# Secure Communication: a New Application for Active Networks

Manuel Günter, Marc Brogle and Torsten Braun

IAM-00-007

July 2000

.

## Abstract

SplitPath is a new application for the easy, well-known and provably secure One-time pad encryption scheme. Two problems hinder the One-time pad scheme from being applied in the area of secure data communication: the random generation and the distribution of this random data. SplitPath exploits the flexibility of code mobility in active networks to address these problems. Especially the random generation is studied in more detail.

**CR Categories and Subject Descriptors:** C.2.0 [Computer-Communication Networks]: General; C.2.4 [Computer-Communication Networks]: Distributed Systems; C.2.6 [Computer-Communication Networks]: Internetworking.

**General Terms:** Security, Experimentation.

**Keywords:** Active networking, intelligent networks, code mobility, random generation, one-time pad, cryptography.

## 1 Introduction

A wide variety of encryption algorithms is in daily use to protect data communications. Good encryption algorithms base on a complex theoretical foundation, which is sometimes secret. The average user can only guess the strength of an algorithm by looking at governmental regulations for the key length of the algorithm (no regulations means bad algorithm). Nevertheless, none of the commercial encryption algorithms offers perfect security. New analysis methods or hidden back doors may always come to daylight. However, there exists a well-known algorithm which is very simple and perfectly secure: the One-time pad [Sch96]. Here is how it works. Assume you have a bit string $P$ of $n$ bits ($P, p_i \in \{0, 1\}, i \in 0..n$) which you want to encrypt. For that purpose you take a string of equally distributed and independent random bits $R, r_i \in \{0, 1\}, i \in 0..n$ and xor (addition modulo 2) it bit-wise with $P$, resulting in the ciphertext $C, c_i = r_i \oplus p_i$. To decrypt $C$, $R$ is xor-ed to $C$ again. This works because $r_i \oplus (r_i \oplus p_i) = p_i$. After the decryption $R$ must be destroyed. It is assumed that once $R$ is generated, it is used for encryption and decryption *only once*. That's why the scheme is called One-time pad. Under these assumptions the algorithm is provably secure. It is impossible to gain any knowledge of $P$ without knowing $R$, because *every possible plaintext* could have lead to a given ciphertext. Furthermore, in contrary to commercial encryption algorithms, the One-time pad needs only one very light-weight operation (xor) for encryption and decryption. However, the One-time pad is not practical for secure data communication for the following reasons: (1) It needs a bit-stream $R$ of true random values with the same length as the message. (2) The receiver and the sender must both possess the same $R$. How can $R$ securely get there?

This paper presents an approach using active networking [CBZS98, TSS$^+$97] to address both problems. An active network consists of active (programmable) network nodes. The data packets that are transmitted through an active network can contain code that the active nodes execute. Active networking is an instance of the mobile agents paradigm tailored to networking needs. Active network packets (also called capsules) access the networking functionalities of the nodes (e.g. forwarding and routing) and change these functionalities for packets or classes of packets.

This paper presents how (for a given application scenario) active networking enables us to use the One-time pad with its provable security and light-weightedness for secure data communication.

In section 2 we present the basic idea how to address the distribution of the random (problem 2). Section 3 describes how to generate the necessary random (problem 1) and what the pitfalls are. An existing implementation using the well-known active networking tool ANTS [WGT98] is presented in section 4. Section 5 presents performance measurements and section 6 concludes.

## 2 Distribution of Keys and Data

We said that with enough good random bits available, we can create an uncrackable bit stream using the One-time pad. However, the receiver must also possess the random bits. A straight-forward solution is to first deliver the random bits in a secure manner and later transmit the data. The sender could, for example, hand a magnetic storage tape to the receiver, containing the random bits. This is a secure but not very flexible application of the One-time pad. The use of a secure communication medium[1] allows the communicating parties to communicate later using an insecure communication medium. But even if the medium for sending the random is not secure, the scheme still works as long as no attacker has access to *both* random and message bits. This principle is e.g. used when encryption keys for data communication are exchanged using the postal service or the telephone system. The SplitPath idea goes one step further. The random bits (interpreted as the key) and the cipher text bits (plaintext xor-ed with the random bits) are sent along at the same time on the same media but *on different paths*. In general this scenario is not secure any more, since an attacker can eavesdrop both the random string and the encrypted message and thus easily decrypt the original message. In a network with centralised management at least the network provider will always be able to do this. However, if the network is partitioned in autonomous sub-networks (domains), as for example the Internet is, and if the two paths are entirely in different domains, an attacker will have significantly more trouble. Thus, the application of SplitPath has the following prerequisites:

- SplitPath traffic enters the untrusted networks only in split form (either random bits or xor-bits).
- The paths of corresponding random bits and xor-bits never involve the same distrusted network.
- The distrusted networks do not trust each other.

