5.2 Tests sur le conditionnement

Le rôle du conditionnement est de réduire l'alignement présent chez certaines sources d'entropie afin de s'assurer que l'entropie est construite à un taux acceptable. Les recommandations sont les suivantes :

1. Les composants de conditionnements doivent être testés dès le démarrage, afin d'être certain que le générateur fonctionne comme prévu
2. Le développeur doit documenter les tests implémentés et inclure les conditions d'échecs pour chacun d'entre eux.

## 1.4 Tests effectifs sur l'entropie fournie par les sources d'entropie

### 1.4.1 Déterminer si les données sont IID

**Définition.**
On dit que des variables sont **IID** (indépendantes et identiquement distribuées) si elle suivent toute la même loi de probabilité et si elles sont mutuellement indépendantes.

Les tests suivants ont été conçus pour montrer si les données en sortie du bruit et/ou en sortie du conditionnement sont bien IID, ceci en effectuant des tests sur la distribution des données. Le but est de vérifier l'hypothèse $H_0$. Pour la vérifier, nous allons considérer ici deux types de tests :

— Des tests de répartition aléatoire sur l'ensemble
— Des tests statistiques

Si un des tests est passé, alors on passe au suivant. La défaillance d'un seul des tests impliquera le caractère non IID des données.

#### 1.4.1.1 Tests sur l'indépendance et la stabilité

Nous travaillons sur le test statistique bilatéral suivant :
— l'hypothèse $H_0$ : "Les données sont IID".
— l'hypothèse $H_1$ : "Les données ne sont pas IID".

Suivant un jeu de données ($|donnees| = n$), on divise celui-ci en 10 sous ensemble de tailles égales ($\frac{N}{10}$). On effectue enfin nos tests sur chacune de ces données. La stratégie de test est effectuée comme suit :

1. On calcule tout d'abord différents scores statistiques parmi les suivants :
   — Score de compression, un score par sous ensemble de données
   — Score *Over/Under Runs*, deux scores par sous ensemble
   — Score excursion, un score par sous ensemble
   — Score *Runs* directionnel, trois scores par sous ensemble
   — Score de covariance, un score par sous ensemble
   — Score de collision, trois scores par sous ensemble

2. Pour chaque calcul de score :
   (a) On stocke pour chaque type de score les résultats dans un vecteur de $J$ scores
   (b) On répète les étapes suivantes 1000 fois :
      i. On permute les sous ensembles précédents en utilisant un générateur pseudo aléatoire, suivant l'algorithme de permutation *Fisher–Yates shuffle*.
      ii. On calcule les nouveaux scores suivant cette nouvelle organisation de sous-ensembles
      iii. On récupère ce vecteur de $J$ scores
   (c) On classifie les scores de 2a) en le comparant avec tous les scores liés permutés. Par exemple, si le score des données originelles est plus grand que tous ceux des permutations, alors ce premier a le score de 1000, et les répertoires permutés on un score inférieur à 1000. Il est possible que les données permutées aient le même score. Lorsque le score originel est le même que certains

de scores des données permutées, on considère le score en récupérant celui le plus proche de la médiane.

Plus généralement, étant donné :
— Un score $S$
— Une liste de scores des données permutées $L$
On détermine le **rang de** $S$ tel que :

$$Rang(S) = \begin{cases} max(j) \text{ tel que } L[j] \leqslant S, & si \quad L[500] > S & (1) \\ 500 \quad si \quad L[500] = S & & (2) \\ min(j) \text{ tel que } L[j] \geqslant S, & si \quad L[500] < S & (3) \end{cases}$$

i. Exemple cas (1) :
— Soit $S = 20$ et $L[500] = 22$. On est dans le cas (1). On va chercher le $max(j)$ tel que $L[j] \leqslant S$.
— On a $L[299] = 19$, $L[300] = 20$, $L[301] = 20$, $L[302] = 22$.
— On retourne au score 301, ainsi $Rang(S) = 301$.

ii. Exemple cas (3) :
A. Soit $S = 20$ et $L[500] = 18$. On est dans le cas (3). On va chercher le $max(j)$ tel que $L[j] \geqslant S$.
B. On a $L[599] = 19$, $L[600] = 20$, $L[601] = 20$.
C. On retourne au score 600, ainsi $Rang(S) = 600$.

