Sliding Encryption: A Cryptographic Tool for Mobile Agents

Adam Young*, Moti Yung**

Abstract. The technology of mobile agents, where software pieces of active control and storage (called mobile agents) travel the network and perform tasks distributively, is of growing interest as an Internet technology. Similarly, smartcard holders can be considered mobile users as they access the network at various points. Such mobile processing can be employed in large scale census applications in statistics gathering, in surveys and tallying, in reading and collecting local control information, etc.

This distributed computing paradigm where local pieces of data are getting accumulated in a mobile unit presents new information security challenges. Here, we point at some problems it poses and suggest solutions. The basic problem considered involves the design of a mobile agent that is capable of traversing an untrusted (curious) network while gathering and securing data from the nodes that it visits. We assume that some subset of the nodes may collaborate to track the agent, and we assume that snapshots of memory are taken at each node at times that are unpredictable to the agent. The data that is gathered must be securely stored within the agent and the adversarial nodes must remain oblivious to what is taken by the agent. In addition, the agent's movement throughout the network should be made difficult to trace. Furthermore, we assume that the agent is limited in storage capacity. To prevent the nodes from getting decryption capability, the agent must carry a public key for (asymmetric) encryption.

We present an economical solution that we call **"sliding encryption"**. This is a new mode of operation of public key cryptosystems that allows the encryption of small amounts of plaintext yielding small amounts of ciphertext. Furthermore, the encryption is performed so that it is intractable to recover the plaintext without the appropriate private key. We also describe how to modify sliding encryption so that the resulting ciphertexts are hard to correlate, thus making it possible to have mobile agents that are not easy to trace. Sliding encryption is applicable to mobile agent technology and may have independent applications to "storage-limited technology" such as smartcards and mobile units.

Key words: space-efficient encryption, sliding encryption, public key, RSA, mode of operation, mobile computing, network software agents, smartcards, WWW, network computing, applets, spiders, worms, viruses, cryptoviruses.

* Dept. of Computer Science, Columbia University, New York, NY, USA. Email: ayoung@cs.columbia.edu.

** CertCo New York, NY, USA. Email: moti@certco.com, moti@cs.columbia.edu

1 Introduction

Distributed agents is a relatively new technology which can be designed to perform many tasks. For example, an agent may travel from site to site in a network, performing local searches and gathering data on certain parameters (e.g., traffic load), so as to help control the network (e.g., redirecting the flow of packets to alleviate traffic). In an untrusted environment of non-malicious but curious nodes, such an agent needs to take measures to prevent traffic related data from falling into the wrong hands. If this data can be intercepted, it can be used for the purposes of traffic analysis, and if it can be tampered with, the performance of the system can be greatly hampered. Like the traffic data example, there are many distributed computations involving gathering of local pieces of data which require hiding the data from non-local nodes. The data eventually reaches the source of the agent. We will use the term *originator* to refer to the source entity that dispatches the agent to perform a certain task.

It is imperative that a data collecting and encrypting mobile agent conceal its decryption capability, thus our problem implies the use of public-key technology. This was pointed out in the context of the cryptovirus agent in [YY96]. The gathered data is made accessible exclusively to the originator when the gathered data is returned to it. The corresponding private key is not contained within the agent, and is kept secret by the originator.

In this paper we consider the problem of designing an agent that must gather small amounts of data from several nodes on a network, where each node is untrusted by the agent, and in particular the node may try to read information carried around by the agent. We further assume that the agent is restricted in the amount of data that it can store (e.g., on the order of kilobytes). As an example of this problem, consider the following. An agent contains a 128 byte public key, and must gather 1024 pieces of information from 1024 different nodes on a network. Suppose that it need only gather 4 bytes of information from each node. The greedy approach to this problem is to use a hybrid cryptosystem which selects a symmetric key per node, or to use the public key cryptosystem itself, both in ECB mode. In this case, ECB mode is most efficient. It follows that the agent must have the capacity to store 128k worth of data. However, the 128k of ciphertext would contain only 4k worth of plaintext. This is so, since each block of public key encrypted data has only a 4 byte value if data is encrypted directly. The rest of the block may be (should, in fact, be) random. The motivation for this work is therefore to study ways in which to public key encrypt small amounts of data, to yield small amounts of ciphertext, without compromising the overall security of the system. Note that due to the agent's mobility and the security constraints, it cannot accumulate a large block from many nodes and then encrypt it.

