Jean Goubault-Larrecq and
Julien Olivain

Detecting Subverted Cryptographic Protocols by Entropy Checking

Research Report LSV-06-13

June 2006

Laboratoire Spécification et Vérification

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

Detecting Subverted Cryptographic Protocols by Entropy Checking

Julien Olivain Jean Goubault-Larrecq
LSV/UMR CNRS & ENS Cachan, INRIA Futurs projet SECSI
61 avenue du président-Wilson, F-94235 Cachan Cedex
$\{$*olivain,goubault*$\}$*@lsv.ens-cachan.fr*

Abstract

What happens when your implementation of SSL or some other cryptographic protocol is subverted through a buffer overflow attack? You have been hacked, yes. Unfortunately, you may be unaware of it: because normal traffic is encrypted, most IDSs cannot monitor it. We propose a simple, yet efficient technique to detect most of such attacks, by computing the entropy of the flow and comparing it against known thresholds.

1 Introduction

Intrusion detection is an important theme in practical computer security. The purpose of this paper is to give a means of detecting some specific attacks targeted at implementations of security protocols such as SSL [6, 16] or SSH [38, 39], in general *cryptographic protocols*.

One might think that nothing differentiates such attacks from the majority of attacks that modern intrusion detection systems (IDS) have to detect, even complex ones [25, 30, 9] (most of which based on buffer overflows). E.g., misuse detection systems can detect these attacks by monitoring flows of (system, or network) events; anomaly detection systems may detect these by, say, noting statistical deviations from normal flows. However, these classical approaches both rest on the assumption that events can be *read* at all by the IDS.

The point in attacks such as [16] or [39] is that traffic is *encrypted*, so that, under normal working conditions, the IDS cannot read it.

The purpose of this paper is to describe a technique that detects such attacks. Our technique is simple, efficient, detects all attacks of this kind without having to write several intrusion profiles or signatures, and does not require key escrows [18, Section 13.8.3] of any form.

To make it brief, our technique is based on an entropy estimator. Define the entropy of an N-character word w over an alphabet $\Sigma = \{0, 1, \ldots, m-1\}$ as $\hat{H}(w) = -\sum_{i=0}^{m-1} f_i \log f_i$, where f_i is the frequency of occurrence of letter i in w, and we take $\log$ to denote logarithms base 2. This concept, due to C. E. Shannon [28], conveys the amount of information stored in w, and is central to physics, coding theory, and statistics [5]. Call *byte entropy* the entropy of words over the byte alphabet (i.e., $n = 256$).

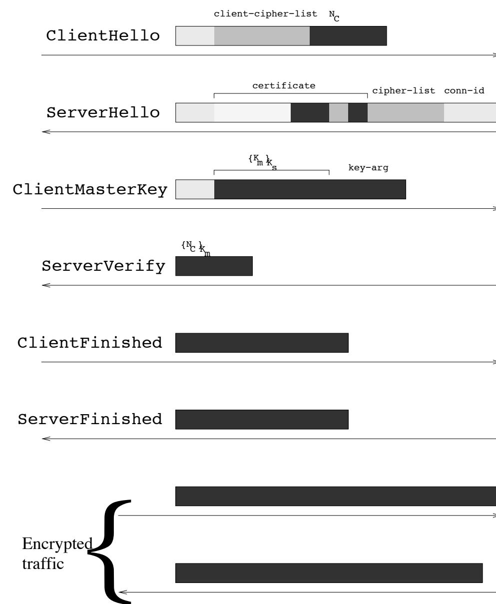

Figure 1: Normal SSL v2 Session

Encrypted traffic is (using state-of-the-art cryptographic algorithms) indistinguishable from random traffic. The byte entropy of a random sequence of characters is 8 bits per byte, at least in the limit $N \to +\infty$. On the other hand, the byte entropy of a non-encrypted sequence of characters is much lower. According to [5, Section 6.4], the byte entropy of English text is no greater than 2.8, and even 0-order approximations do not exceed 4.26.

Let us convey the idea of using byte entropy to detect attacks. For example, the `mod_ssl` attack [16] uses a heap overflow vulnerability during the key exchange (handshake) phase of SSL v2 to execute arbitrary code on the target machine. A normal (simplified) execution of this protocol is tentatively pictured in Figure 1. Flow direction is pictured by arrows, from left (client) to right (server) or conversely. The order of messages is from top to bottom. The handshake phase consists of the top six messages. Encrypted traffic then follows. We have given an indication of the relative level of entropy by levels of shading, from light (clear text, low entropy) to dark gray (encrypted traffic, random numbers, high entropy).

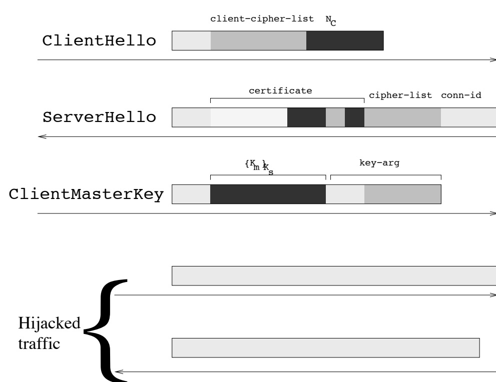

Figure 2: Hijacked SSL v2 Session

Shell codes that are generally used with the `mod_ssl` attack hijack one session, and reuse the https connection to offer basic terminal facilities to the remote attacker. We detect this by realizing that the byte entropy of the flow on this connection, which should quickly approach 8, remains low. See Figure 2 for an illustration of what this should look like. Note that, since the shell code communicates in clear after the key exchange phase, entropy will be low in the bottom messages. (The last three messages of the handshake have been skipped by the shell code. Note that, since they are encrypted, there is no way to distinguish them from post-handshake encrypted traffic.) In fact, since the shell code itself, whose entropy is low, is sent in lieu of a session key, the entropy is already low in some parts of the key exchange. This can be used to detect the attack even if the shell code does not communicate over the https channel, which is also common.

Just the same method applies to detect the SSH CRC32 attack [39], or more recent attacks that *subvert* traffic that ought to be encrypted, or random, or compressed under normal conditions of use. (See examples in Section 7.3)

Outline. We start by reviewing related work in Section 2. This will be an opportunity for us to review alternate detection mechanisms, and to state the known limitations of each approach (including ours). We then introduce the notion of sample entropy and its properties in Section 3. We show in Section 4 how the sample entropy can be evaluated, and used to give reliable estimators of whether a given piece of traffic is scrambled (encrypted, compressed, random) or not. In particular, we shall see that the sample entropy is capable of estimating this on very short bursts of characters with high confidence. We briefly review other possible estimators of disorder in Section 5. In Section 6, we examine how this can be put to use in detecting attacks, where only parts of the protocol messages are meant to be scrambled, and may be subverted—as in Figure 2. We describe how we implemented this in the Net-Entropy sensor, see Section 7. Finally, we conclude in Section 8.

2 Related Work

Entropy and sample entropy have been standard notions in statistics since the pioneering work of C. E. Shannon [28]. They have been used in countless works. We shall review related work on sample entropy and entropy estimators, as needed, in the next section, where we review basic notions and some required mathematical results.

In security, the idea of using the sample entropy to collect some statistical information about a network has already been used, e.g., in [10]. However, our purpose is different. We do not attempt to detect information about a network (e.g., detecting what a user types from timing delays between keystrokes over SSH [29]), rather we wish to detect typical attacker behavior. The entropy of data has already been used as heuristic in virus and malware detection: most of common binary executable files have an average entropy around 6.5 (depending of the compiler, processor architecture, operating system, binary file format). Malicious software executables are usually packed, compressed and/or encrypted. This operation increases the entropy of files. An entropy analysis phase is included in the *PEiD* tool [12]. Data entropy has also been included into file system forensic analysis tools such as *WinHex Forensic* [37], which need to guess the type of files on given file systems (low entropy files are text, XML, mails, binary files; high entropy files are multimedia, compressed, encrypted files). Our focus is different, and the idea of detecting subverted cryptographic protocols, i.e., instances where scrambled flow is expected but clear text is found, is new. We shall see that some of the problems that crop up in this setting require new solutions, e.g., see Section 6.

There is also abundant literature on formal verifica-

tion of cryptographic protocols (see e.g. [11] for an entry point). These works are concerned with verifying certain properties such as secrecy, or authentication on idealized models of communication. Our point here is to analyze actual network flows. In particular, we consider distributions of characters in these flows, which is out of reach of the aforementioned methods.

Next, let us investigate what other methods are available to detect subverted cryptographic protocols, and in general how attacks such as [16] or [39] can be detected today.

Snort [26] detects these attacks by comparing all flows against signatures for known shellcodes and traffic they generate. This is reasonable, because shellcodes will appear in clear during the attack phase, and because only a few dozen standard shellcodes are routinely used in current exploits. (This is very similar to virus detection.) It therefore suffices to list a few characteristic byte sequences, corresponding to each particular shellcode. One problem with this approach is that the signature base has to be maintained, and enriched each time a new shellcode or attack appears on the hacking scene.

Our approach, on the contrary, applies independently of the actual shellcode used, and applies to zero-day attacks. This is also the case for PAYL [33], where worms are detected by evaluating a so-called Manhattan distance between reference one-character distributions and observed traffic. Other possible distances include the Mahanobis distance [34]. One could also argue for the *Kullback-Leibler distance* [5], which in general lends itself to more rigorous mathematical argumentation. Our use of the sample entropy can be seen as a special case of this distance, where the reference distribution is uniform.

Naturally, there is no silver bullet, and every detection technique can be countered. To counter ours, it would suffice for an attacker to use a relatively small shellcode that encrypts its own communication, or even does not communicate at all on the monitored ports, and whose binary code is itself scrambled (i.e., encrypted, or compressed, namely whose binary code achieves a high entropy). Technology to scramble binary code can be taken from the world of encrypted, polymorphic, or metamorphic viruses [31]. It is therefore possible to defeat our mechanism—at least in principle; we shall see in Section 4.2 that our mechanism is so sensitive that it is in fact able to tell polymorphic shellcodes apart from encrypted traffic.

Another countermeasure against cryptographic protocol subversion is to check that all key exchanges are properly formatted. E.g., in the `mod_ssl` attack [16], the `ClientMasterKey` message (3rd message in the handshake) will hold a `key_arg` field of the wrong size. If the intruder detection system is able to monitor each protocol, and recompute all sizes on the fly, this would be a way of detecting these attacks. This is rather complex, and we shall argue against it in Section 6.1. Computing entropies is also much simpler.

3 Sample Entropy and Estimators

First, a note on notation. Recall that we take $\log$ to denote base 2 logarithms. Entropies will be computed using $\log$, and will be measured in *bits*. The notation $\ln$ is reserved to natural logarithms. Some papers we shall refer to use natural logarithms. We shall then adapt their results without mentioning it explicitly; this will usually involve introducing a factor $1/\ln 2 = \log e \sim 1.4427$.

Let w be a word of length N, over an alphabet $\Sigma = \{0, 1, \ldots, m-1\}$. We may count the number n_i of occurrences of each letter $i \in \Sigma$. The *frequency* f_i of i in w is then n_i/N. The *sample entropy* of w is:

$$\hat{H}_N^{MLE}(w) = -\sum_{i=0}^{m-1} f_i \log f_i$$

(The superscript MLE is for *maximum likelihood estimator*.) If w is a random word over Σ, where each character is drawn uniformly and independently, the frequencies f_i will tend to $1/m$ in probability as N tends to infinity, by the law of large numbers.