3. Le rang alors obtenu est considéré comme une *p-value* pour un test bilatéral. Pour rappel, une *p-value* est la probabilité d'obtenir la même valeur de test si l'hypothèse nulle est vraie. Ainsi, on compte pour chacun des 6 tests un ensemble de $10 * J$ *p-values*.

4. On gère enfin les *p-values* de la façon suivante :
(a) Tous les rangs compris entre 50 et 950 sont relevés. Ces événements ont une probabilité d'arriver de 10% au total.
(b) Si le nombre de rangs relevés est supérieur ou égal à 8, on considère le test comme échoué.
i. Si le test est effectué sur la source du bruit, alors celle-ci n'est pas considérée comme IID
ii. Si le test est effectué sur le conditionnement, alors la source d'entropie n'est pas validée.

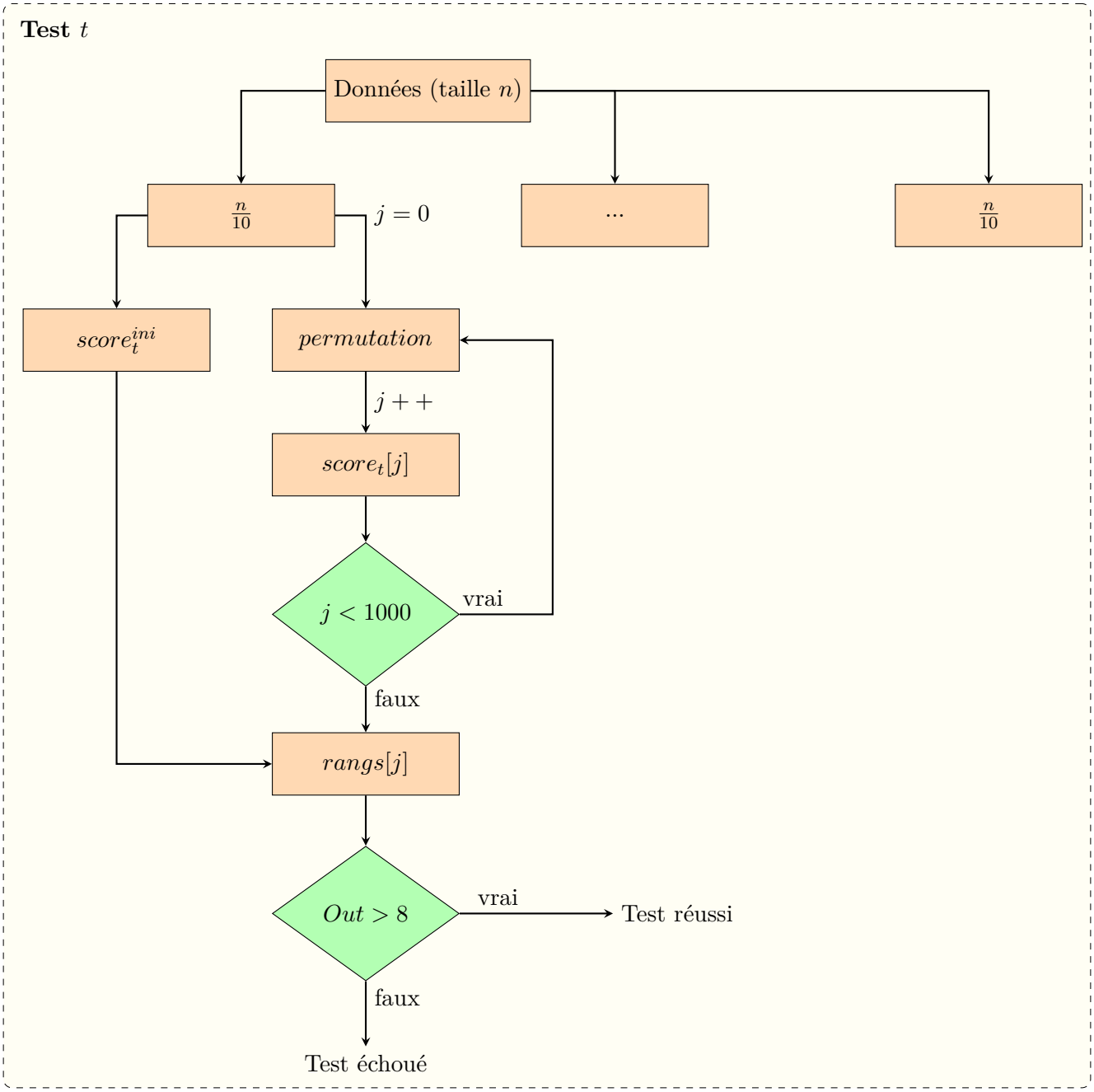


FIGURE 1.2 – Structure de validation de données $d$ par un test $t$. $Out$ vaut "$|Rangs \leq 50$ && $Rangs \geq 950|$"

#### 1.4.1.1.6 Score de compression

Les algorithmes de compressions sont habituellement bien adaptés pour supprimer les données redondantes des chaînes de caractère. Suivant un algorithme de compression choisi, le score de compression est la longueur de la donnée compressée obtenue.