Specifically, we introduce what we call 'sliding' encryption that accomplishes exactly this. In a nutshell, sliding encryption is a way of enciphering a small amount of data within a larger block, and sliding away a small fraction of the result. This fraction constitutes the 'ciphertext' of the small piece of input plaintext. The sliding encryption scheme that we describe is based on RSA [RSA78], though the scheme is general enough to be used with any deterministic public key encryption algorithm.

2 Definitions and Background

Sliding encryption is a mode of operation of public key cryptosystems akin to using symmetric algorithms for stream ciphers. It is aimed at conserving space, rather than "fast" encryption (due to technological constraints the solution is public-key, and thus cannot be too fast). Recall that a self-synchronous stream cipher is a cipher in which each key block is derived from a fixed number n of preceding ciphertext blocks (e.g., [De83]). Sliding encryption has the same flavor, and it incorporates the idea of chaining from the mode of operation known as Cipher Feed-Back. However, in the scheme that we describe, each ciphertext block is derived from a state affected by all preceding ciphertext blocks, not just the preceding n blocks. It is possible to implement a sliding scheme that uses a fixed dependency of length n, but this is not applicable to our purposes. In this paper we do not concern ourselves with serial communications, but rather the problem of implementing efficient agents. Hence, the fact that all preceding encrypted data is lost if a ciphertext block is lost is not applicable to our problem (an interfering party can simply delete the entire agent if it is found).

Definition 1:

Sliding Encryption is a mode of operation of public key cryptosystems based on chaining, satisfying the following properties:

1. Encryption is granular, that is, a small amount u of plaintext bytes can be encrypted using a sufficiently large public key (We cannot delay the encryption by accumulating many small pieces together!)
2. It is computationally difficult to determine each u byte piece of plaintext without knowing the correct private key
3. The system is resistant against known plaintext attacks

The purpose of (1) is so that an agent that employs sliding encryption can gather small amounts of data from several nodes across the network, and public key encrypt each piece of data without wasting an excessive amount of storage space. Since we assume that an adversary may take a snapshot of the agent at any time, we cannot simply gather small amounts of data until we have enough to perform a space efficient encryption, because during this period a snapshot may be taken. The purpose of (2) goes without saying. Requirement (3) is needed in the case that an adversary has a good chance of guessing what the agent has gathered, and merely wishes to know what the agent has gathered. The sliding implementation that we present incorporates the notion of probabilistic encryption [GM84] so as to foil such an attack. Hence, the adversary must guess the random string used in the probabilistic encryption to verify that the guessed plaintext is correct.

At times throughout the rest of this paper we will refer to the agent as a probabilistic Turing machine (coin flipping computation). This model of computation is necessary, since a deterministic agent cannot hope to conceal, at the very least, the nature of the information which it gathers (its location, its structure, etc.), and cannot resist known plaintext attacks for small plaintexts. This randomization is achieved via inclusion of a function that generates random bits using events in the environment, such as AT&T's truerand function [MB95].

Related Work

Agents are loosely defined as personal software assistants with authority delegated from their users [Ch96]. Agent technology traces back to the early 1980's when the notion of a worm was invented at Xerox Palo Alto Research Center [Mc89]. John Shoch and Jon Hupp, two researchers at PARC, were interested in the concept of distributed processing, and thought that worms may be a novel way of accomplish distributed computing [Sl94]. Agent technology has evolved to include computer viruses, worms, web wanderers, etc. In the Crypto community they were suggested for factoring and other exhaustive search tasks [Wh89]; also, more recently, malicious agents called cryptoviruses have been proposed that make use of public key cryptography to mount attacks on their hosts [YY96]. Modern Internet technology gives rise to mobile applets and other mechanisms that exploit agents. An example of collective computation tasks and an algorithm allowing agents to "meet" after their deployment within a network has been discussed in [YuYu96].

Agents can be roughly categorized into those that are mobile and those that are immobile. Web wanderers, like Matthew Gray's WWW Wanderer (1993), are immobile, and gather information about the WWW for its user, while worms and distributed migratable tasks are examples of mobile agents. In this paper we will concern ourselves with the study of mobile agent technology in untrusted network environments. Also, smartcard technology which is readable, can nevertheless use encryption to perform secure and economical mobile data collection.