These prerequisites limit the application scenario of SplitPath, but there are cases where the prerequisites are met. For example in the Internet some commercial network providers compete with each other and offer redundant paths. Also, physical networks using different technologies (e.g. optical networks and satellite links) provide an application platform for SplitPath. Furthermore, the geographical location of-, and the relation between some nations can provide an ideal application scenario for the SplitPath scheme. The generic situation is depicted in figure 1.

[1]Assuming that the physical delivery of the magnetic tape is secure.

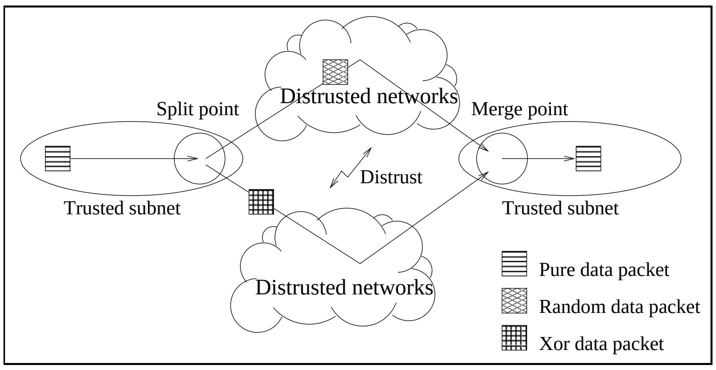


Figure 1: The application scenario for SplitPath.

We distinguish between the sender, the receiver, a split point and a merge-point. Obviously, the split- and merge point needs to be located on a trustworthy site. Depending on the sender, receiver and the network topology the ideal location of these points varies, thus their implementation cannot be preconfigured in few network nodes. Active networking brings the flexibility we need here. With active networking the data packets can dynamically setup their private split (merge) functionality in the nodes which are appropriate for them. No router configurations are necessary. SplitPath capsules contain code that dynamically implements the split and merge capabilities and controls the routing of the packets. These mechanisms are described in section 4.

The next section shows how active networking can also help us getting the ‘good’ random bits which are needed in the split point.

## 3 Generating Random

For SplitPath, like for most crypto-systems, the availability of high quality random material is crucial. The random bits must be independent and equally distributed in order to be unpredictable. However, it is not easy to acquire such random in a computing environment. Real random values can solely be created by real world interaction. Examples are: mechanical random devices (e.g. lottery machines, dices), physical random (e.g. radioactive decay) and human behaviour (e.g. keyboard interrupt times). Multiprocessing and networking devices can also be a source of random bits [Wob98].

Here’s an example (adapted from the UNIX security tool SATAN [FV93] which uses the current process and network status to generate random bits:

```
(ps -el & netstat -na & netstat -s & ls -lLRt /dev & w) | md5
```

Note that the Message Digest 5 (MD5) [IETF RFC 1321] secure hash algorithm is used to scramble the text output of the other programs into a 16 byte sequence.

In SplitPath, we propose to use unpredictable traffic characteristics as seen in networking devices and generate random bits with them. Active networking allows the capsule to use its *own* varying queueing times within the network. The capsule, being autonomous, can keep information about its creation time and ask the nodes on its way about the local time. Note, that clock skew is actually good for the random generation because it introduces another factor of incertitude.

The idea is that by traveling through the net each capsule generates random bits for its own use. However, e.g. an IP packet can contain 64 KBytes of data. It needs the same amount of random bits for applying the One-time pad.