**Calcul du score.**

Le score est calculé de la façon suivante :

1. Les sous ensembles de données sont encodés en chaîne de caractère séparés par une virgule

2. La chaîne de caractère est compressée suivant l'algorithme de compression de bzip2 [2]
3. Le score retourné correspond à la longueur de la chaîne de caractère compressée

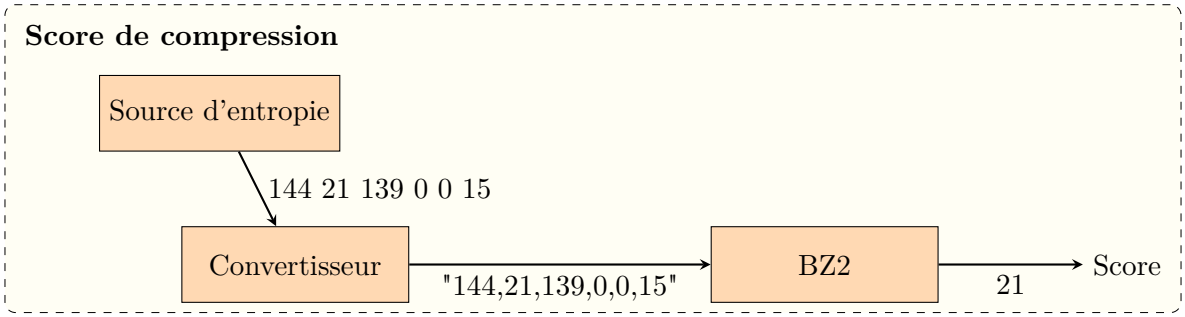


FIGURE 1.3 – Calcul du score de compression

#### 1.4.1.1.7 Scores *Over/Under Runs* (2)

**Définition.**

On définit le ***run test*** comme la série des valeurs montantes ou la série des valeurs décroissantes. Le terme de `run` peut être expliqué comme la successions d'éléments de la même classe. Le nombre d'augmentations ou de réductions définit la longueur du test. Dans un jeu de données aléatoire, la probabilité que la $(i+1)$ème valeur est plus grande ou plus petite que la $i$ème valeur doit suivre la loi binomiale.