The formula is close enough to the notion of entropy of a random source, but the two should not be confused. Given a probability distribution $p = (p_i)_{i \in \Sigma}$ over Σ, the *entropy* of p is

$$H(p) = -\sum_{i=0}^{m-1} p_i \log p_i$$

In the case where each character is drawn uniformly and independently, $p_i = 1/m$ for every i, and $H(p) = \log m$.

It is hard not to confuse H and $\hat{H}_N^{MLE}$, in particular because a property known as the asymptotic equipartition property (AEP, [5, Chapter 3]) states that, indeed, $\hat{H}_N^{MLE}(w)$ converges in probability to $H(p)$ when the length N of w tends to $+\infty$, as soon as each character of w is drawn independently according to the distribution p. In the case of the uniform distribution, this means that $\hat{H}_N^{MLE}(w)$ tends to $\log m$. When characters are bytes, $m = 256$, so $\hat{H}_N^{MLE}(w)$ tends to 8 (bits per byte).

Before we continue, note that the approximation $\hat{H}_N^{MLE}(w) \sim H(p)$ is valid when $N \gg m$.

However, we are not interested in the limit of $\hat{H}_N^{MLE}(w)$ when N tends to infinity. There are several reasons for this. First, actual messages we have to monitor may be of bounded length. E.g., the encrypted payload may be only a few dozen bytes long ($N \leq 100$, whereas $m = 256$) in short-lived SSH connections. Second, even though we may count on N being large for

some encrypted connections, we have to decide *at some time point* whether traffic is scrambled or not: we use a cutoff, typically $N \leq 2^{16} = 65\ 536$, and will only compute the entropy of the first 65 536 bytes of traffic. Third, it is important to detect intrusions at the *earliest* time possible. If we can detect unscrambled traffic after just a few dozen bytes, we should emit an alert, and take countermeasures, right away.

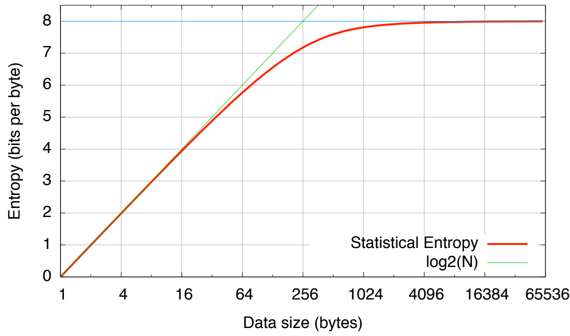

Figure 3: Average Sample Entropy $\hat{H}_N^{MLE}(w)$ of Words w of Size N

Let us plot the average $\hat{H}_N^{MLE}$ of $\hat{H}_N^{MLE}(w)$ when w is drawn uniformly among words of size N, for $m = 256$, and N ranging from 1 to 65 536 (Figure 3). Please note that the x-axis has logarithmic, not linear scale. ($\hat{H}_N^{MLE}$ was evaluated by sampling over words generated using the `/dev/urandom` source.) The value of $H(p)$ is shown as the horizontal line $y = 8$. As theory predicts, when $N \gg m$, typically when N is of the order of roughly at least 10 times as large as m, then $\hat{H}_N^{MLE} \sim H(p)$. On the other hand, when N is small (roughly at least 10 times as small as m), then $\hat{H}_N^{MLE} \sim \log N$. Considering the orders of magnitude of N and m cited above, clearly we are interested in the regions where $N \sim m$... precisely where $\hat{H}_N^{MLE}$ is far from $H(p)$.

At this point, let us ask ourselves what the state of the art is in this domain. The field of research most connected to this work is called *entropy estimation* [2], and the fact that $N \sim m$ or $N < m$ is often characterized as the fact that the probability p is *undersampled*. In classical statistics, our problem is often described as follows. Take objects that can be classified into m *bins* (our bins are just bytes) according to some probability distribution p. Now take N *samples*, and try to decide whether the entropy of p is $\log m$ (or, in general, the entropy of a given, fixed probability distribution) just by looking at the samples. The papers [1, 23, 24] are particularly relevant to our work, since they attempt to achieve this precisely when the probability is undersampled, as in our case.

The problem that Paninski tries to solve [23, 24] is finding an *estimator* $\hat{H}_N$ of $H(p)$, that is, a statistical quantity, computed over randomly generated N-character words, which gives some information about the value of $H(p)$. Particularly interesting estimators are the *unbiased estimators*, that is those such that $E(\hat{H}_N) = H(p)$, where E denotes mathematical expectation (i.e., the average of all $\hat{H}_N(w)$ over all N-character words w).

If we have an unbiased estimator $\hat{H}_N$ of $H(p)$, then our detection problem is easy: compute $\hat{H}_N$ over the N-character input word w, then if $\hat{H}_N(w) = 8$ up to some small tolerance $\epsilon > 0$, then w is random enough (hence almost certainly scrambled), otherwise w is unscrambled. Because we reason up to ϵ, we may even tolerate a small bias; the *bias* of $\hat{H}_N$ is $E(\hat{H}_N) - H(p)$.

The sample entropy $\hat{H}_N^{MLE}$, introduced above, is an estimator, sometimes called the *plug-in estimate*, or *maximum likelihood estimator* [23]. As Figure 3 demonstrates, it is biased, and the bias can in fact be rather large. So $\hat{H}_N^{MLE}$ does not fit our requirements for $\hat{H}_N$.

Let us say right away that, while the *limit* $N \to +\infty$ of the estimator $\hat{H}_N^{MLE}$ *is* unbiased, a surprising result due to Antos and Kontoyiannis is that the error between an estimator of $H(p)$ and $H(p)$ converges to 0 arbitrarily slowly, when p is arbitrary: see [1, Theorem 4], or [23, Section 3.2]. In a sense, this means that finding an unbiased estimator of the actual entropy of the flow is difficult. However, our problem is less demanding: we only want to distinguish this actual entropy from the entropy of the uniform distribution $\mathcal{U}$.

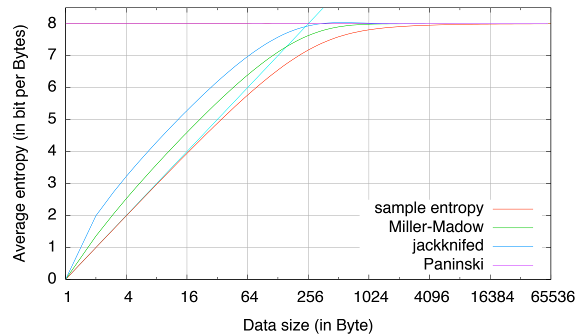

Figure 4: Sample Entropy Estimators

Comparing $\hat{H}_N^{MLE}$ to the entropy at the limit, $\log m$, is wrong, because $\hat{H}_N^{MLE}$ is biased for any fixed N. Nonetheless, we may introduce a correction to the estimator $\hat{H}_N^{MLE}$. To this end, we must estimate the bias. Historically, the first estimation of the bias is the Miller-Madow bias correction [19] $(\hat{m} - 1)/(2N \ln 2)$, where $\hat{m} = |\{i | f_i \neq 0\}|$ is the number of characters that do appear at all in our N-character string w, yielding the

Miller-Madow estimator:

$$\begin{aligned}\hat{H}_N^{MM}(w) &= \hat{H}_N^{MLE}(w) + \frac{\hat{m}-1}{2N\ln 2} \\ &= -\sum_{i=0}^{m-1} f_i \log f_i + \frac{\hat{m}-1}{2N\ln 2}\end{aligned}$$

Another one is the *jackknifed MLE* [8]:

$$\hat{H}_N^{JK}(w) = N\,\hat{H}_N^{MLE}(w) - \frac{N-1}{N}\sum_{j=1}^{N}\hat{H}_N^{MLE}(w_{-j})$$

where w_{-j} denotes the $(N-1)$-character word obtained from w by removing the jth character. While all these corrected estimators indeed correct the $1/N$ term from biases at the limit $N \to +\infty$, they are still far from being unbiased when N is small: see Figure 4.

In the case that interests us here, i.e., when p if the uniform distribution over m characters, an exact asymptotic formula for the bias is known as a function of $c > 0$ when N and m both tend to infinity and N/m tends to c [23, Theorem 3]. (See also Appendix A.) The corrected estimator is:

$$\begin{aligned}\hat{H}_N^{P}(w) &= \hat{H}_N^{MLE}(w) - \log c \\ &\quad + e^{-c}\sum_{j=1}^{+\infty}\frac{c^{j-1}}{(j-1)!}\log j\end{aligned} \tag{1}$$

While the formula is exact only when N and m both grow to infinity, in practice $m = 256$ is large enough for this formula to be relevant. On our experiments, the difference between the average of $\hat{H}_N^P(w)$ over random experiments and $\log m = 8$ is between -0.0002 and 0.0051 for $N \leq 100\ 000$, and tends to 0 as N tends to infinity. (On Figure 4, it is impossible to distinguish $\hat{H}_N^P$—the "Paninski" curve—from 8.) We can in particular estimate that $\hat{H}_N^P$ is a reasonably unbiased estimator of $H(p)$, when p is the uniform distribution.

4 Evaluating the Average Sample Entropy

Instead of trying to compute a correct estimator of the actual entropy $H(p)$, which is, as we have seen, a rather difficult problem, we turn the problem around.

Let $H_N(p)$ be the N*-truncated entropy* of the distribution $p = (p_i)_{i\in\Sigma}$. This is defined as the average of the sample entropy $\hat{H}_N^{MLE}(w)$ over all words w *of length* N, drawn at random according to p. In other words, this is what we plotted in Figure 3. A direct summation shows that

$$H_N(p) = \sum_{\substack{n_0,\ldots,n_{m-1}\in\mathbb{N} \\ n_0+\ldots+n_{m-1}=N}} \left[\binom{N}{n_0,\ldots,n_{m-1}} p_0^{n_0}\ldots p_{m-1}^{n_{m-1}} \times \left(\sum_{i=0}^{m-1} -\frac{n_i}{N}\log\frac{n_i}{N}\right)\right]$$

where

$$\binom{N}{n_0,\ldots,n_{m-1}} = \frac{N!}{n_0!\ldots n_{m-1}!}$$

is the multinomial coefficient.

When p is the uniform distribution $\mathcal{U}$ (where $p_i = 1/m$ for all i), we obtain the formula

$$H_N(\mathcal{U}) = \frac{1}{m^N}\sum_{\substack{n_0,\ldots,n_{m-1}\in\mathbb{N} \\ n_0+\ldots+n_{m-1}=N}} \left[\binom{N}{n_0,\ldots,n_{m-1}} \times \left(\sum_{i=0}^{m-1} -\frac{n_i}{N}\log\frac{n_i}{N}\right)\right] \tag{2}$$

By construction, $\hat{H}_N^{MLE}$ is then an unbiased estimator of H_N. Our strategy to detect unscrambled text is then to take the flow w, of length N, to compute $\hat{H}_N^{MLE}(w)$, and to compare it to $H_N(\mathcal{U})$. If the two quantities are significantly apart, then w is not random. Otherwise, we may assume that w looks random enough so that w is likely to be scrambled.

Not only is this easier to achieve than estimating the actual entropy $H(p)$, we shall see (Section 4.2) that this provides us much narrower confidence intervals, that is, much more precise estimates of non-randomness.