## 3.1 Quantity and Quality of SplitPath Random

The quantity of random bits gained by network performance measurement is limited. This is a problem but we have many options to cope with the situation:

- Limit the payload of the capsule. Programmable capsules can fragment themselves to smaller payload sizes. This produces more packets and thus more random data per payload byte. It also adds bandwidth overhead. Note however, that congestion eases the production of random bits, since it involves a lot of non-predictable behaviour (see also section 5).

- Generate more random bits by sending empty packets. When capsules can store data in nodes (see section 4) then we can use the idle time to send packets without payload that store their performance statistics in the split-node. Later, when real payload is transported, the capsule can use the stored information to generate random.

- Multi-hop random generation. If the capsule executes at several nodes before the split node, it can calculate its performance statistics there, too. The capsule takes the gained random bits with it and uses them at the split node. Care must be taken when distilling random bits from several nodes, because they are probably not completely independent.

These options do not affect the strength of the One-time pad, but they limit the effective throughput of data. Another approach is to use a pseudo-random function [Sch96]. This function uses the collected random data as a seed and generates a bit sequence of arbitrary length that can be used to xor the data. However, by 'stretching' the random bits like that, we give up the perfect security of the One-time pad. The next subsection contains a detailed analysis.

## 3.2 Pitfalls of Stretching Random Bits

When less random bits than message bits are available, we need a function $f$ (the pseudo random generator) that uses the available random bits to generate bits to replace the missing random bits. The generated bits should be equally distributed. The function $f$ cannot be assumed to be secret. This is problematic, since we loose the provable security of the One-time pad. In this section we outline a possible attack given such an $f$ and knowledge about the syntactical structure of the encrypted message. Then we describe how to rule out this attack.

**The assumptions and the attack.** Let's assume that $f$ works block-wise and stretches the random bits by factor $b$ to encrypt all data bits in the block. E.g. if $b = 2$, one random bit is extended to generate 2 pseudo-random bits which then are used to generate the One-time pad. Assume, that the data bits (clear text) encode language-specific digits using $l$ bits per digit (e.g. $l = 8$ for one ASCII byte). Furthermore, be $p$ the average probability that a *random* language-specific digit in the message data stream is syntactically correct. E.g. in the ASCII string 'What is the next lette_', not all letters can be appended, so the $p$ for ASCII encoded English language is obviously smaller than 1. These assumptions given, an attacker can break the 'stretched random' One-time pad by a brute-force attack. Here is a stupid[2] version of an attacking algorithm. (1) choose a search depth $d$ (additional search depth may help attacking). (2) For all bit-combinations of $dl$ bits do: (2.1) extend the bits with the stretching function $f$. (2.2) Apply the One-time pad to the ciphertext. (2.3) Apply a syntactical check to the result. If the check is okay then (2.4) output the result, consume the ciphertext bits and go to (1). (3) Output that the cracking has failed.

**Example.** Assume that an attacker can sniff the One-time pad output `7wØ=:Þ?/` (8 bytes), of which he knows that it encodes ASCII text ($l = 8$). He knows the function $f$ and also that the encryption used $b = 8$. Thus, the sender expanded one (unknown) byte of true random to eight bytes and xor-ed them to the (unknown) plaintext. The attacker uses the presented algorithm with $d = 1$. The algorithm generates all bit-combinations of 8 bits (all byte values), expands them with $f$ (since $b = 8$ this produces 8 bytes) and xor-s the result to the ciphertex (decryption try). Then it applies a simple dictionary check. Here's what the output of our prototypical implementation of the cracking algorithm looks like (`?` represents a non-ASCII byte value):

```
??u???hw    ??u???h?    ??u????x    ??u?????    ??u??nix    ??u??ni?    ??u??n?y    ??u??n??
a ?d'?cr    a ?d'?c?    a ?d'??s    a ?d'???    a ?d'uds    a ?d'ud?    a ?d'u?t    a ?d'u??
a ?d??ds    a ?d??d?    a ?d???t    a ?d????    a ?d?ret    a ?d?re?    a ?d?r?u    a ?d?r??
a ??c?ds    a ??c?d?    a ??c??t    a ??c???    a ??cret    a ??cre?    a ??cr?u    a ??cr??
```