Remark: We note that there are numerous other security concerns with mobile agents. For example, validating the authenticity of the agent at the node and tracing (audit trail kept at or for the agent), assuming that the processing at the nodes is "as expected" and does not introduce "anomalies". Here we do not treat such security concerns, concerns which should be part of "network wide computing".

3 RSA Based Sliding Encryption

We will describe a sliding encryption implementation for m byte RSA keys (where m is a power of 2) [RSA78]. We will assume that the granular size of the plaintexts to be encrypted is (w.l.o.g.) fixed and that each is u bytes in length. Each piece of plaintext will be encrypted along with a v byte random string, where v is at least, say, 12 bytes in length. Let $t = u$ concatenated with v. We assume that t is

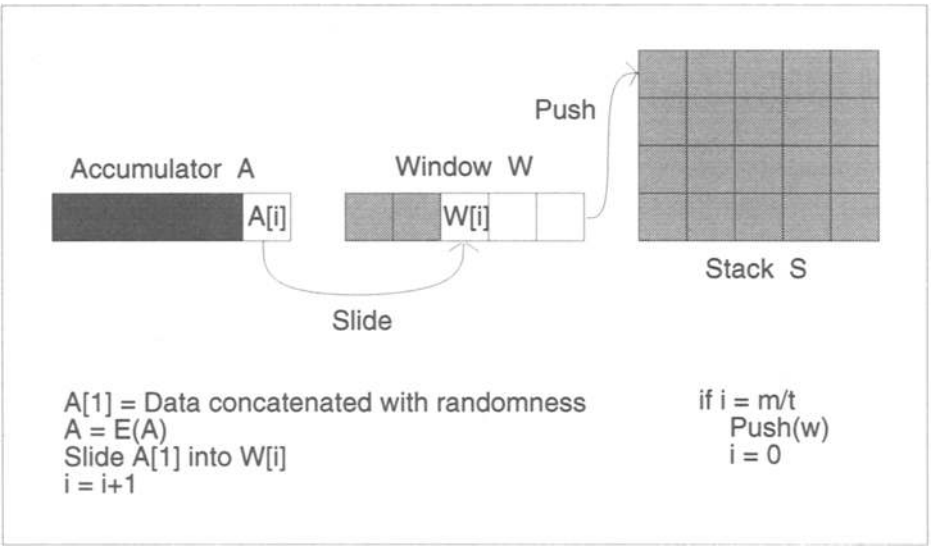

Fig. 1. Sliding Encryption Mode.

a power of 2 and $t \ll m$ and also that t divides m. As we shall see, the v random bytes that are added make the scheme a probabilistic encryption scheme. The v bytes constitute the random string used in the probabilistic encryption (analogous to an IV). The sliding encryption mode of operation (SE) makes use of a stack S, an m byte accumulator A, and an m byte window W. Stack elements $S[i]$ are m bytes in length.

The procedure Push(x) pushes an m byte quantity onto the stack. The function Pop(x) returns an m byte quantity from the stack. The function Empty(X) returns true iff stack X is empty. We assume the existence of an implicit state variable that points to the top of the stack. Accumulator elements $A[i]$ and window elements $W[i]$ are t bytes in length, where $1 \leq i \leq m/t$. $A[1]$ contains the least significant bytes of A, and $A[m/t]$ contains the most significant. The same applies for W. We assume the existence of a source of truly random bits (the bits must not be easily "guess-able"). Let the plaintexts be denoted by $a_1, a_2, ..., a_k$. Let the public key encryption function be denoted by $E()$, and let the corresponding private decryption function be denoted by $D()$.