For example, if w is the word

```
0x55 0x89 0xe5 0x83 0xec 0x58 0x83 0xe4
0xf0 0xb8 0x00 0x00 0x00 0x00 0x29 0xc4
0xc7 0x45 0xf4 0x00 0x00 0x00 0x00 0x83
0xec 0x04 0xff 0x35 0x60 0x99 0x04 0x08
```

of length $N = 32$ (which is much less than $m = 256$), then $\hat{H}_N^{MLE}(w) = 3.97641$, while $H_N(\mathcal{U}) = 4.87816$, to 5 decimal places. Since 3.97641 is significantly less than 4.87816 (about 1 bit less information), one is tempted to conclude that w above is *not* scrambled. (This is indeed true: this w is the first 32 bytes of the code of the `main()` function of an ELF executable, compiled under `gcc`. However, we cannot yet conclude, until we compute confidence intervals, see Section 4.2.)

Consider, on the other hand, the word

```
0x85 0x01 0x0e 0x03 0xe9 0x48 0x33 0xdf
0xb8 0xad 0x52 0x64 0x10 0x03 0xfe 0x21
0xb0 0xdd 0x30 0xeb 0x5c 0x1b 0x25 0xe7
0x35 0x4e 0x05 0x11 0xc7 0x24 0x88 0x4a
```

This has sample entropy $\hat{H}_N(w) = 4.93750$. This is close enough to $H_N(\mathcal{U}) = 4.87816$ that we may want to conclude that this w is close to random. And indeed, this w is the first 32 bytes of a text message encrypted with `gpg`. Comparatively, the entropy of the first 32 bytes of the corresponding plaintext is only 3.96814.

Note that, provided a deviation of roughly 1 bit from the predicted value $H_N(\mathcal{U})$ is significant, the $\hat{H}_N^{MLE}$ estimator allows us to detect deviations from random-looking messages extremely quickly: using just 32 characters in the examples above. Actual message sizes in SSL or SSH are of the order of a few kilobytes.

4.1 Computing $H_N(\mathcal{U})$

There are basically three ways to compute $H_N(\mathcal{U})$. (Recall that we need this quantity to compare $\hat{H}_N^{MLE}(w)$ to.) The first is to use Equation (2). However, this quickly becomes unmanageable as m and N grow. Indeed, the outer summation is taken over all m-tuples of integers that sum to N, of which there are exactly $\binom{N+m-1}{m-1}$. For $m = 2$, this would be an easy sum of $N + 1$ terms. For $m = 256$, this would mean summing $O(N^{255})$ terms. E.g., for a 1 kilobyte message ($N = 1\ 024$), this would amount to 2^{635}, or about 10^{191} terms.

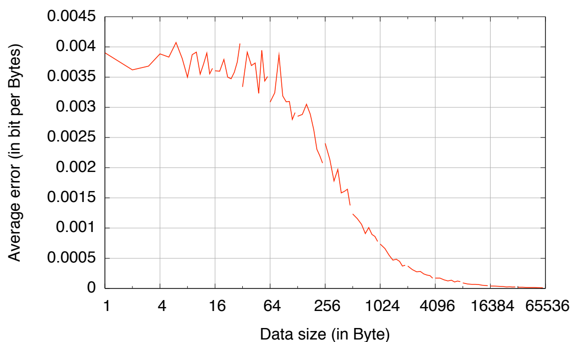

Figure 5: Error Term in (3)

A much better solution is to recall Equation (1). Another way of reading it is to say that, for each constant c, when N and m tend to infinity in such a way that N/m is about c, then

$$H_N(\mathcal{U}) = \log m + \log c - e^{-c} \sum_{j=1}^{+\infty} \frac{c^{j-1}}{(j-1)!} \log j + o(1) \quad (3)$$

As we have seen, when $m = 256$, this approximation should give a good approximation of $H_N(\mathcal{U})$. In fact, this approximation is surprisingly close to the actual value of $H_N(\mathcal{U})$. The $o(1)$ error term is plotted in

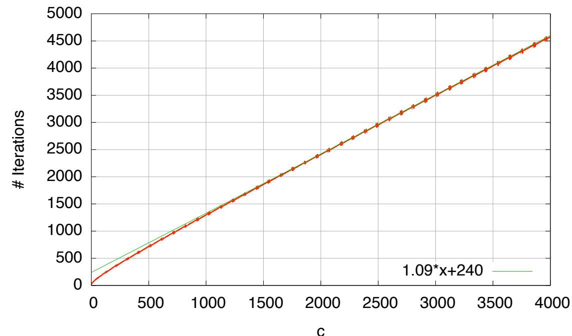

Figure 6: Number of Iterations to Convergence (96-Bit Floats)

Figure 5. It is never more than 0.004 bit, and decreases quickly as N grows.

The above series converges quickly. The series is a sum of positive numbers, so that rounding errors tend not to accumulate. We have implemented this series using 96-bit IEEE floating-point numbers, by summing all terms until the sum stabilizes (i.e., until the next term is negligible compared to the current partial sum, yielding results precise to about 51–63 bits of mantissa). The number of iterations needed is roughly linear in c, see Figure 6. Note that we have gone as far as $c = 4\ 000$, corresponding to $N = 1\ 024\ 000$. This would only be needed for connections lasting for about 1 Mb. In practice, with a cutoff of 64 Kb, we never need more than 415 iterations, see Figure 7.

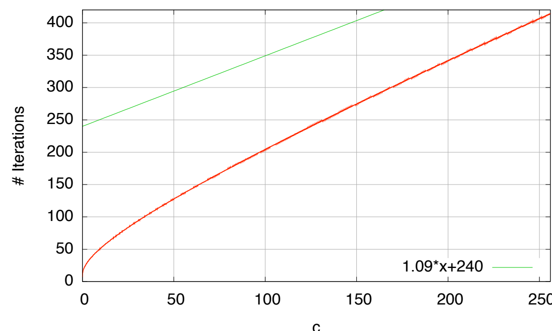

Figure 7: Number of Iterations to Convergence (96-Bit Floats)

On a 1.6 GHz Pentium-M laptop, computing all values of this function from 0.001 to 128 by steps of 0.001 takes 2.65 s., i.e., each computation of the series takes 21μs. on average. (Recall that we compute this series up to roughly 51 bits, i.e., 18 decimal digits, although we only really need 4 or 5 digits.) Depending on the context, this may be fast enough or not. If this is not fast enough, tabulating all the values of the function from $1/256 \sim$

0.004 to 128 by steps of 0.004 requires us to store 384Kb of data, using 96 bit floats, or just 128Kb using 32 bit floats (which is enough for the precision we need).

The third method to evaluate $H_N(\mathcal{U})$ is the standard Monte-Carlo method consisting in drawing enough words w of length N at random, and taking the average of $\hat{H}_N(w)$ over all these words w. This is how we evaluated $H_N(\mathcal{U})$ in Figure 3, and how we defined the reference value of $H_N(\mathcal{U})$ which we compared to (3) in Figure 5. To be precise, we took the average over 100 000 samples for $N < 65\ 536$, taking all values of N below 16, taken one value in 2 below 32, one value in 4 below 64, ..., and one value in 4 096 below 65 536. The spikes are statistical variations that one may attribute to randomness in the source. Note that they are in general smaller than the error term $o(1)$ in (3).

We can then compute $H_N(\mathcal{U})$ by this Monte-Carlo method, and fill in tables with all required values of $H_N(\mathcal{U})$ (note that m is fixed). While this is likely to use only about 128Kb already, we can save some memory by exploiting the fact that $H_N(\mathcal{U})$ is a smooth and increasing [23, Proposition 3] function of c, by only storing a few well-chosen points and extrapolating; and by using the fact that values of c that are not multiples of $4/256$ or even $8/256$ are hardly ever needed.

In the sequel, and in particular in Section 6, we shall need to evaluate more complicated functions, notably functions of which we know no asymptotic approximation such as (3). We shall always compute them by looking up tables, filled in by such Monte-Carlo methods.

4.2 Confidence Intervals

Evaluating $\hat{H}_N^{MLE}(w)$ only gives a statistical indication of how close we are to $H_N(\mathcal{U})$. Recall our first example, where w was the first 32 bytes of the code of the `main()` function of some ELF executable. We found $\hat{H}_N^{MLE}(w) = 3.97641$, while $H_N(\mathcal{U}) = 4.87816$. What is the actual probability that w of length $N = 32$ is unscrambled when $\hat{H}_N^{MLE}(w) = 3.97641$ and $H_N(\mathcal{U}) = 4.87816$?

It is again time to turn to the literature. According to [1, Section 4.1], when N tends to $+\infty$, $\hat{H}_N^{MLE}$ is asymptotically Gaussian, in the sense that $\sqrt{N}\ln 2(\hat{H}_N^{MLE} - H)$ tends to a Gaussian distribution with mean 0 and variance $\sigma_N^2 = Var\{-\log p(X)\}$. In non-degenerate cases (i.e., when $\sigma_N^2 > 0$), the expectation of $(\hat{H}_N^{MLE} - H)^2$ is $\Theta(1/N)$... but precisely, the $p = \mathcal{U}$ case *is* degenerate.

As we have already said, our interest is not in the limit of large values of N. Unfortunately, much less is known about the variance of $\hat{H}_N^{MLE} = \hat{H}_N^{MLE}$ when $N \sim m$ or $N < m$ than about its bias. One useful inequality is that the variance of $\hat{H}_N^{MLE}$ is bounded from above by $\log^2 N/N$ [1, Remark (iv)], but this is extremely conservative.

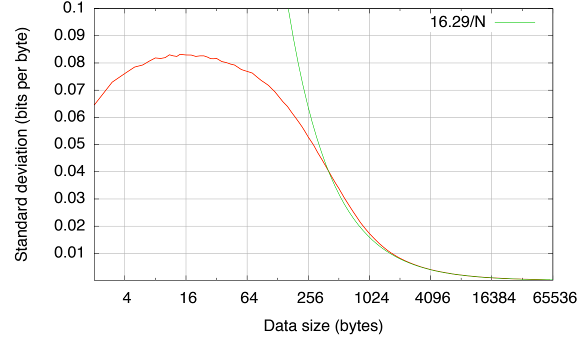

Figure 8: Standard Deviation of $\hat{H}_N^{MLE}(w)$

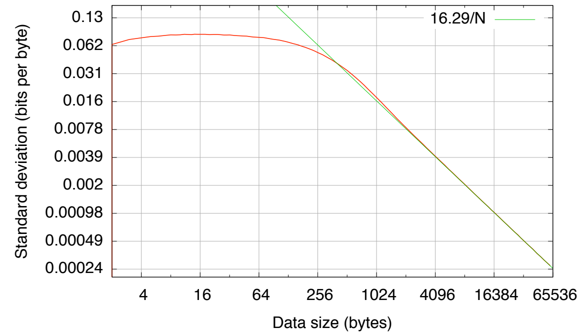

Figure 9: Standard Deviation of $\hat{H}_N^{MLE}(w)$, in Log Scale

On the other hand, we may estimate the standard deviation of $\hat{H}_N$ by a Monte-Carlo method, estimating the statistical standard deviation $SD(\hat{H}_N^{MLE})$ of $\hat{H}_N^{MLE}(w)$, on random words w of length N. The result is shown as the thick curve in Figure 8. It turns out that $SD(\hat{H}_N^{MLE})$ evolves as $16.29/N$ when $N \to +\infty$, as Figure 9 (y-axis in logarithmic scale) makes clear. The coefficient 16.29 here is $\sqrt{\frac{m-1}{2}\frac{1}{\ln 2}}$: as predicted by [20, Equation (12)], the variance of $\hat{H}_N^{MLE}$ evolves as $\sigma_N^2 + \frac{m-1}{2N^2\ln^2 2}$ when $N \to +\infty$, and in our case $\sigma_N^2 = 0$, as we have seen. $SD(\hat{H}_N^{MLE})$ reaches its maximum (about 0.08 bit) for N of the order of 16, while for $N \geq m$ the standard deviation is so close to $16.29/N$ that we can equate the two for all practical purposes.