Le *run test* est défini tel que :
— $H_0$ : la séquence a été produite de manière aléatoire
— $H_1$ : la séquence n'a pas été produite de manière aléatoire
— Test statistique :

$$Z = \frac{R - \overline{R}}{s_R}$$

où :
— $R$ : le nombre de *runs* observés
— $\overline{R}$, la moyenne (le nombre de runs attendus) telle que :

$$\overline{R} = \frac{2N_+N_-}{N} + 1$$

— $s_R$ la variance (déviation standard) calculée telle que :

$$s_R^2 = \frac{2N_+N_-(2N_+N_- - N)}{N^2(N-1)} = \frac{(\overline{R}-1)(\overline{R}-2)}{N-1}$$

— Niveau significatif : $\alpha$
— Zone critique : Le *run test* rejette l'hypothèse nulle $H_0$ si $|Z| > Z_{1-\alpha/2}$

2. cf. `www.bzip.org`

**Exemple de Run test.**

Soit la suite binaire $X = (x_1, x_2, ..., x_n)$ sortie d'un générateur aléatoire. Pour tous les $i$ allant de 1 à $n-1$. On stocke dans un vecteur de taille $n-1$ le signe de $x_{i+1} - x_i$. Le nombre de run consiste au nombre de changements de signe.

Si $X = 111001111000110$, alors son vecteur associé sera $(+ + - + + + + + - + + + + -)$, soit 6 *runs*. Dans le cas des suites binaires, il s'agit simplement de compter le nombre de changement d'éléments (ou classes), l'ensemble de données étant $\{0, 1\}$. Si l'élément $i$ est équivalent au précédent, alors on passe au suivant, sinon on incrémente de un le nombre de *runs*.

Ici on a donc :

- $R = 6$
- $\overline{R} = \frac{2*3*3}{15} + 1 = 2.2$
- $s_R^2 = \frac{1.2*0.2}{14} = 0,017142857$
- D'où $Z = \frac{6-2.2}{0,130930734} = 29.0229794$
- On choisit un niveau $\alpha = 0.05$
- On cherche $Z_{1-\alpha/2}$. $\frac{1-\alpha}{2} = 0.475$ soit suivant la courbe normale une valeur de $Z_{1-\alpha/2} = 1.96$. Ceci correspond à la valeur critique. La zone de rejet de $H_0$ est telle que $|Z| > 1.96$
- Étant donné que $|Z| > 1.96$, on en conclu que les données ne sont pas aléatoires au risque de 5%.

**Calcul des scores *Over/Under Runs*.**

Pour chaque sous-ensemble, on calcule la médiane des données. On identifie ensuite les données en 3 sous-ensembles :

- Soit elles sont égales à la médiane
- Soit elles sont inférieures à la médiane
- Soit elles sont supérieures à la médiane

Les sous ensembles inférieurs et supérieurs à la médiane sont susceptible d'avoir un *run score* relativement faible, si les données sont suffisamment bien IID.

Les deux scores de *over/under runs* sont calculés comme suit :

1. On récupère la médiane de notre sous ensemble de données. Pour des données binaires, la médiane sera de 0.5
2. Pour chaque sous ensemble original et chaque sous ensemble permuté, un sous ensemble temporaire est construit comme suit. Pour chaque élément :
   (a) Si l'élément est plus grand que la médiane, on ajoute `+1` au vecteur temporaire
   (b) Si l'élément est plus petit que la médiane, on ajoute `-1` au vecteur temporaire
   (c) Si l'élément est égal à la médiane, on passe à l'élément suivant
3. La plus grande taille du *run* sur les `+1` ou les `-1` est considéré comme le premier score
4. La taille du *run* sur les `+1` et les `-1` est considéré comme le second score.

**Exemple.**

Considérant l'ensemble de taille 7 ayant les données suivantes : $\{5, 15, 12, 1, 13, 9, 4\}$.

1. On récupère la médiane : 9

2. Création du vecteur temporaire : $5 < 9 \Rightarrow -1$.... Vecteur final : $\{-1, +1, +1, -1, +1, -1\}$. On note que pour ce vecteur la valeur 9 a été omise.
3. Le *run* le plus long pour le `+1` et le `-1` est de taille 2, (score 1)
4. Le *run* global est de taille 5, (score 2)

#### 1.4.1.1.8 Score excursion

Le score d'excursion mesure la déviation en chaque point d'une somme d'éléments suivant leur moyenne.