Initially the stack is empty, and the accumulator is set to a random element in Z_n^*. To encrypt, new data is put in $A[1]$. We take a_1, The u lower order bytes of $A[1]$ are then set to be a_1. The v upper order bytes of $A[1]$ are set to be random. E is then applied to the accumulator, thus modifying all the elements of the accumulator. Note that an uneven "Feistel-like" preprocessing as in [BR94] which assures the accumulator is in Z_n^* and which pseudo-randomizes the accumulator based on the v random bits can take place here. In this case the definition of $E()$

and $D()$ will include this Feistel-like pre- and post-processing (the processing involves running v through a pseudorandom generator generating a pad that is EXORed with the rest of the bits then hashing the resulting value and EXORing the result with v.) Base on [BR94] the value encrypted is a "probabilistic encryption" if the preprocessing is via "random oracle like functions". We then set $W[m/t] = A[1]$. By setting $W[m/t] = A[1]$ we thereby 'slide' (shift) t-bytes of ciphertext from the accumulator into the window. To encrypt a_2, we set the u lower order bytes of $A[1]$ equal to a_2, and make the v upper order bytes of $A[1]$ random. We then apply E to A, and slide $A[1]$ into $W[(m/t) - 1]$. To encrypt a_3, we set the u lower order bytes of $A[1]$ equal to a_3, and make the v upper order bytes of $A[1]$ random. We then apply E to A, and slide $A[1]$ into $W[(m/t) - 2]$. This process is continued until $a_{m/t}$ has been slid into $W[1]$. At this point Push(W) is executed.

The next values $a_{(m/t)+1}$ through $a_{2m/t}$ are encrypted in the same way, and also pushed onto S. A count of the number of array elements in W that are currently in use is stored in the variable 'count'. This entire process is described by the following pseudo-code.

```
function InitializeSE
Input: Nil
Output: S, A, W, count

/* Initializes data structures to allow sliding encryption */
begin
count = 0       /* stores the number of elements of W in use */
S = empty
Initialize A to contain all random bytes
A = E(A)     /* make sure A is less than public key modulus */
end

function SlidingEncrypt
Input: RSA Public Key, S, A, W, count, a[1],a[2],...,a[k]
Output: S, A, W, count

/* Encrypts one or more plaintext blocks using RSA in SE mode */
begin
for i = count up to k-1 do
        set the v upper order bytes of A[1] to be random
        set the u lower order bytes of A[1] to be a[i]
        A = E(A)
        index = i mod m/t
        W[m/t - index] = A[1]  /* slide ciphertext into W */
        if index = m/t - 1 then
              Push(W)
              count = 0
        else
```

```
            count = count + 1
end
```

To perform sliding encryption, the function InitializeSE is first invoked to initialize the necessary data structures. SlidingEncrypt is then invoked for each plaintext value a_i. Decryption is exactly the opposite of encryption. We pop elements off the stack and slide elements from the window into the accumulator and decrypt them. The following function decrypts all values at once.

```
function SlidingDecrypt
Input: RSA Private Key, S, A, W, count
Output: a[k],a[k-1],...,a[1]

/* Decrypts all ciphertext blocks contained in A, W, and S. */
begin
if count > 0    /* first empty out the window */
        for i = (m/t - count + 1) up to m/t do
                set A[1] = W[i]
                A = D(A)
                output the u lower order bytes of A[1]
while Empty(S) = false do
        W = Pop(S)
        for i = 1 up to m/t do
                set A[1] = W[i]
                A = D(A)
                output the u lower order bytes of A[1]
end
```

Note that any deterministic public key functions E and D can be used. Probabilistic encryption where the ciphertext space is larger than the plaintext space will not work. For example, if ElGamal [El85] were used, the accumulator would double in size with each application of E, hence preventing efficient storage of ciphertext.

Note also that A's state can be modified to be a function of various portions of the history or be pseudo-randomized. In addition a number of data blocks can be processed together, e.g., there are 10 elements so nine are put together and one is put by itself with an indication that the previous block was size nine (so that decryption can be made).

To perform the cryptographic operations (namely, the RSA encryption above) the agent can utilize cryptographic libraries at the nodes or must include such cryptographic tools which have to be small in size themselves. See the discussion in the appendix.