In particular, for typical packet sizes of 1, 2, 4, and 8 Kb, the standard deviation $SD(\hat{H}_N^{MLE})$ is 0.016, 0.008, 0.004, and 0.002 bit respectively. This is small.

Then, we can also estimate percentiles, again by a Monte-Carlo method, see Figure 10: the y values are given so that a proportion of all words w tested falls within $y \times SD(\hat{H}_N^{MLE})$ of the average value of

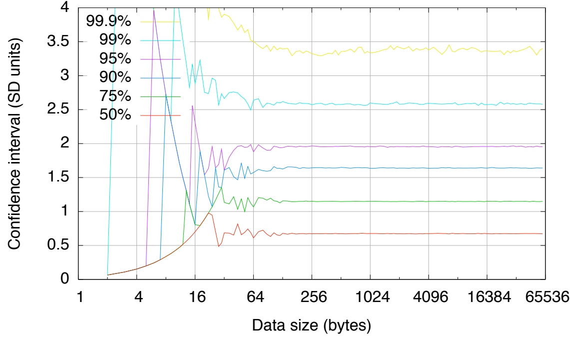

Figure 10: Percentiles

$\hat{H}_N^{MLE}$. The proportions go from 50% (bottom) to 99.9% (top). Note that, unless $N \leq 16$ (which is unrealistic), our estimate of $\hat{H}_N^{MLE}$ is exact with an error of at most $4SD(\hat{H}_N^{MLE})$, with probability 99.9%, and that $4SD(\hat{H}_N)$ is at most $64/N$, and in any case no larger than 0.32 bit (for words of about 16 characters).

Let's return to our introductory question: What is the actual probability that w of length $N = 32$ is unscrambled when $\hat{H}_N^{MLE}(w) = 3.97641$ and $H_N(\mathcal{U}) = 4.87816$? For $N = 32$, $SD(\hat{H}_N^{MLE})$ is about maximal, and equal to 0.081156. So we are at least 99.9% sure that the entropy of a 32-byte word with characters drawn uniformly is $4.87816 \pm 4 \times 0.081156$, i.e., between 4.55353 and 5.20279: if $\hat{H}_N^{MLE}(w) = 3.97641$, we can safely bet that w is *not* scrambled. Note that $N = 32$ is almost the worst possible case we could dream of. Still, $\hat{H}_N^{MLE}$ is already a reliable estimator of randomness here.

For packets of sizes 1, 2, 4, and 8 Kb, and a confidence level of 99.9% again, $\hat{H}_N^{MLE}$ is precise up to ± 0.0625, ± 0.0313, ± 0.0156, and ± 0.0078 bit respectively.

Data source	**Entropy** (bits/byte)	
	$\hat{H}_N^{MLE}$	H_N
Binary executable (elf-i386)	6.35	8.00
Shell scripts	5.54	8.00
Terminal activity	4.98	8.00
1 Gbyte e-mail	6.12	8.00
1Kb X.509 certificate (PEM)	5.81	7.80 ± 0.061
700b X.509 certificate (DER)	6.89	7.70 ± 0.089
130b bind shellcode	5.07	6.56 ± 0.24
38b standard shellcode	4.78	5.10 ± 0.28
73b polymorphic shellcode	5.69	5.92 ± 0.27
Random 1 byte NOPs (i386)	5.71	7.99

Figure 11: Sample Entropy of Some Common Non-Random Sources

We report some practical experiments in Figure 11, on non-cryptographic sources. This gives an idea of the amount of redundancy in common data sources. The entropy of binary executables (ELF format, i386 architecture) was evaluated under Linux and FreeBSD by collecting all `.text` sections of all files in `/bin` and `/usr/bin`. Similarly, the entropy of shell scripts was computed by collecting all shell scripts on the root volume of Linux and FreeBSD machines (detected by the `file` command). Terminal activity was collected by monitoring a dozen `telnet` connections (port 23) on `tcp` from a given machine with various activity, such as text editing, manual reading, program compilation and execution (about 1 Mb of data). As far as e-mail is concerned, the measured entropy corresponds to 3 years of e-mail on the first author's account. These correspond to large volumes of data (large N), so that H_N is 8 to 2 decimal places, and confidence intervals are ridiculously small.

The next experiments were made on smaller pieces of data. Accordingly, we have given H_N in the form $H \pm \delta$, where δ is the 99.9% confidence interval. Note that X.509 certificates are definitely classified as unscrambled. We have also tested a few shellcodes, because, first, as we have seen in Figure 1 and Figure 2, it is interesting to detect when some scrambled piece of data is replaced by a shellcode, and second, because detecting shellcodes this way is challenging. Indeed, shellcodes are typically short, so that H_N is significantly different from 8. More importantly, modern polymorphic and metamorphic virus technologies, adapted to shellcodes, make them look more scrambled. (In fact, the one we use *is* encrypted, except for a very short prolog.) While the first two shellcodes in Figure 11 are correctly classified as unscrambled (even a very short 38 byte non-polymorphic shellcode), the last, polymorphic shellcode is harder to detect. The 99.9% confidence interval for being scrambled is $[5.65, 6.19]$: the sample entropy of the 73 byte polymorphic shellcode is at the left end of this interval. The 99% confidence interval is 5.92 ± 0.19, i.e., $[5.73, 6.11]$: with 99% confidence, this shellcode is correctly classified as non-scrambled. In practice, shellcodes are usually preceded with padding, typically long sequences of the letter `A` or the hexadecimal value `0x90` (the No-OPeration i386 instruction), which makes the entropy decrease drastically, so the examples above are a worst-case scenario. Detecting that the random `key-arg` field of Figure 1 was replaced by a shellcode in Figure 2 is therefore feasible.

Another worst-case scenario in polymorphic viruses and shellcodes is given by mutation, whereby some specific instructions, such as `nop`, are replaced with other instructions with the same effect, at random. This fools pattern-matching detection engines, and also increases entropy. However, as the last line shows on a large

amount of random substitutes for `nop` on the i386 architecture, this makes the sample entropy culminate at a rather low value compared to 8.

We conclude this section by noting that $\hat{H}_N^{MLE}$ is a remarkably precise estimator of $H_N(\mathcal{U})$, even in very undersampled cases.

5 Other Estimators of Randomness

We have chosen to estimate the entropy $H(p)$ of an unknown distribution p to detect whether p is close to the uniform distribution or not. While this works well in practice, there are other ways to evaluate randomness: see [13, Section 3.3], or [21].

For example, we may check that the mean $\sum_{i=0}^{m-1} i f_i$ is close to $(m-1)/2$. (Recall that $f_i = n_i/N$ is the frequency of letter i.) We may check that the χ^2 statistic

$$\hat{V}_N(w) = \sum_{i=0}^{m-1} \frac{(n_i - Np_i)^2}{Np_i} = \sum_{i=0}^{m-1} \frac{(n_i - N/m)^2}{N/m}$$

(when $p_i = 1/m$ is the uniform distribution $\mathcal{U}$) is not too large. Recall [13, Section 3.3.1.C] that, in the limit $N \to +\infty$, the probability that $\hat{V}_N(w) \leq v$, for any $v > 0$, tends to the χ^2 function with $m-1$ degrees of freedom. It is also well-known that, in this case, and letting $\nu = m-1$, then $\hat{V}_N(w)$ is less than $\nu + \sqrt{2\nu}x_\pi + 2/3x_\pi^2 - 2/3 + O(1/\sqrt{\nu})$ with probability π, where $x_\pi = 1.64$ for $\pi = 0.95$, say, and $m > 30$ [13, Section 3.3.1.A]. Let us just note that $\hat{V}_N(w)$ should be of the order of $m-1$ when $p = \mathcal{U}$, and $N \to +\infty$.

One may think of computing all, or at least some of these quantities, to get an improved test of randomness. However, here is an informal argument that suggests that this would be pointless. To simplify things, assume that N is large (recall that the χ^2 approximation is only valid for large N).

If the sample entropy $\hat{H}(w)$ is close to its maximum value $\log m$ (when $p = \mathcal{U}$), it is known that the frequencies f_i are close to $1/m$, say $f_i = 1/m + \delta_i$. Since the derivative of $-x \log x$ is $-\log x - 1/\ln 2$, and its second derivative is $-1/(x \ln 2)$, we may approximate $\hat{H}(w)$ by

$$\begin{aligned}\hat{H}(w) &= \log m + \sum_{i=0}^{m-1} \delta_i (\log m - 1/\ln 2) \\ &\quad - \sum_{i=0}^{m-1} m \frac{\delta_i^2}{2 \ln 2} + o\left(\sum_{i=0}^{m-1} \delta_i^3\right)\end{aligned}$$

The first sum vanishes, since $\sum_{i=0}^{m-1} \delta_i = 0$. So

$$\hat{H}(w) - \log m = -\frac{m}{2 \ln 2} \sum_{i=0}^{m-1} \delta_i^2 + o\left(\sum_{i=0}^{m-1} \delta_i^3\right)$$

Since $n_i - N/m = N\delta_i$, $\hat{V}_N(w) = mN \sum_{i=0}^{m-1} \delta_i^2$, so

$$\hat{H}(w) - \log m = -\frac{1}{2N \ln 2} \hat{V}_N(w) + o\left(\sum_{i=0}^{m-1} \delta_i^3\right)$$

With probability $\pi = 0.95$, $\hat{V}_N(w)$ will be of the order of $\nu = m-1$, and we retrieve a form of the Miller-Madow estimator: $\hat{H}(w) - \log m$ is roughly $-(m-1)/(2N \ln 2)$ when $p = \mathcal{U}$. In other words, a χ^2 test essentially amounts to estimating the Miller-Madow bias correction, and checking that it is small. This is only valid in the limit $N \to +\infty$, and we estimated the bias much more precisely in Section 3 anyway.

We also note that Fu *et al.* [10] compared empirically the sample mean, sample variance, and sample entropy as indicators of randomness, and observed that of the three, sample entropy was the most robust, i.e., the least sensitive to noise.

There are many other statistical tests. One weakness of tests based on sample entropy is that $\hat{H}_N(w)$ depends only on the frequencies f_i. Any permutation of the letters in w would give rise to the same sample entropy. In particular, $\hat{H}_N(w)$ is unable to note the difference between the regular sequence of letters $0, 1, \ldots, m-1, 0, 1, \ldots, m-1, 0, \ldots$, and a truly random one. (To be fair, it *will* detect a difference for small N, where $\hat{H}_N$ will rise more slowly on the regular sequence.) Testing the average, or a χ^2 statistic, suffers from the same problem. Some other tests, such as the serial test or the gap test [13, Section 3.3.2], or Maurer's universal statistical test [14], or compression tests [5, Chapter 5], do not, and are able to detect correlations between letters. The defect that most of them share is that they require large data volumes, i.e., large values of N, to be significant. In fact, variants of the entropy test also do detect correlations between letters, as first shown by Shannon [28]: just compute the entropy of digraphs (pairs of letters at positions i, $i+1$), trigraphs (letters at positions i, $i+1$, $i+2$), for example.

We have chosen not to investigate this. While this indeed improves our estimate of randomness, the simple sample byte entropy $\hat{H}_N(w)$, with $m = 256$, was enough in practice. Furthermore, computing the latter can be done by maintaining an array of 256 values of $-f_i \log f_i$ over each connection. This is relatively straightforward to implement, and uses small computing resources. On the other hand, computing the entropy of digraphs requires up to 65 536 entries per array, which may clog the machine. It is imaginable that computing the entropy of consecutive nibble pairs would offer some advantage here. (A nibble is a half-byte; and no, the entropy of nibble digraphs is not the same as the byte entropy, because of byte-straddling pairs of nibbles).