[2] An improved version would also contain backtracking.

```
a ????et    a ????e?    a ?????u    a ??????    a ???sfu    a ???sf?    a ???s?v    a ???s??
a sec?ds    a sec?d?    a sec??t    a sec???    a secret ***Check ok!***

The original message is revealed to be 'a secret'.
```

**Analysis.** The success of this algorithm is determined by the fact, that in case the stretching of $f$ produces a meaningful result, the result must be the cleartext data. This assumption is not necessarily true. The following analysis shall bring more clarity. Step (2) generates all value-combinations of $dl$ bits. By applying $f$ to these bits we get $bdl$ bits, which encode $db$ message specific digits. The probability that these digits are meaningful is $p^{db}$. Therefore, the expected number of meaningful texts $E$, that the algorithm produces in the iteration of (2) is:

$$E = 2^{dl} p^{db} = 2^{d(l + blog_2(p))} \tag{1}$$

If this number is significantly smaller than 1, we can be sure that a meaningful decryption try (step 2.4) really *is* the clear text and not just occurred by chance. If the number is significantly higher than 1, the decryption try can as well be correct by pure hazard. Note, that for $b = 1$ (pure One-time pad), the formula reflects the mean number of *all* syntactically correct texts of that length. We can also see that the search depth $d$ only helps cracking, if the term $l + blog_2(p)$ is smaller than 0. Let us assume, that the data carry bytes so $l = 8$. Thus the brute force attack is possible if $b > -8/log_2(p)$. In the English language, there are only an average of two letters appendable to English text. If the text is encoded in ASCII this leads to $p \approx 1/128$. We can now apply the formula to see what the maximum secure stretching factor $b$ in that case is: $b < -8/log_2(1/128) = 8/7$. This means, that at least 7 true random bits must be used to encrypt 8 bits of an English ASCII encoded text message, in order to avoid the possibility of a brute force attack. We can also come to this result by following the information theoretical approach [Sha49].

**Information theoretical analysis.** In [Hel77] Hellman showed that the expected number $P$ of different keys that will decipher a ciphertext message to some intelligible plaintext of length $n$ (in the same language as the original plaintext) is given by the following formula:

$$P = 2^{H(K) - nD} - 1 \tag{2}$$

$H(K)$ is the entropy of the crypto-system used, $n$ is the number of digits sent and $D$ is the redundancy of the coding of the plaintext message (e.g. ASCII encoded English text).

$P$ is basically the same as $E$ in our previous calculation, but it deals also with the fact, that there must be one intelligible plaintext (the original one), which boils down to the -1 in this formula. Without this, we are now going to show that the equations (1) and (2) are equivalent. Since we use an equally distributed keyspace (the seeds are random) $H(K)$ is equal to the key length in bits[3]. However, our seed bits are generated per packet thus the number of key bits (and thus $H(K)$) is dependent on the number of bytes sent ($n$). The stretching factor $b$ of the previous calculation is useful here, since we can express the key length for a message of length $n$ as follows: $H(K) = nl/b$ (as before, $l$ is the number of bits used to encode a digit). Thus we get:

$$P = 2^{nl/b-nD} - 1 = 2^{n(l/b-D)} - 1$$

Then, $n$ is the number of encrypted digits. Using the parameters of the first approach this is the search depth multiplied with the stretching factor: $n = db$.

$$P = 2^{db(l/b-D)} - 1 = 2^{d(l-bD)} - 1$$

Finally, we need to transform the redundancy $D$ which is the number of bits which is not really used to encode a digit. The number of bits used to encode a digit is $l$, but only $p2^l$ digits are appendable on the average (assumption of the previous paragraph). Therefore, only $log_2(p2^l) = l + log_2(p)$ are necessary to encode a digit. So the redundancy is $D = l - (l + log_2(p)) = -log_2(p)$. By substitution of $D$ we have now reduced the formula (2) to (1) (except of the substraction of the one know solution):

$$P = 2^{d(l-b(-log_2(p))} - 1 = 2^{d(l+blog_2(p))} - 1$$

**Random expansion in practice.** We have shown that when we want to 'stretch' the random data by larger factors we can not rely any more on the perfect secrecy of the One-time pad. A brute-force attack is then theoretically possible. Instead, we have to carefully design the pseudo-random generator (the $f$ function). First of all, the seed length must be large. We propose 128 bits. The previous paragraph showed that a brute attack in principle will lead to the decryption of the packets since in the worst case there is only one random bit per packet. For IP packets $b$ can therefore become as large as $2^{19}$. However, in practice the attack will not be successful, because there are too many bit combinations to try (an average of $2^{127}$). If a million computers would each apply a billion decryption tries per second the average search would still last about 5*$10^{15}$ years.