4 Security of RSA Based Sliding Encryption

Consider the case of an agent that employs sliding encryption in an untrusted environment. The data a_i gathered by the agent is at risk of being included in a

snapshot from the time the agent gathers it through the time at which E is applied to it. Consider the time before and after this event. In the case where the agent is a probabilistic automaton and its choice of which data to encrypt is probabilistic, and since we use probabilistic encryption the agent do not reveal its choice of data a_i to the observers. Namely, a_i cannot be inferred from the state of the agent prior to taking a_i. After a_i is encrypted, even assuming the adversary properly guesses that a_i was taken, the adversary must also guess v correctly, re-encrypt with v and a_i using a preimage of the agent, and compare with the current image. If v is on the order of 96 bits in length (12 bytes), then an exhaustive comparison will require $O(2^{95})$ encryptions on the average. An exhaustive search for a_i will require at least that. Clearly any previous data that is within the stack is at least as difficult to decrypt. (Recall that the Optimal Asymmetric Encryption like mechanism, turn each block encryption to "probabilistic encryption"). Thus, criterion 2 and 3 of the definition of sliding encryption is met. Clearly criterion 1 is met. The system therefore constitutes a sliding encryption scheme as defined in definition 1.

Now consider the efficiency of the scheme. For each piece of plaintext we add v bytes. For k pieces of plaintext, the scheme requires $O(kv)$ storage in addition to the $O(ku)$ bytes corresponding to the plaintext. The total storage is therefore $O(k(u+v))$. This contrasts with the greedy approach that requires $O(km)$, where $m \gg v$.

Note that overall we use more space than an ECB encryption would if we could delay the encryptions and treat the entire gathered information as a stream of data. But, recall that we cannot delay encryptions due to the risk of having the plaintext captured in a snapshot.

5 Applications

Sliding encryption can be used to efficiently store data in embedded cryptographic devices such as smartcards that have limited storage capacity. Depending on the "guess-ability" of the plaintext and the number of bytes in the plaintexts a_i, the number of additional v bytes of randomness may be tuned. In addition, the accumulator can be a special machine register that is specifically designed to have multi-precision arithmetic performed on it.

Another obvious application of sliding is for tracking the path traversed by an agent generating private audit trails. In this case, the a_i can be 32 bit IP addresses corresponding to the nodes on the Internet that the agent has traversed. We can take v to be 12 bytes. In this case t is a 16 byte quantity. By using a 512 bit RSA key (trying to minimize size overall), we save 48 bytes on each encryption of an IP address. For the sake of argument, suppose that b is 32. The agent therefore contains an accumulator of 64 bytes, a window of 64 bytes, and an array S of 2048 bytes, for a total of 2,176 bytes. The originator releases several such agents, and upon returning, the originator is able to see the last 32 Internet nodes that were traversed by the returning agent. In this example, the adversaries clearly know the plaintext.

5.1 Making Agents Less Traceable

Another security consideration is traceability, which is apart from the security of the plaintext itself and concerns concealing the agent's whereabouts from the environment (related to the last application). Suppose that a subset of the adversaries have realized that an agent passed through after the fact, and all of them took snapshots of the agent, but failed to take snapshots while the a_i's were present in the agent in plaintext form (this is just a working assumption). Suppose further that these adversaries are willing to collaborate and want to verify that the same agent passed through their nodes, and they want to learn in what order the nodes were traversed. So, here we are interested in what can be learned about the path of the agent based on the state information of the agent. The collaborating adversaries are equipped with a set of images of agents, and with each image is a corresponding time stamp. We will assume that the agent chooses the nodes that it traverses uniformly at random from the set of all nodes on the network.

We must assume that the agent can modify the version of itself that moves on just before moving to another node and that this cannot be caught by the current node itself (otherwise untraceability is impossible). Alternatively, we can assume that the subset of adversaries is disconnected in the network and the agent manages to modify itself before it reappears at a node controlled by this set. This problem is similar to problems of untraceability in anonymous remailing schemes [Cha81]. In this respect, a mobile agent needs to be able to, in some sense, anonymously "remail" *itself*. Note that we are trying to minimize the risk here rather than avoiding the problem all-together. Our major problem being the fact that an agent cannot "encrypt and decrypt" itself, since if it can, the nodes can also decrypt it; decryption is made available only to the originator of the agent.