**Définition.**
Étant donné les éléments $(s_0, s_1, ..., s_i)$, et leur moyenne $\mu_i$, **l'excursion** $Esc_i$ est définie telle que :

$$Esc_i = s_0 + s_1 + ... + s_i - i * \mu$$

Le score retourné est l'excursion maximale en valeur absolue du sous ensemble de données.

**Calcul du score.**
Le score est ainsi calculé comme suit. $\mu$ est défini comme la moyenne des valeurs d'un ensemble donné.

1. Pour $j = 1$ à $\lfloor \frac{N}{10} \rfloor$ (taille des données du sous ensemble), on calcule $d_j = |Esc_j| = |\sum_{i=0}^{j} s_i|$
2. Score $= \underset{j=0, j<=\lfloor \frac{N}{10} \rfloor}{\text{Max}} (d_j)$

**Exemples.**
Étant donné le sous ensemble suivant : $\{2, 15, 4, 10, 9\}$.

1. On calcule $\mu$, ici $\mu = 8$
2. Calcul des $d_j$ :
   $d_1 = |2 - 8| = 6$, $d_2 = |2 + 15 - 2 * 8|$, $d_3 = 3$, $d_4 = 1$, $d_5 = 0$
3. Le score est donc de 6

#### 1.4.1.1.9 Scores Runs directionnel (3)

Le principe du run test reste le même que celui évoqué précédement à l'exception suivante près : lorsqu'il y a égalité entre deux éléments consécutifs, on le référence comme étant 0 dans le vecteur temporaire plutôt que de passer directement à l'élément suivant.

**Calcul des scores pour des données quelconques.**
Le score est calculé de la façon suivante :

1. Le nombre total de runs (sans considérer le 0 comme un changement de "signe") (score 1)
2. La longueur du plus long run (les 0 sont également ignorés) (score 2)
3. Le maximum entre le nombre de données `+1` et le nombre de données `-1` (score 3)

**Exemple :**
Étant donné l'ensemble suivant : $\{2, 2, 2, 5, 7, 7, 9, 3, 1, 4, 4\}$.

1. On calcule son vecteur temporaire lié $\{0, 0, +1, +1, 0, +1, -1, -1, +1, 0\}$
2. Nous comptons 3 séries de run : $(0, 0, +1, +1, 0, +1)$, $(-1, -1)$ et $(+1, 0)$ (score 1)
3. Le run le plus long est de 4 : $(+1, +1, 0, +1)$ (score 2)
4. On compte 4 `+1` et 4 `-1`, le maximum des deux est donc de 4 (score 3).

**Données binaires.**

Dans le cadre des données binaires en sortie de source d'entropie, il convient de faire un pré-traitement sur les données :

1. On convertit les bits en bytes.
2. On calcule le poids de hamming pour ces éléments :
   Pour $i = 0$ à $\frac{\lfloor \frac{N}{10} \rfloor}{8} - 1$, on stocke dans $W_i$ son poids de Hamming, tel que $W_i = hamming\_weight(s_i, ..., s_{i+7})$
3. On réitère l'opération des données quelconques à partir de ce vecteur $W_i$.

**Poids de Hamming.**

Le **poids de hamming** d'un sous ensemble $(s_i, ..., s_{i+n})$ , donné par $hamming\_weight(s_i, ..., s_{i+n})$ est défini comme le nombre de 1 de la suite $(s_i, ..., s_{i+n})$.

#### 1.4.1.1.10 Score de covariance

Le score de covariance permet de détecter la relation entre les valeurs numériques successives. En effet, toute relation linéaire entre des paires successives va affecter directement ce score. On pourra alors constater une différence entre la covariance calculée sur le sous ensemble de données originales et la covariances calculée sur le sous ensemble de données permutées. La covariance est calculée entre chaque paire consécutive du sous ensemble S tel que $s_i$ est apparié avec $s_{i+1}$.