Second, the pseudo random generator should resist cryptanalysis. Many such generators exist and are used for so-called stream ciphers [Sch96]. Using SplitPath with expanded random is very similar to using a stream cipher in that both xor

[3]The formula was deduced with a fixed key length in mind.

the plaintext with a secure pseudo random bit stream. However, SplitPath differs from stream ciphers in that it uses the pseudo random generator only at the sender side. Furthermore, the flexibility of an active network platform allows SplitPath to dynamically change the generator used, even during an ongoing communication. Finally, the seed of the generator is updated frequently (as soon as enough random bits have been collected by the capsules), and the seed is random. This is different from e.g. the stream cipher A5 (used for mobile telephony) which uses a preconfigured seed.

## 4 Implementing SplitPath in an Active Network

### 4.1 Implementing SplitPath with the Active Node Transfer System ANTS

We implemented the SplitPath application using the active node transfer system ANTS [WGT98]. ANTS is a Java based toolkit for setting up active networking testbeds. ANTS defines active nodes, which are Java programs possibly running on different machines. The nodes execute ANTS capsules and forward them over TCP/IP. ANTS defines a Java class `Capsule`. The class contains the method `evaluate` which is called each time the capsule arrives at a node. New capsule classes can implement new behaviour by overriding the `evaluate` method. The node offers services to the capsule such as the local time, forwarding of the capsule and a private soft-state object store (called *node cache*). Collaborating capsules can be grouped to protocols. Capsules of the same protocol can leave messages for each other using the node cache. We defined such a protocol to implement the SplitPath concept as presented in section 2 by introducing three new capsule subclasses.

- The `Pathfinder` capsule marks nodes as splitting or merging points and sets up the split paths using the node caches. Note that several split and merge points can be set up per communication.

- The `Normal` capsule is the plaintext message carrier. It checks if it is on a splitting point. If not, it normally forwards itself towards the destination. If it is on a splitting point, it applies the One-time pad computation to its payload. This results in two `Splitted` capsules, one carrying the random bits, the other the xor-ed data in the payload. The `Normal` capsule tells the node to forward the two newly produced `Splitted` capsules instead of itself, thereby using the information that was setup by the `Pathfinder`.

- The `Splitted` carries the encrypted- or the random data along a seperate path. It forwards itself using the information that was setup by the

`Pathfinder`. It checks if it has arrived on a merge point. If so, it checks if its split twin has already arrived. In that case it xor-s their contents (decryption) and creates an `Normal` capsule out of the result. If the twin is not there, it stores itself on the node cache to wait for it.