6 Taking into Account Unscrambled Sections

Until now, we have explored a sweetened situation, where we had one word (a packet, a flow of characters during a connection, say), and we wanted to decide whether it was scrambled or not, as a whole. Referring to our coloring conventions of Figure 1 and Figure 2, this allows us to detect whether observed traffic is all dark gray, or contains enough light gray spots.

However, as we can see on Figure 1—and although colors were exaggerated—, some sections of the observed traffic *have* to be light gray. In typical cryptographic protocols, at least some of the first messages exchanged are in clear, and not random. We should not conclude that an attack is in progress here.

Moreover, some specific protocols insert a more or less important bias. The ideal case of a cryptographic protocol over TCP would exhibit only encrypted traffic. This is what SSH2 does, for example. Other protocols add some control information in clear to encrypted packets. This adds a bias to the entropy estimator, which may or may not be negligible. A case in point is the binary packet format for SSH1 (seen at the TCP level), which starts with the packet length in clear (4 bytes), then contains 1 through 8 padding bytes (usually pseudo-random), and finally the encrypted SSH1 packet. The most significant bytes of the 4-byte packet length field will be zero most of the time. (In practice, TCP packets are no longer than a few kilobytes.) Over a full SSH1 connection spanning several TCP packets, the sample entropy will therefore tend to a value smaller than 8, depending on distribution of packet lengths. In our experience, we routinely observed limiting values of around 7.95: see Figure 12, in particular the two SSH1 curves. One particular concern is that this bias is in effect controlled by the untrusted client user, who may force the SSH1 client to send identical-sized packets: it suffices for the client to send the same message over and over, e.g., using a shell script. The extreme case is that of a client typing an e-mail message, where each typed letter gives rise to a 1-byte packet, followed by an echo 1-byte packet.

The same problem occurs in the TLS protocol: each TLS record begins with a content type identifier (1 byte, `0x17` for application data, `0x15` for alerts, etc.), the version of the protocol (two fixed bytes, `0x0301` in the case of TLS 1.0), the record length (2 bytes), and finally the encrypted payload.

The above examples show that entropy may be lower than expected. In other cases, the entropy can also be higher than expected. A typical example is when counters or sequence numbers are encountered in clear, which re-increases entropy, compared to constant data such as version numbers or type identifiers.

This incurs two problems. First, we should use the $\hat{H}_N^{MLE}$ estimator to detect whether some *specific*, not all, sections of traffic are scrambled (dark gray, high entropy). These specific sections are dependent on the protocol used. If some of these specific sections has low entropy (light gray), an alert is reported. To this end, we may track connections, and compute the sample entropy of various sections of traffic, depending on the port on which communication takes place (e.g., 443 for https). We explore how this can be done for known protocols, such as SSL, in Section 6.1. We shall see that this approach is unrealistic. We describe simpler and more realistic approaches in Section 6.2 and Section 6.3.

The second problem is that the intuition of coloring that we used in Figure 1 and Figure 2 is meaningless, at least formally. There is no such thing as the sample entropy *at a given byte* in the flow. The concept of entropy only makes sense on words (or subwords) of sufficient length. Fortunately, as we have seen in previous sections, we only need to explore rather short subwords to get a reliable enough estimate of scrambledness.

6.1 Known Protocols

For well-known cryptographic protocols such as SSH (port 22), or SSL—including `https` (port 443), `nntps` (563), `ldaps` (636), `telnets` (992), `imaps` (993), `ircs` (994), `pop3s` (995), `smtps` (465)—, it is feasible to document which parts of traffic should have high entropy.

Most if not all of these protocols start with a key exchange, or handshake, phase, usually followed by encrypted traffic. The most critical sections are found in this initial handshake phase. Once keys have been established, there is no practical way to break the protocol. (We do not consider attacks which use the protocol without breaking it, such as the SUN login-over-SSH buffer overflow exploit [7].) It seems therefore particularly important that sample entropies are estimated precisely during the handshake phase.

If all sections are of fixed size, this is easy to do. E.g., on the example of Figure 1, compute $\hat{H}_N^{MLE}(w)$ on the topmost dark subword N_C of the `ClientHello` message (N_C is 16 bytes long, about the worst-case as far as confidence intervals are concerned, but as we have seen, sample entropy can already be estimated accurately in this case); then compute it on the light-and-dark `certificate` subword of the `ServerHello` message when it arrives, and so on. Provided that these subwords are large enough, this will provide us a reliable estimate whether these sections are scrambled or not.

Note that the `certificate` subword is not uniformly light or dark. This is because this is an X.509

certificate, with some data in clear (light), and some encrypted signatures (dark). We should therefore also parse X.509 certificates, and only compare the entropy of those parts of the certificate that contain scrambled data (typically signatures).

In general, however, sections are of varying sizes. For example, in SSL versions 2 and up, key sizes are themselves subject to negociation between client and server during the handshake phase. Some fields are optional: in the `ClientHello` message of Figure 1, an optional session ID field can be inserted before the N_C field in case of session resumption. X.509 certificates may have varying sizes, depending on the number of fields filled in, on the sizes of clear text descriptions of issuers, certificate authorities, etc.

In other words, to implement this solution, we need to implement a full parser for the protocol at hand (e.g., SSL), together with parsers for X.509 certificates and other fields of varying size and structure. This is done anyway in products such as Ethereal [4]. The only simple way we know of doing this is to include standard code that does it. E.g., in the SSL case, we may just import relevant code from the OpenSSL project [22]. But the complete implementation of a new protocol parser may introduce more security flaws than using an already mature library. A vivid illustration of this principle was recently produced when it was discovered that the Snort intrusion detection system [26] was remotely vulnerable through its Back Orifice protocol parser [17].

In general, the approach sketched here, of embedding a full parser for the protocols at hand, has a major drawback: any buffer overflow attack on the original library (e.g., OpenSSL) will be present in the entropy monitoring tool… and will be triggered in exactly the same situations. The need for maintaining code as standards evolve is also a drawback of this approach.

We must therefore conclude that the approach of parsing known protocols completely is unrealistic.

6.2 Aggregating Fields

Instead, we shall *aggregate* fields, or even whole messages. In other words, we shall compute the sample entropy of fields or messages as a whole. For example, the X.509 certificate of the `ServerHello` message of Figure 1 is expected to have an overall entropy of roughly 6.89, according to Figure 11.

Naturally, aggregating fields means that confidence intervals must be made wider. In the current state of the art, it seems that no mathematical theory is available to predict the actual mean and variance of $\hat{H}_N^{MLE}$ on whole fields or messages, similarly to the results of previous sections. We shall however give an idea how $\hat{H}_N^{MLE}$ evolves when we switch from one light to one dark area

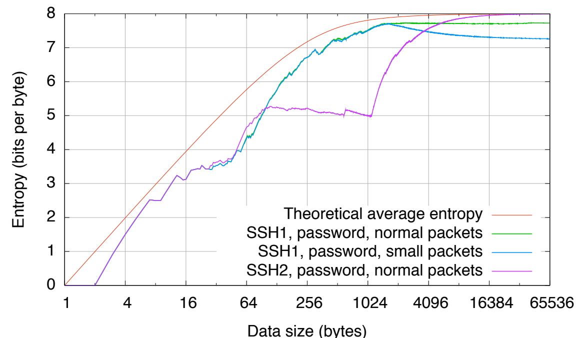

Figure 12: Entropy of SSH traffic

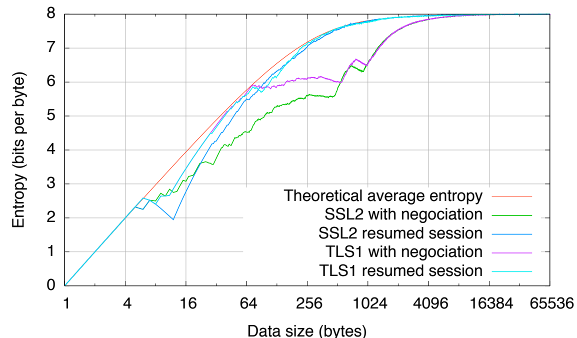

Figure 13: Entropy of SSL traffic

in the next section.

6.3 Bumps

To give an idea how $\hat{H}_N^{MLE}$ evolves when several fields of different entropies are concatenated, we illustrate this on random messages of length N, with a prefix of length N_1 generated using a low entropy source, say of entropy H_1; followed by a high-entropy suffix, say with source entropy equal to H_2. Actual messages naturally exhibit a more complex mix of zones, but our point here is to make clearer what curves should be expected on simple examples.

Asymptotic formulae for this case are not too hard to obtain, but are rather obscure. Making statistical experiments and plotting the resulting curves is far more instructive. We plotted $\hat{H}_N^{MLE}$ for varying values of N from 0 to 65 535, and for varying values of N_1 (32, 64, 256, 1 024, 4 096, and 16 384) in Figure 14. This represents what should be expected from messages with a Base64 encoded prefix of length N_1 followed by a sequence of scrambled bytes.

While we only see the low-entropy prefix, i.e., for the first N_1 bytes, the sample entropy follows the usual curve for $\hat{H}_{N_1}^{MLE}$, which we have already discussed at length, and tends to the constant value H_1 (H_1 being about 6 in

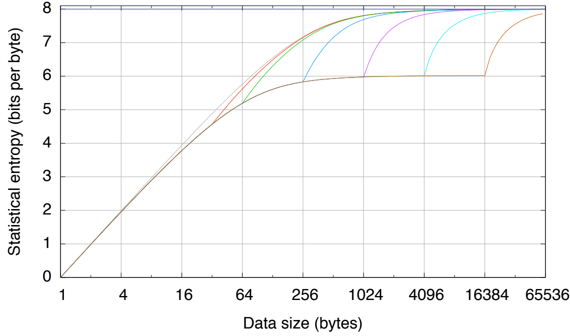

Figure 14: Switching From Random Base64 to Random Characters

our experiment). After the N_1 first characters, we see a sharp increase in entropy, connecting the first curve to a similar-shaped curve whose new limit is H_2 (8 bits in our experiment).

It is only to be expected that any switch from one field to another field of differing entropies will then be observable as a sharp turn in direction, either upwards (going to a higher-entropy field) or downwards (exiting a high-entropy field).

More interesting is the standard deviation obtained this way. This is plotted in Figure 15, together with two reference straight lines. The $16.29/N$ line is the asymptotic standard deviation for the sample entropy of the high-entropy source. The standard deviation for the low entropy source would be aligned on a lower line, of y-value $8.10/N$. (Remember the formula is $\sqrt{\frac{m-1}{2}\frac{1}{\ln 2\ N}}$; for Base64 encoding, $m \sim 6$, yielding $8.10/N$.) This figure reads as follows. While we are looking at the low-entropy prefix, the standard deviation is as low as predicted in earlier sections. Long after we have entered the high-entropy suffix, the standard deviation progressively approaches its asymptote $16.29/N$—a very low value. The new effect here is that the standard deviation jumps to higher values just after we have switched from one field to the next.

This might be detrimental to the quality of the estimation. However, notice that the top of the bump, for each value of N_1, always remains below the $\log e/\sqrt{N}$ line. Remember that, in non-degenerate cases (i.e., when $\sigma_N^2 > 0$), the expectation of $(\hat{H}_N^{MLE} - H)^2$ is $\Theta(1/N)$ (Section 4.2), i.e., that, up to a $1/\ln 2$ factor to make up for our use of base-2 logarithms, the standard deviation in non-degenerate cases is of the order $\Theta(1/\sqrt{N})$. Remember also that the steady-state behavior, i.e., on long fields with the same source entropy, *is* degenerate.