**Calcul du score.**

Le score est calculé comme suit :

1. la variable *count* est initialisée à 0.
2. $\mu$, moyenne des données de $s_0$ à $s_{\lfloor \frac{N}{10} \rfloor - 1}$ est calculée.
3. Pour $i = 1$ à $\lfloor \frac{N}{10} \rfloor$ on incrémente *count* tel que :
$$count = count + (s_i - \mu)(s_{i-1} - \mu)$$
4. On obtient enfin le score : $Score = \lfloor \frac{count}{\lfloor \frac{N}{10} \rfloor - 1} \rfloor$

**Exemple.**

Étant donné le sous-ensemble : $\{15, 2, 6, 10; 12\}$ de 5 éléments :

1. $\mu = 9$
2. — Pour $i = 1$, $count = 0 + (2 - 9)(15 - 9) = -42$
   — Pour $i = 2$, $count = -42 + (6 - 9)(2 - 9) = -21$
   — Pour $i = 3$, $count = -24$
   — Pour $i = 4$, $count = -21$
3. $Score = \lfloor \frac{-21}{4} \rfloor = -5$

#### 1.4.1.1.11 Score de collision (3)

Une façon naturelle de tester l'entropie d'un générateur est de mesurer le nombre d'essais nécessaires pour obtenir une valeur identique à la première, en d'autres termes, le nombre d'essais nécessaires pour obtenir une collision. Ainsi, notre score de collision va mesurer le nombre d'essais successifs jusqu'à ce qu'un élément identique soit trouvé.

Dans le cas des données binaires, un sous ensemble de données binaires est converti en séquence de 8bits avant d'être testé.

**Calcul des scores.**

Les 3 scores de collisions sont calculés comme suit :

1. *Counts* est une liste d'échantillons nécessaires pour trouver une collision. La liste est vide à l'initialisation.
2. $pos = 0$
3. Tant que $pos < \lfloor \frac{N}{10} \rfloor$
   (a) Trouver le plus petit $j$ tel que $s_{pos} ... s_{pos+j}$
      i. Si aucun $j$ de ce type n'existe, on sort de la boucle tant que
   (b) Ajouter $j$ à la liste *Counts*
   (c) $pos = pos + j + 1$
4. On retourne les scores suivants :
   (a) La valeur minimale de la liste *Counts* (score 1)
   (b) La moyenne de la liste *Counts* (score 2)
   (c) La valeur maximale de la liste *Counts* (score 3)

**Exemple.**

Considérant les données : $\{2, 1, 1, 2, 0, 1, 0, 1, 1, 2\}$ de taille 10.

1. On exécute le contenu de la boucle tant que :
   (a) Considérant 2 comme la 0e valeur, la première collision apparaît à $j = 2$, pour la valeur `1`.
   (b) On passe à une analyse sur le reste de l'ensemble initial non analysé : $\{2, 0, 1, 0, 1, 1, 2\}$. La première collision de ce sous-ensemble apparaît en $j = 3$, pour la valeur `0`.
   (c) On travaille à présent sur l'ensemble $\{1, 1, 2\}$. La première collision apparaît en $j = 1$
   (d) Enfin, on ne trouve aucune collision sur l'ensemble $\{2\}$
2. On retourne les scores :
   (a) $min(Counts) = 1$ (score 1)
   (b) $\mu_{Counts} = 2$ (score 2)
   (c) $Max(Counts) = 3$ (score 3)

### 1.4.1.2 Test statistique spécifique : $\chi^2$

Dès lors que source d'entropie est considérée comme IID, alors la distribution de ces valeurs peut être considérée comme une distribution indépendante, une distribution multinomiale.