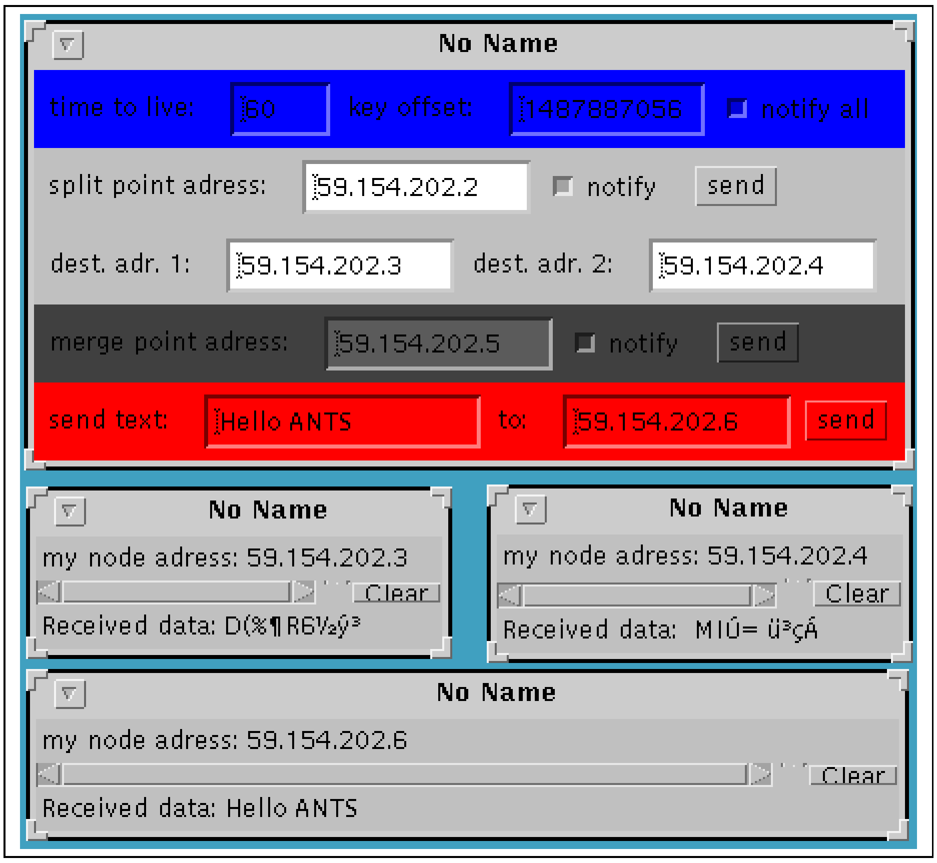
**Applications and Interfaces.** We wrote two applications that send and interact with the implemented capsules. The first application provides a graphical interface which allows the user to dynamically set up split and merge points using the `Pathfinder`. Furthermore, the user can enter and send text data using the `Normal` capsule. We also extended the nodes with a sniffing functionality. Such 'spy nodes' log capsule data to validate and visualise the SplitPath encryption. Figure 2 shows the application window (top) with two spy nodes and the receiver node (bottom). The received and the sniffed data is displayed in the status bar of the windows.

We implemented a second application which transfers a file (optionally several times) in order to test the performance by generating load.

Due to space limitations we cannot go into further details. Those are available in [Bro00]. However, we have to get a closer look at the collection of random bits because the random generation is crucial for the security of SplitPath.

## 4.2 Random Generation and the Application of the One-time pad

In order to be able to send large packets, we decided to use pseudo random generators to extend the collected random bits (see section 3). Each `Normal` capsule contains its creation time. When arriving at a split node, it uses this time and the node's current time to calculate the delay it has so far. It stores the last few bits as random seed. The number of bits is configurable and depends on the clock resolution. Unfortunately, the Java system clock resolution used by ANTS is bound to one millisecond and the delay is in the order of few milliseconds. Therefore, we used only the least significant bit. Thus, every packet stores one random bit in the split node (using the node cache). This bit can be considered as random, since it is influenced by the speed and current usage of computing and networking resources. As said before, the chosen seed length is 128 bits. So for every 128th capsule a complete seed is available. This capsule uses the seed to store a new random generator in the node cache. The next capsules use the generator for their encryption, until enough fresh random has been collected to install a generator with the new seed (key refreshing). For the bootstrapping we foresee two schemes. Either 128 empty packets are sent, or a (less secure) seed is used that can be generated by the first packet.



No Name

time to live: 60 key offset: 1487887056 notify all

split point adress: 59.154.202.2 notify send

dest. adr. 1: 59.154.202.3 dest. adr. 2: 59.154.202.4

merge point adress: 59.154.202.5 notify send

send text: Hello ANTS to: 59.154.202.6 send

No Name

my node adress: 59.154.202.3

Clear

Received data: D(%¶R6½ý³

No Name

my node adress: 59.154.202.4

Clear

Received data: MIÛ= ü³çÁ

No Name

my node adress: 59.154.202.6

Clear

Received data: Hello ANTS

Figure 2: The graphical user interface.