In other words, the standard deviation is low ($\Theta(1/N)$) on long enough sequence of bytes with the same entropy, and goes up to about $\Theta(1/\sqrt{N})$ when we switch from one field to another with different source entropy.

This is bad news in a sense, because one consequence is that confidence intervals will necessarily be larger than in the simple cases studied before Section 6. But observe that the actual bit values are low anyway. On Figure 15, the standard deviation never exceeds 0.15 bits globally (as in previous experiments), and the standard deviation at the highest point in the bump after N_1 is 0.046 bit (reached after 352 bytes, when $N_1 = 256$), resp. 0.024 bit (reached after $1\ 280$ bytes, when $N_1 = 1\ 024$), resp. 0.0116 bit (reached after $5\ 120$ bytes, when $N_1 = 4\ 096$), resp. 0.0056 bit (reached after $20\ 480$ bytes, when $N_1 = 16\ 384$). Remember that 99.9% percentiles were at about 3.5 times the standard deviation around the average. This corresponds to 99.9% confidence intervals of ± 0.161, resp. ± 0.084, resp. ± 0.041, resp. ± 0.020 bit around the average.

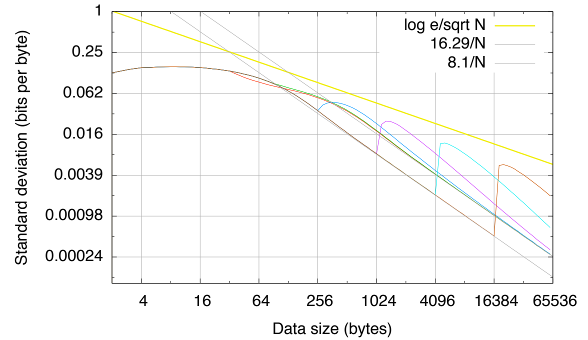

Figure 15: Standard Deviation (Base64 + Random)

We conducted a similar experiment when the low-entropy prefix (the first N_1 bytes) was not a random Base64 file, but rather the first N_1 bytes of a given text—namely the TEX source of the paper [27]. The results are shown in Figure 16 and Figure 17. While the curves are slightly different, we are forced to reach the same conclusions: going from one field to the next forces a sharp jump in the sample entropy, and grows small bumps into the standard deviation curve.

In other words, to detect abnormal values of the sample entropy, confidence intervals should not be an issue. But the expected shapes of entropy curves should show some sharp jumps at some specific positions in the incoming flow. These positions are not entirely known. However, we can classify areas in the plane where the sample entropy is representative of normal behavior. For example, Figure 18 shows $5\ 398$ valid connections, plus one attack, which were observed from real traffic at LSV. Every valid connection falls into a series of rectangular areas. Note that valid connections produce varying curves, depending on protocol version and actual imple-

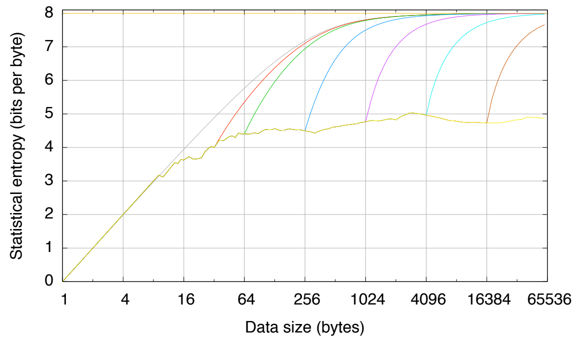

Figure 16: Switching From Given Text to Random Characters

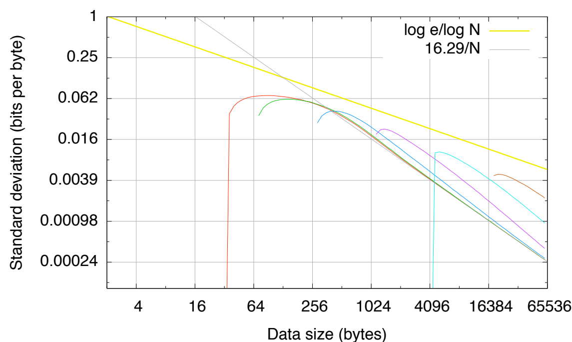

Figure 17: Standard Deviation (Text + Random)

mentation of the protocol. They can all be classified by the displayed sequence of rectangles. Observe also that the unique attack shown here exhibits a sharp difference from the curves, and exits the rectangular areas fast. We only represent the sample entropy of the whole connection, but measured on a per-packet basis. The attack curve exits after the 6th packet (1 939 cumulated bytes), in a clear way.

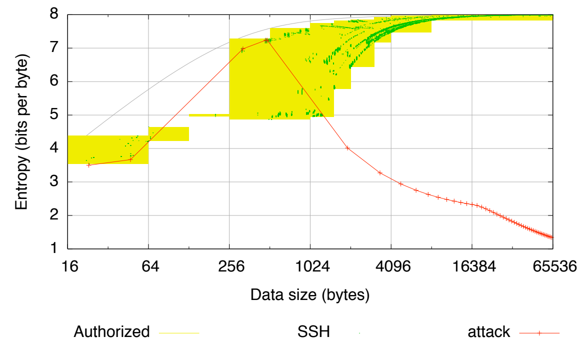

Figure 18: SSH Normal Behavior, and an Attack

7 Implementation in the Net-Entropy Sensor

Net-entropy is a user program (not a kernel module) written in C. We use the *pcap* library [32] to implement low-level network frame capture, and the *LibNIDS* library [35, 36] to implement IP defragmentation and TCP stream reassembly. The latter library is a user-land port of the Linux TCP/IP stack (kernel 2.0.36).

While this seems simple enough, there is a catch. We have observed that, in certain situations, LibNIDS incorrectly classified sniffed packets as invalid. This is *not* a problem with LibNIDS. Instead, this is a side-effect of modern implementations of TCP/IP stacks. E.g., kernels will not compute packet checksums on machines with recent network interface cards, which implement hardware acceleration feature (TCP segmentation offload, hardware Rx/Tx checksumming). In other words, the kernel relies on the network interface card to compute checksums in hardware, and just sets these checksums to zero. But then, any sensor that captures network flow while running on one peer of the connection will get packets from the local loop, i.e., with zero checksums, and these will appear to be invalid. This is totally inconsequential if Net-entropy runs on a router, which is the intended application, since the router is never one of the peers. In case Net-entropy were to be used directly on a workstation, we would need to get around this, typically by adding a special case to the LibNIDS checksum verification routine, to ignore invalid checksums in data coming

```
Port: 22
Direction: both
Cumulative: yes
RangeUnit: bytes
# Range: start end min_ent max_ent
Range:    1    63 0          4.38105154
Range:   64   127 4.22877741 4.64838314
Range:  128   255 4.95194340 5.02499151
Range:  256   511 4.86894369 7.28671360
Range:  512  1023 4.86310673 7.59574795
Range: 1024  1535 4.94409609 7.74570751
Range: 1536  2047 5.77497149 7.81915951
Range: 2048  3071 6.44314718 7.85139179
Range: 3072  4095 7.17234325 7.92034960
Range: 4096  8191 7.46498394 7.96606302
Range: 8192 65536 7.82608652 7.99687433
```

Figure 19: An example range file, for SSH (port 22)

from the local host.

The Net-entropy sensor keeps track of all connections of configured protocols and emits an alarm via the *Unix Syslog* system when sample entropy is out of specified bounds. Net-Entropy was designed to run on routers or intrusion detection hosts, by passively sniffing the network.

For performance reasons, Net-entropy limits the analysis to the first 64 Kb of data in network connections. We also drop connection monitoring after some predefined timeout. Indeed, it may happen that the sensor misses the termination of the connection, for example in cases of network or CPU overload. It is also well-known that limiting monitoring processes is good practice. In particular, this avoids attacks where the malicious attacker starts many simultaneous connections, e.g., testing for passwords, without terminating them properly (with a TCP FIN or RST packet). A timeout is used to get rid of these dead connections. This timeout is set by default to 15 minutes (900 seconds). Every connection that has remained inactive for this long will be removed from monitoring. This behavior makes Net-entropy more resistant to denial-of-service and brute force attacks (SYN flood, SSH scan).

Let us recall that we want to check key exchanges, not user activity. It is extremely unlikely that a key exchange take any longer than a few seconds, even on low bandwidth networks. Note that a delayed key exchange is abnormal, whatever its entropy.

7.1 Configuring Net-Entropy

As already hinted at the end of Section 6.3, Net-entropy checks the sample entropy of monitored connections on specified ranges. Configuring these ranges is done through protocol-specific range files. Figure 19 shows an example of a range file, for the SSH protocol (port 22).

Looking directly at the `Range:` lines, one observes that ranges are specified as rectangular areas. E.g., the first row states that, for SSH connections, the sample entropy between the first byte and byte number 63 should be between 0 and 4.38105154. *Size ranges*, e.g., from byte 1 to byte 63, are specified here in byte units, but can also be specified on a per-packet basis, by replacing the `bytes` keyword in the `RangeUnit:` line by `packets`.

It would be a hassle if the user of Net-entropy had to include all these information by hand. Instead, we provide a supervised learning mode where the user only specifies size ranges, and Net-entropy fills out the least and highest entropy values, from either a live network capture or from a *libpcap* capture file.

Each range file contains a definition for one protocol exactly. As the example above shows, we identify protocols by their server ports, at the TCP destination. Formally, a range file consists of a list of *dirlist* directives, obeying the following grammar:

dirlist:	*dirlist directive* | *directive*
yesno:	"`yes`" | "`no`"
direction:	"`cli2srv`" | "`srv2cli`" | "`both`"
rangeunit:	"`bytes`" | "`packets`"
directive:	"`Port:`" INT
|	"`Direction:`" *direction*
|	"`Cumulative:`" *yesno*
|	"`RangeUnit:`" *rangeunit*
|	"`Range:`" INT INT FLOAT FLOAT

where INT stands for a positive integer (included in $[1, +\infty)$) and FLOAT for a rational number (included in $[0, 8]$). Empty lines and lines beginning with a '`#`' are silently ignored.

A range file *must* contain a port number and at least one range. The `Direction` directive allows one to choose which data to analyze, depending on the direction of the TCP flow. Possible directions are `cli2srv` for analyzing only data from client to server, `srv2cli` for server to client only and `both` for all data. The default value is `both`, in case this directive is omitted.

The default value of the `Cumulative` directive is `yes`, meaning that statistical entropy will be computed on a per-connection basis, periodically checking whether it is in the indicated valid ranges. If the `Cumulative` directive is set to `no`, statistical entropy will be computed for each packet separately, restarting from zero at each packet boundary.

As we said earlier, the `RangeUnit:` directive the unit in which size ranges are measured. It is either `packets` or `bytes` (default).

The `Range:` directive allows one to define a new range. It takes four parameters. The first two denote the size range, the other two are the least and highest allowed entropies for this range. Net-entropy does a few consistency checks, e.g., that entropies are between 0 and

8, that the highest entropy for a range n_1–n_2 is at most $\log n_2$, that size ranges do not overlap, and are sorted in increasing order.

Entropy fields can be left empty, by writing an IEEE 754 NaN (Not a Number) in the corresponding field. This is taken as an indication by the supervised learning mode that the range file should be updated. Using the -l option to Net-entropy will then modify the range file and fill out the required entropy fields. Additionally, using Net-entropy with the -l option will replace entropy alerts by mere update messages.