**Définition.**

La **loi multinomiale** est une généralisation de la loi binomiale. On considère $m$ résultats possibles (2 pour la loi binomiale). Soit $N_i$ pour $i \in \{1, ..., m\}$ la variable aléatoire multinomiale ayant une probabilité $p_i$, telle que :

$$\sum_{i=1}^{m} N_i = 1 \text{ et } \sum_{i=1}^{m} p_i = 1$$

La densité de probabilité de cette loi s'écrit :

$$\mathbb{P}(N_1 = n_1, ..., N_m = n_m) = \frac{n!}{n_1!...n_m!} p_1^{n_1}...p_m^{n_m}$$

Pour respectivement les espérances et variances suivantes :

$$\mathbb{E}[N_i] = np_i \text{ et } \mathbb{V}[N_i] = np_i(1 - p_i)$$

**Approximation.**

Si les variables sont indépendantes, $\sum_{i=1}^{m} \frac{(N_i - np_i)^2}{np_i(1-p_i)}$ suit une loi du $\chi^2$à $m$ degrés de libertés. Ainsi, le test du $\chi^2$peut ici être utilisé pour tester si des données suivent une distribution multinomiale.

Plus généralement, le test peut être utilisé pour vérifier si des données suivent une distribution particulière, pourvu que celles-ci ne soient pas de trop grande taille. Deux types de tests sont proposés sur la source d'entropie :

- Le test d'indépendance entre des données successives obtenues (tests différents pour des données binaires et non binaires)
- Le test d'ajustement sur les 10 sous-ensembles de données, qui permet de vérifier si le modèle adopté pour les données est satisfaisant, ie. dans quelle mesure les résidus sont dus au hasard (tests différents pour des données binaires et non binaires).
  Nous ne détaillerons pas dans ce rapport ces tests respectifs. Ils sont toutefois décrits dans le document du NIST SP800-90b.

### 1.4.2 Déterminer l'entropie minimale des sources IID

On souhaite à présent estimer l'entropie fournie par une source IID (données indépendantes et identiquement distribuées). Ce test est basé sur le nombre d'observations d'un échantillon le plus courant de la source de bruit ($pmax$). Le but est de calculer la borne minimale d'entropie de la source.

**Calcul de l'entropie minimale.**

Étant donné N échantillons $\{x_1, ...x_n\}$

1. Trouver la valeur la plus souvent rencontrée dans le jeu de données
2. Compter le nombre d'occurrences de cette valeur, stocké dans $C_{MAX}$
3. Calculer $pmax = \frac{C_{MAX}}{N}$
4. Calculer $C_{BOUND} = C_{MAX} + 2.3\sqrt{N * pmax(1 - pmax)}$
5. Calculer $H = -log_2(\frac{C_{BOUND}}{N})$
6. Calculer le nombre d'éléments dans l'échantillon, que l'on stocke dans W.
7. $min(W, H)$ est l'entropie minimale

**Exemple.**

Considérant le jeu de données $\{0, 1, 1, 2, 0, 1, 2, 2, 0, 1, 0, 1, 1, 0, 2, 2, 1, 0, 2, 1\}$ de 20 données :

- La valeur la plus courant du jeu de données est `1` (On compte 6 `0`, 8 `1` et 6 `2`)
- $C_{MAX} = 8$
- $pmax = \frac{8}{20} = 0.4$
- $C_{BOUND} = 8 + 2.3\sqrt{20 * 0.4 * 0.6} = 13.04$
- $H = -log_2(\frac{13.04}{20}) = 0.186$
- $W = 3$
- Entropie minimale calculée : $min(3, 0.186) = 0.186$