We have implemented two random generators. The first one is based on the secure one-way hash function MD5[4]. The seed is stored in a byte array of 16 bytes. Furthermore, there is an output buffer of 16 bytes containing the MD5 hash value of the seed buffer. The generator delivers 8 bytes of the output buffer as pseudo-random values. Then, the seed is transformed and the output buffer is updated (MD5 hash). The 'one-way' property of MD5 assures that an attacker cannot reconstruct the seed. Thanks to the avalanche property of MD5 the transformed seed produces an entirely different hash. Our seed transformation is equivalent to the increment of a long integer (8 bytes) by one. Thus, the seed only repeats after $2^{64}$ transformations. Long before that SplitPath will replace the seed with freshly gathered random values. We think that for the presented reasons this pseudo random generator is reasonably secure. However, since we are not cryptographers, we implemented also the pseudo random generator of RC4. RC4 is a stream cipher developed by RSADSI. It is used in applications of e.g. Lotus, Oracle and Netscape (for details see [Wob98]).

## 5 Evaluation of SplitPath

In order to evaluate SplitPath we ran the implementation on our institute network. Six ANTS nodes were set up on six different machines (sender, receiver, a split and a merge node and two sniffers; see figure 3). The split node ran on a SPARCstation 5/170. The encrypted capsules ran over two different subnets and were merged in a machine of a third subnet. The subnets are 100 Mbps Ethernets. We used the aforementioned file transfer application to generate load. Our interest was focussed on the quality of the encryption. We measured this by collecting all generated seeds and apply statistical tests. Furthermore, we applied statistical tests to the MD5 pseudo random generator presented in the previous section.

In order to test the quality of the MD5 random generator, we initialised it with all seed bytes set to zero. Then we fed its output into a framework for statistical tests. Testing with samples of 40 MByte size, the produced data succeeded the byte frequency test, the run test [Knu81] and the Anderson-Darling test [IETF RFC 2330]. This is no prove that the generated pseudo-random bits are of high quality, but it shows that they are not flawed.

**Seed generation.** We evaluated the generated seed bytes using statistical tests. For example we analysed 3K seed bytes (192 complete seeds, protecting 24576 packets). The seeds pass the byte frequency test ($\chi^2$ test on the distribution of the measured byte values [Knu81]). The byte frequency of this set is shown in figure

[4]ANTS includes the Message Digest 5 (MD5) [IETF RFC 1321] functionality.

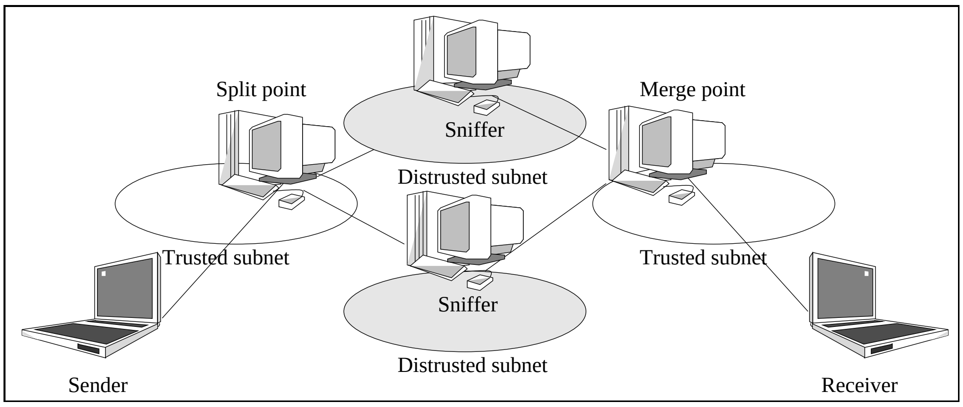


Figure 3: The network topology for the evaluation.

5 (left), where the number of occurrences of each byte value (0-255) is counted. Unfortunately, we also experienced seed generation that was not uniformly distributed in situations with low load. Figure 5 (right) shows the value of each seed byte (interpreted as 2-complement) as it was created in time.