To avoid congestion, the range file is only updated once a connection is terminated for which some entropy bound has to be modified. In particular, this means that training Net-entropy through an interactive SSH session won't change the range file, until the user quits the session.

As is customary in every anomaly detection technique, it is important to trust the learning set, otherwise the protocol ranges will be wrong or biased. Tools such as Ethereal [4] allow one to select packets from network capture files very precisely, and to edit captures so as to remove offending outliers before sending the training data to Net-entropy with options -l, and -r (offline capture).

The command line options of Net-Entropy are summarized in the following table:

Parameter	Arg	Default value
Interface	-i <if>	First interface
Off-line file	-r <file>	No file
Packet capture filter	-f <filter>	No filter
Runtime user	-u <user>	root
Memory limit	-m <lim>	No limit
Connection data size limit	-t <lim>	65536 bytes
Connection timeout	-T <timeout>	900 seconds
Rule file	-P <file>	No rule
Learning mode	-l	Disabled
Erase old range file	-d	Disabled
Change config file	-c	net-entropy .conf
Statistics files	-s <dir>	No statistics
Flush statistics	-F	Disabled
Per-byte statistics	-b	Per packet

Net-entropy is also configured through the use of a system-wide configuration file, organized as a list *paramlist* of parameters. The configuration file format is defined by the following grammar:

paramlist: *paramlist param* | *param*
param: "Interface:" STRING
| "PcapFilter:" STRING
| "RuntimeUser:" STRING
| "MemoryLimit:" INT
| "MaxTrackSize:" INT
| "ConnectionTimeout:" INT
| "ProtoSpec:" STRING

where INT stands for a positive integer and STRING for a character string. The Interface parameter, as well as the -i option, allows one to choose the network interface used for the capture. If this parameter is omitted, the first available network interface reported by the operating system is used. The PcapFilter parameter (-f option) is the filter used by the *pcap* library. The RuntimeUser parameter (-r) sets the user account name which will be used for dropping root privileges, e.g. the pcap or nobody users, or a dedicated system account. (Net-entropy is meant to start up as root, if only to allow network capture to function properly.) The MemoryLimit parameter (-m) sets the maximum memory limit that the Net-entropy sensor can use. This limit is a security against denial of service (DoS) attacks. The MaxTrackSize parameter (-t) sets the number of bytes after which a connection will cease to be monitored. The ConnectionTimeout parameter (-T) is the delay after which connections will be assumed to be inactive, and their monitoring will cease. The ProtoSpec parameter (-P) adds range files for additional protocols to be monitored.

The performance of the Net-entropy sensor can be significantly enhanced when *BSD Packet Filters* [15] are used in the *libpcap*. The point is that protocols which we are not interested in will simply be ignored at the kernel level. For example, it is meaningful to monitor SSH connections coming from outside a local area network (LAN), so as to detect intrusions on the LAN. It is less interesting to monitor outgoing SSH connections. Another case where BSD packet filters are useful is when Net-entropy is deployed on a router for some LAN, but we know that only a few machines on the LAN implement a given protocol (e.g., https will only run on secured servers on the LAN). In this case, BSD packet filters can be used to focus on packets on given ports *and* given addresses or sub-networks. This can be done with a filter of the form:

(dst net *LAN* and
 (dst port $port_1$ or ... or dst port $port_n$)) or
(src net *LAN* and
 (src port $port_1$ or ... or src port $port_n$))

where *LAN* is the address of the network to protect, and $port_n$ are TCP ports of protocols to protect. This basic filter template works on a common network. More complex filters should be used depending of the network structure to protect.

7.2 Net-Entropy Alert Messages

Net-entropy emits different kinds of alerts. We illustrate this on Figure 20, which exhibits a fictitious connection that generates all kinds of alerts. Alerts can be emitted during a connection:

- **Entropy alarm start**: entropy is below the minimum or above the maximum for the corresponding

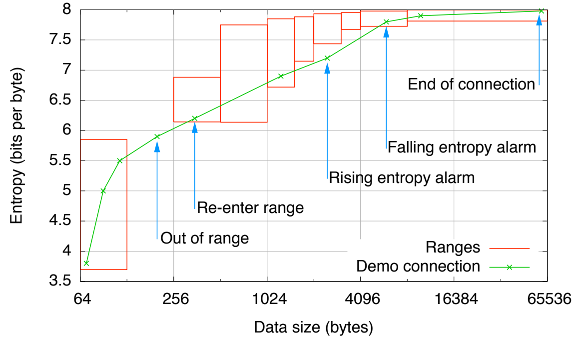

Figure 20: Net-Entropy alert messages

size range.

- **Entropy alarm stop**: after an entropy alarm start, entropy is within bounds again.
- **Out of size range**: no size range corresponds to connection data. This means that the range file has gaps. This may be intentional, then the out of range alert signals that a given connection has an unexpected size: this is then an actual alert.
- **Reenter size range**: after an out of size range alert, connection size is back to some value inside some size range.

Some other alerts are emitted when a connection ceases to be monitored, either because it is terminated, or because some size limit or some timeout was reached. This can only happen if at least one alert was emitted during the connection.

- **End of connection**: the TCP connection has ended. The reason of the disconnection is given: '*connection closed*' means that one of the peers has closed the connection normally, using a TCP FIN packet, while '*connection reset*' signals that the connection was terminated forcefully, with an RST packet.
- **Maximum data size reached**: the connection data size limit was exceeded.
- **Connection timed out**: the connection was inactive for a duration larger than the allowed timeout.

7.3 Attack Examples

Let us demonstrate two attacks. This will illustrate how Net-entropy detects them. The two attacks we have chosen are well-known and have been frequently used by hackers. The first one is the `mod_ssl` attack [16]. This attack example has been made with a vulnerable server running a Linux RedHat 7.3 operating system, Apache 1.3.23 web server, mod_ssl 2.8.7 and OpenSSL 0.9.6b (the three later software are the version bundled with the RedHat distribution). This exploits a bug in the SSL module of the Apache web server. On Figure 21, we draw valid `https` connections that have been learned by Net-entropy as green dots. The light green zone corresponds to valid ranges. The red curve corresponds to a run of the `mod_ssl` attack. The grey curve in the background is the theoretical average entropy $\hat{H}_N^{MLE}$. With this set of valid connections, the attack is detected at the second packet, but is still very close to normal packets at this point. The third and next upcoming packets are farther and farther from normal behavior. At the end of the `mod_ssl` attack, the connection entropy is 5.7 bits per byte for 16.75 kilobytes of data, which is very far from the expected 8.0.

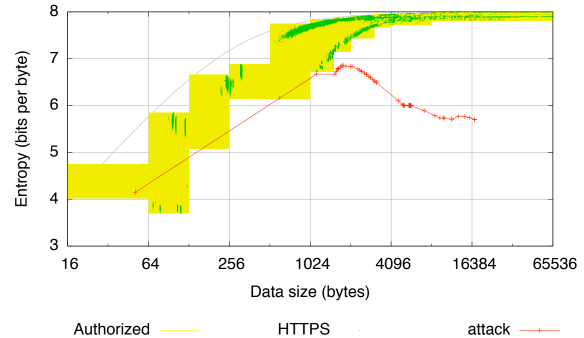

Figure 21: Net-Entropy Limits with `mod_ssl` Attack

The second demonstration attack [39] exploits a bug in SSH servers. The SSH1 CRC32 attack has also commonly been used by hackers. This attack example has been tested on the same computer of the previous attack: a Linux Redhat 7.3 with SSH 1.2.20 secure shell server. With the configuration shown in light green in Figure 18, this attack is detected at the 6th packet (1 939 bytes seen). The first 5 packets are similar to normal connection packets for SSH1 with password-based authentication.

7.4 Benchmarks

Net-Entropy was tested for a duration of one month on a test network, without any malicious network traffic, and for a duration of six months on a front router of the local area network of the *Laboratoire Spécification Vérification* laboratory.

The computers used for tests were based on Pentium 4 processors running at 2.4 GHz, with 1 Gb of RAM and 4 100 Mbps Ethernet interfaces.

On the test network, a normal SSH connection and a normal SSL connection were started every 6 minutes. At

this rate, 480 connections were started each hour, 11 520 each day, 345 600 connections for the 30 days of the test. This was meant to reflect a realistic activity rate in a purely attack-free scenario. Under these conditions, Net-entropy only uses 4.5 Mb of RAM and 33.90 seconds of CPU time over the whole month. The average CPU usage is very low: 0.0013%. The CPU time requirement is in the order of seconds per month.

On the other hand, and as should be expected, the real network is more hostile: more undesirable traffic is seen. In six months, we have seen a total of 563 964 cryptographic network connections, broken down as 511 164 `ssh` connections, 39 970 `https` connections, 8 411 `pop3s` connections, and 4 419 `imaps` connections. Among the 511 164 `ssh` connections, only 32 135 were valid connections—only 6.3% of all `ssh` traffic. The other 479 029 were the result of about 135 scans. Other scans, typically high-speed scans, were automatically blocked by other utilities running on routers at the provider's site, and were not taken into account in the experiment. Under these conditions, Net-entropy used 58 minutes and 23 seconds and 16.8 Mb of memory over the 6 month period. The average CPU usage was 0.022%. This increase in CPU usage, compared to the test network, is due to the fact that scanning packets exhibit burst traffic. This entails that more parallel connections must be reassembled by the LibNIDS library. Still, we observe that CPU and memory usage remain low.

8 Conclusion

In this paper, we have shown how to implement the simple idea of sample entropy checking to detect subverted cryptographic flows. As we have argued in the introduction, this has practical import: the OpenSSL [16] and SSH CRC32 [39] exploits are examples of this kind of attack, where traffic that should be encrypted just isn't. Our technique also adapts to detecting traffic that looks scrambled but shouldn't (e.g., polymorphic viruses). Such attacks are hard to detect by conventional techniques, which rely on virus signatures for example. We have shown that, in the simple case of one data field that should be random-looking, sample entropy was a remarkably precise estimator of randomness. We have also described how computing entropy profiles could be adapted in more complex, realistic situations. Practical experiments show that this technique is also extremely efficient; remember that over a six-month period in a real network environment, the average CPU usage was 0.022%.

Net-Entropy is currently a userland process. A kernel version would be in order—but the lack of generic hardware support for floating-point operations in current Linux kernels makes this harder than expected. Another profitable extension would be for Net-Entropy to be able to monitor traffic at the IP level, and to include support for IPv6, IPSec and ESP; LibNIDS currently only allows us to plug Net-Entropy onto TCP/IPv4 traffic—which already accounts for a good deal of connections. It would also be sensible to use Net-Entropy as a firewall module: the sensor could directly execute firewall actions, like logging, rejecting or redirecting.

9 Acknowledgments

Thanks to Mathieu Baudet, Elie Bursztein and Stéphane Boucheron for judicious advice. This work was partially supported by the RNTL Project DICO, and the ACI jeunes chercheurs "Sécurité informatique, protocoles cryptographiques et détection d'intrusions".

10 Availability

Net-Entropy is a free open source project. It is available under the CeCILL2 license [3]. The project homepage can be found at `http://www.lsv.ens-cachan.fr/net-entropy/`.

References

[1] ANTOS, A., AND KONTOYIANNIS, I. Convergence properties of functional estimates for discrete distributions. *Random Structures and Algorithms 19* (2001), 163–193.

[2] BIALEK, W., AND NEMENMAN, I., Eds. *Estimation of Entropy and Information of Undersampled Probability Distributions—Theory, Algorithms, and Applications to the Neural Code* (Whistler, BC, Canada, Dec. 2003), `http://www.menem.com/~ilya/pages/NIPS03/`. Satellite of the Neural Information Processing Systems Conference (NIPS'03).