# Chapitre 2

# Génération des clés

## 2.1 Définitions et contexte

## 2.2 Audits

### 2.2.1 Audit 1 : X

#### 2.2.1.1 Norme visée

#### 2.2.1.2 Faille

#### 2.2.1.3 Implémentation

**Version OpenSSL.**

**Fonction.**

La fonction liée à cette norme est accessible sous le paquetage `bla/bla/bla`, dont les composantes principales sont listées en *listing* 3.1.

```
#include <stdio.h>
#define N 10
/* Block
 * comment */

int main()
{
    int i;

    // Line comment.
    puts("Hello world!");

    for (i = 0; i < N; i++)
    {
        puts("LaTeX is also great for programmers!");
    }

    return 0;
}
```

||

Listing 2.1 – codeAleatoire.c

**Audit.**

## 2.3 Recommandations générales

FIGURE 2.1 – Titre de figure 2.1

# Chapitre 3

# Poignée de main

## 3.1 Définitions et contexte

## 3.2 Audits

### 3.2.1 Audit 1 : X

#### 3.2.1.1 Norme visée

#### 3.2.1.2 Faille

#### 3.2.1.3 Implémentation

**Version OpenSSL.**

**Fonction.**

La fonction liée à cette norme est accessible sous le paquetage `bla/bla/bla`, dont les composantes principales sont listées en *listing* 3.1.

```
#include <stdio.h>
#define N 10
/* Block
 * comment */

int main()
{
    int i;

    // Line comment.
    puts("Hello world!");

    for (i = 0; i < N; i++)
    {
        puts("LaTeX is also great for programmers!");
    }

    return 0;
}
```

||

Listing 3.1 – codeAleatoire.c

**Audit.**

## 3.3 Recommandations générales

FIGURE 3.1 – Titre de figure 3.1

# Conclusion

Ceci est la conclusion

# Bibliographie


[1] Random number generation. https://wiki.archlinux.org/index.php/Random_Number_Generation.

[2] Miros manual : arandom, prandom, random, srandom, urandom, wrandom. https://www.mirbsd.org/htman/i386/man4/arandom.htm, Octobre 2013.

[3] 3rd, D. E., Schiller, J., and Crocker, S. Randomness Requirements for Security. RFC 4086 (Best Current Practice), June 2005.

[4] Amaury, and MICHEL, B. Debian : Découverte d'une faille de sécurité critique dans openssl de debian. http://linuxfr.org/news/d%C3%A9couverte-dune-faille-de-s%C3%A9curit%C3%A9-critique-dans-openssl-de-deb, Mai 2008.

[5] BOBLET, M., and WONG, E. Openssl prng is not (really) fork-safe. http://emboss.github.io/blog/2013/08/21/openssl-prng-is-not-really-fork-safe/, Août 2013.

[6] Debian-Security. Bulletin d'alerte debian : Générateur de nombres aléatoires prévisibles. https://www.debian.org/security/2008/dsa-1571.

[7] Debian-Security. dowkd.pl. http://security.debian.org/project/extra/dowkd/dowkd.pl.gz.

[8] Debian-Security. openssl-blacklist.deb. https://packages.debian.org/fr/sid/openssl-blacklist.

[9] DELLAQUILA, K., ILAND, D., and LOHR, A. Fms attack on rc4. http://www.cs.rit.edu/~axl6334/crypto/Report.pdf, Juillet 2009.

[10] Fluhrer, S., Mantin, I., and Shamir, A. Weaknesses in the key scheduling algorithm of rc4. In *PROCEEDINGS OF THE 4TH ANNUAL WORKSHOP ON SELECTED AREAS OF CRYPTOGRAPHY* (2001), pp. 1–24.

[11] GUNDERSON, S. H. Some maths. http://plog.sesse.net/blog/tech/2008-05-14-17-21_some_maths.html, Mai 2008.

[12] KLYUBIN, A. Some securerandom thoughts. http://android-developers.blogspot.de/2013/08/some-securerandom-thoughts.html, Août 2013.

[13] MICHAELIS, K., MEYER, C., and SCHWENK, J. Randomly failed! the state of randomness in current java implementations. http://www.nds.rub.de/media/nds/veroeffentlichungen/2013/03/25/paper_2.pdf, 2013.

[14] MUELLER, S. Entropy generator with 100 kb/s throughput. http://lkml.iu.edu//hypermail/linux/kernel/1302.1/00479.html, Février 2013.

[15] WARNER, B. Egd : The entropy gathering daemon. http://egd.sourceforge.net/, Juillet 2002.