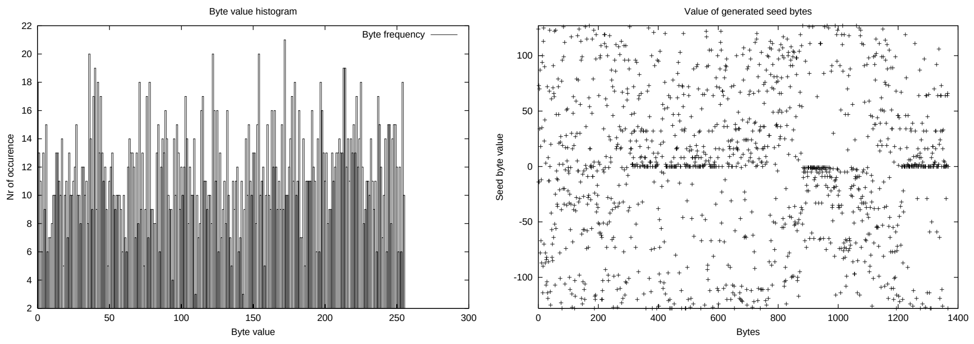


Figure 4: Good seed (left), bad seed (right).

We see some concentrations of byte values especially around the value 0. Our investigation revealed three reasons that come together to form this effect. (1) The coarse resolution of Java’s clock. (2) ANTS does not send one capsule per packet, and the packet code is not send within the capsule, but dynamically loaded and cached. Thus, consecutive capsules are handled immediately after each other without delaying I/O operations. (3) The local network used has very low delay and jitter.

These problems are not discouraging because normally they do not all come together and there are also countermeasures. By introducing congestion we could show that given realistic wide area delays as studied e.g. for the Internet [EM99], the seeds will be equally distributed. Also, SplitPath does not necessarily rely on one single random source (the network jitter). We also foresee the exploitation of additional random sources offered by active networking e.g. execution times of capsules or properties of the node cache. Finally, we can once again exploit the fact that the capsules are programmable. The procedure for collecting seed values can easily be extended to contain statistical methods to test the seed before it is used. So if for some unforeseen reason the seed is not good enough, the capsule uses the old random generator a little longer until more random bits are collected. Also, methods described in [Sch96] can be used by the capsule to distill random from biased random bits.

## 6 Conclusion

In this paper we presented SplitPath, a new application for the well-known and provably secure One-time pad encryption scheme. SplitPath uses the ability of active networks and the trust relations in heterogeneous networks to solve the two problems which otherwise render the One-time pad scheme useless: the random generation and the distribution of this random data. SplitPath can dynamically set up and use disjunct paths through network domains that do not collaborate, such as competitive network providers or countries. Encrypted data and random data is forwarded on different paths. The One-time pad assures that only when both data sets are available, the data can be decrypted again. With the active networking of SplitPath the decryption can be done at the receiver of the data or at a merging point in a trusted domain of the active network. Active networking not only allows SplitPath to dynamically set up the One-time pad splitting inside of the network, it also helps to collect good random. This can be very useful for other crypto-systems, too. SplitPath implements data capsules that use the network delays that they experience as initialisation for the random generation. With SplitPath we present an application in a (albeit specific) environment which cannot be implemented using conventional ‘passive’ networking, thus we promote the future study and (hopefully) deployment of active networking.

## References

[Bro00] Marc Brogle. Active networking with ANTS. http://www.brogle.com/marc/uni/ants/ants.php, March 2000. Student project.

[CBZS98] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in active networks. *IEEE Communications*, 36(10), October 1998.

[EM99] Tamas Elteto and Sandor Molnar. On the distribution of round-trip delays in TCP/IP Networks. In *Proceedings of the 24th Conference on Local Computer Networks LCN'99*, pages p.172–181. IEEE Computer Society, October 1999.

[FV93] Dan Farmer and Wietse Venema. Improving the security of your site by breaking into it. ftp://ftp.porcupine.org/pub/security/admin-guide-to-cracking.101.Z, 1993.

[Hel77] M. E. Hellman. An extension to the shannon theory approach to cryptography. *IEEE Transactions on Information Theory*, IT-23(3):p. 289–294, May 1977.

[Knu81] D. E. Knuth. *The art of computer programming*, volume 2 Seminumerical Algorithms. Addison-Wesley, 2 edition, 1981.

[Sch96] B. Schneier. *Applied Cryptography*. John Wiley and Son, 1996.

[Sha49] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):p. 656–715, 1949.

[TSS$^+$97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[WGT98] D. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH '98*, April 1998. San Francisco.

[Wob98] Reinhard Wobst. *Abenteuer Kryptologie*. Addison-Wesley, 1998.