[3] CEA, CNRS, AND INRIA. Cecill free software license agreement, 2005. `http://www.cecill.info/licences/Licence_CeCILL_V2-en.html`.

[4] COMBS, G. Ethereal. `http://www.ethereal.com/`.

[5] COVER, T. M., AND THOMAS, J. A. *Elements of Information Theory*. John Wiley & sons, 1991.

[6] DIERKS, T., AND ALLEN, C. Request for Comments 2246 – The TLS Protocol Version 1.0, January 1999.

[7] DOWD, M. Multiple vendor system V derived 'login' buffer overflow vulnerability. `http://www.securityfocus.com/bid/3681`, Dec. 2004. BugTraq Id 3681.

[8] EFRON, B., AND STEIN, C. The jackknife estimate of variance. *Annals of Statistics 9* (1981), 586–596.

[9] ESSER, S. CVS malformed entry modified and unchanged flag insertion heap overflow vulnerability. `http://www.securityfocus.org/bid/10384`, May 2004. BugTraq Id 10384.

[10] FU, X., GRAHAM, B., BETTATI, R., AND ZHAO, W. Active traffic analysis attacks and countermeasures. In *Proc. 2nd IEEE Int. Conf. Computer Networks and Mobile Computing* (2003), pp. 31–39.

[11] GOUBAULT-LARRECQ, J. Special issue on models and methods for cryptographic protocol verification. *Journal of Telecommunications and Information Technology 4* (2002).

[12] JIBZ, QWERTON, SNAKER, AND XINEOHP. PEiD. http://peid.has.it/.

[13] KNUTH, D. E. *The Art of Computer Programming*, 2nd ed., vol. 2. Addison-Wesley, 1981.

[14] MAURER, U. A universal statistical test for random bit generators. *Journal of Cryptology 5*, 2 (1992), 89–105.

[15] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of the Winter 1993 USENIX Conference* (San Diego, CA, USA, Jan. 1993), pp. 259–270.

[16] MCDONALD, J. OpenSSL SSLv2 malformed client key remote buffer overflow vulnerability. http://www.securityfocus.com/bid/5363, May 2003. BugTraq Id 5363.

[17] MEHTA, N. Snort back orifice preprocessor remote stack buffer overflow vulnerability. http://www.securityfocus.com/bid/15131, Oct. 2005. BugTraq Id 15131.

[18] MENEZES, A. J., VANSTONE, S. A., AND OORSCHOT, P. C. V. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[19] MILLER, G. A. Note on the bias of information estimates. In *Information Theory in Psychology; Problems and Methods II-B* (Glencoe, IL, USA, 1955), H. Quastler, Ed., Free Press, pp. 95–100.

[20] MODDEMEIJER, R. The distribution of entropy estimators based on maximum mean log-likelihood. In *Proc. 21st Symp. Information Theory in the Benelux* (Wassenaar, the Netherlands, May 2000), J. Biemond, Ed., pp. 231–238.

[21] NIST. NIST/SEMATECH e-handbook of statistical methods. http://www.itl.nist.gov/div898/handbook/.

[22] The OpenSSL Project. http://www.openssl.org/.

[23] PANINSKI, L. Estimation of entropy and mutual information. *Neural Computation 15* (2003), 1191–1253.

[24] PANINSKI, L. Estimating entropy on m bins given fewer than m samples. *IEEE Transactions on Information Theory 50*, 9 (Sept. 2004), 2200–2203.

[25] PURCZYŃSKI, W. Linux kernel privileged process hijacking vulnerability. http://www.securityfocus.com/bid/7112, Mar. 2003. BugTraq Id 7112.

[26] ROESCH, M. Snort: Lightweight intrusion detection for networks. In *13th Systems Administration Conference (LISA'99)* (1999), USENIX Associations, pp. 229–238.

[27] ROGER, M., AND GOUBAULT-LARRECQ, J. Log auditing through model checking. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW'01), Cape Breton, Nova Scotia, Canada, June 2001* (2001), IEEE Comp. Soc. Press, pp. 220–236.

[28] SHANNON, C. E. A mathematical theory of communication. *Bell System Technical Journal 27* (1948), 379–423.

[29] SONG, D. X., WAGNER, D., AND TIAN, X. Timing analysis of keystrokes and timing attacks on SSH. In *10th USENIX Security Symposium* (Washington, D.C., USA, Aug. 2001).

[30] STARZETZ, P. Linux kernel do_brk function boundary condition vulnerability. http://www.securityfocus.com/bid/9138, Dec. 2003. BugTraq Id 9138.

[31] Understanding and managing polymorphic viruses. Enterprise Papers, 1996. Symantec Corporation, http://securityresponse.symantec.com/avcenter/reference/striker.pdf.

[32] TCPDUMP TEAM. Libpcap, May 1994. http://www.tcpdump.org/.

[33] WANG, K., CRETU, G., AND STOLFO, S. J. Anomalous payload-based worm detection and signature generation. In *RAID'2005* (2005), Springer-Verlag LNCS 3858, pp. 227–246.

[34] WANG, K., AND STOLFO, S. J. Anomalous payload-based network intrusion detection. In *RAID'2004* (2004), Springer-Verlag LNCS 3224, pp. 203–222.

[35] WOJTCZUK, R. Libnids, 1999. http://libnids.sourceforge.net/.

[36] WOJTCZUK, R. Libnids: The infallible e-component of network intrusion detection systems. Master's thesis, Warsaw University, 1999.

[37] X WAYS SOFTWARE TECHNOLOGY AG. Winhex forensic. http://www.x-ways.net/forensics/.

[38] YLONEN, T., AND LONVICK, C. RFC 4253 – SSH transport layer protocol, Mar. 2005.

[39] ZALEWSKI, M. SSH CRC-32 compensation attack detector vulnerability. http://www.securityfocus.com/bid/2347, Feb. 2001. BugTraq Id 2347.

A Deriving the Paninski Formula

Let us give an informal argument for Equation (1). Recall that:

$$H_N(p) = \sum_{\substack{n_0,\ldots,n_{m-1}\in\mathbb{N}\\ n_0+\ldots+n_{m-1}=N}} \left[\binom{N}{n_0,\ldots,n_{m-1}} p_0^{n_0}\ldots p_{m-1}^{n_{m-1}} \times \left(\sum_{i=0}^{m-1} -\frac{n_i}{N}\log\frac{n_i}{N} \right) \right]$$

where

$$\binom{N}{n_0,\ldots,n_{m-1}} = \frac{N!}{n_0!\ldots n_{m-1}!}$$

is the multinomial coefficient.

Recall the multinomial expansion formula:

$$\sum_{\substack{n_0,\ldots,n_{m-1}\in\mathbb{N}\\ n_0+\ldots+n_{m-1}=N}} \left[\binom{N}{n_0,\ldots,n_{m-1}} x_0^{n_0}\ldots x_{m-1}^{n_{m-1}} \times \right] = (x_0+\ldots+x_{m-1})^N$$

On the other hand, recall that binomial distributions can be approximated by Poisson distributions. When $N \to +\infty$ and $p \to 0$ with $pN \sim \lambda$,

$$\frac{N!}{n!(N-n)!} p^n (1-p)^{N-n} \sim e^{-\lambda}\frac{\lambda^n}{n!} \qquad (4)$$

This can be derived from Stirling's formula and elementary calculation.

For convenience, we shall use the hat notation $n_0,\ldots,\hat{n}_i,\ldots,n_{m-1}$ to denote the list $n_0,\ldots,n_{m-1}$

without the entry numbered i. Also, $n_0 + \ldots + \hat{n}_i + \ldots + n_{m-1}$ denotes the sum of $n_0, \ldots, \hat{n}_i, \ldots, n_{m-1}$, and similarly for products.

$$\begin{aligned}
H_N(p) &= \sum_{\substack{n_0,\ldots,n_{m-1}\in\mathbb{N}\\ n_0+\ldots+n_{m-1}=N}} \left[\binom{N}{n_0,\ldots,n_{m-1}} p_0^{n_0}\ldots p_{m-1}^{n_{m-1}} \times \left(\sum_{i=0}^{m-1} -\frac{n_i}{N}\log\frac{n_i}{N} \right) \right] \\
&= \log N + \frac{1}{N} \sum_{\substack{n_0,\ldots,n_{m-1}\in\mathbb{N}\\ n_0+\ldots+n_{m-1}=N}} \left[\binom{N}{n_0,\ldots,n_{m-1}} \times p_0^{n_0}\ldots p_{m-1}^{n_{m-1}} \sum_{i=0}^{m-1} -n_i \log n_i \right] \\
&= \log N + \frac{1}{N} \sum_{i=0}^{m-1} \sum_{n_i=0}^{N} -n_i \log n_i \times \frac{N!}{n_i!(N-n_i)!} p_i^{n_i} \times f(p_i, n_i)
\end{aligned}$$

where

$$\begin{aligned}
f(p_i, n_i) &= \sum_{\substack{n_0,\ldots,\hat{n}_i,\ldots,n_{m-1}\in\mathbb{N}\\ n_0+\ldots+\hat{n}_i+\ldots+n_{m-1}=N-n_i}} \left[\binom{N-n_i}{n_0,\ldots,\hat{n}_i,\ldots,n_{m-1}} \times p_0^{n_0}\ldots\widehat{p_i^{n_i}}\ldots p_{m-1}^{n_{m-1}} \right] \\
&= (1-p_i)^{N-n_i}
\end{aligned}$$

by using the multinomial expansion formula. Since moreover $-n_i \log n_i$ is 0 when n_i equals 0, we may sum over n_i from 1 to N, hence $H_N(p)$ equals

$$\log N - \frac{1}{N} \sum_{i=0}^{m-1} \sum_{n_i=1}^{N} n_i \log n_i \frac{N!}{n_i!(N-n_i)!} p_i^{n_i} (1-p_i)^{N-n_i}$$

Assume $N \to +\infty$, $m \to +\infty$, and for each i, $0 \leq i \leq m-1$, $p_i \to 0$ with $p_i N \sim \lambda_i$; in particular, $\sum_{i=0}^{m-1} \lambda_i \sim N$. Then by (4),

$$\begin{aligned}
H_N(p) &\sim \log N + \frac{1}{N} \sum_{i=0}^{m-1} \sum_{n_i=1}^{N} -n_i \log n_i \, e^{-\lambda_i} \frac{\lambda_i^{n_i}}{n_i!} \\
&= \log N + \frac{1}{N} \sum_{n=1}^{N} -n \log n \frac{\sum_{i=0}^{m-1} e^{-\lambda_i} \lambda_i^n}{n!} \\
&= \log N - \frac{1}{N} \sum_{n=1}^{N} \log n \frac{\sum_{i=0}^{m-1} e^{-\lambda_i} \lambda_i^n}{(n-1)!}
\end{aligned}$$

In the particular case where p is the uniform distribution $\mathcal{U}$, $N, m \to +\infty$ and $N/m \sim c$, then $p_i = 1/m$ and $\lambda_i = c$ for every i, so

$$\begin{aligned}
H_N(\mathcal{U}) &\sim \log N - \sum_{n=1}^{N} \log n \frac{e^{-c} c^{n-1}}{(n-1)!} \\
&\sim \log m + \log c - e^{-c} \sum_{n=1}^{+\infty} \log n \frac{c^{n-1}}{(n-1)!}
\end{aligned}$$

and we conclude since $\log m$ is the value $H(\mathcal{U})$ of the source entropy.