To reduce traceability we need to pay careful attention to all the state information in the agent that is revealed to the adversaries after the fact. This state information consists of the accumulator values, the window values, the values associated with the stack, the count, and the stack pointer. As described, the stack will grow monotonically in size. This will help reveal the order in which the nodes are traversed. We must therefore conceal the size of the stack. We can minimize this problem if we are willing to fix the maximum amount of information that can be obtained, and if we adopt a 'most recently used' algorithm, for instance. In this case, the agent stores only the most recently gathered information. S is an array that initially contains random elements (that can be recognized after decryption), and a pointer to S is maintained. Let this pointer be denoted by 'ptr'. ptr is initially set at a random offset, and is incremented from the lowest index to the highest index. When the highest index element is filled with m bytes of ciphertext, the pointer wraps around back to the lowest index, thus overwriting any data previously stored there.

So far we have shown how to fix the size of S, thereby insuring that A, W, S, count, and ptr remain fixed in size over time. Now consider the contents of accumulator A. The accumulator cannot be used to link two images of agents,

since each accumulator value is a probabilistic RSA encryption of the previous. So, it remains to consider W, S, count, and ptr. It turns that in the scheme described thus far, both W and S can be used to successfully track the agent. Let b denote the maximum number of m byte blocks that can be stored in S. If b adversaries are willing to collaborate, then they have a chance at determining a path of length b that the agent follows Similarly the window can be used to trace out paths of length $m/t - 1$. In this case the adversaries track the agent by comparing the contents of S and W, respectively.

The following is how the traceability problem with W can be minimized. Note that when each a_i is encrypted, v truly random bytes are incorporated into the encryption. These v bytes of entropy can be used to seed a cryptographically secure random number generator, to generate a stream of size (count$*t$) bytes. This stream is EXORed with the contents of W that is currently in use. A is then encrypted as usual, thereby concealing the seed v. $A[1]$ is then slid over to the window. We redo this process if the resulting value for W will end up being greater than or equal to the public modulus. The v random bytes are therefore used to conceal the previous contents of the window and to probabilistically encrypt a_i, thereby concealing the previous contents of the accumulator as well.

It remains to show how to conceal the contents of the array S. To conceal S, we take the same v random bytes and use them on S in much the same way as we did for W. Using a different cryptographically secure pseudo-random number generator, we proceed as follows. We generate a stream of length $b * m$ bytes and EXOR it with S. We can do so, since we will be overwriting the element currently pointed to in S anyway. We then proceed as usual by overwriting the value currently pointed to in S with the encryption of W, if necessary. We thus distribute the v random bytes over the array S as well as over W.

The random choices for the probabilistic encryptions are thus critical in decrypting the gathered data. Unlike before, they cannot simply be discarded. The pseudo-random key streams need to be reconstructed during decryption and EXORed with the ciphertexts in order to decrypt.

Note that if an agent moves from A to B, then both count and ptr will have increased by 1. Count reveals exactly how many elements of W are currently in use, and ptr reveals the next array element in S to overwrite. The values for count and ptr can be used to help identify the order in which nodes are traversed. It seems that there is no easy way to avoid revealing such information when we require maintaining all information in order, unless "generalized secure computation" is made possible. To reduce the problem we may "randomize" the task of the agents. Namely, we replicate agents in nodes and add dummy encryptions and 'detours' as described in [GT96], allowing also the nodes along the detour to encrypt "dummy" data elements randomly – thus making different copies look differently and forcing tracing adversaries to follow the detours.

6 Conclusion

We have shown a new mode of operation of deterministic public key cryptosystems. This sliding encryption technique can be used to encipher small amounts of data, yielding small amounts of ciphertext, while insuring that the data remains secure. This technique was shown to be applicable in smart card applications with limited storage capacity. We also showed how to make tracing of an information gathering agent somewhat difficult, by further randomizing its state and task.

References

[BR94] M. Bellare, P. Rogaway, Optimal Asymmetric Encryption, Eurocrypt 94.

[Cha81] D. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. In *Communications of the ACM*, v. 24, n. 2, Feb 1981, pages 84–88.

[Ch96] F. Cheong. Internet Agents: Spiders, Wanderers, Brokers, and Bots. New Riders Publishing, page 5, 1996.

[De83] D. Denning. Cryptography and Data Security, Addison-Wesley, page 137, 1983.

[El85] T. ElGamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology—CRYPTO '84*, Springer-Verlag, pages 10–18, 1985.

[GM84] S. Goldwasser, A. Micali. Probabilistic Encryption. In *Journal of Computer and Systems Science*, v. 28, pages 270–299, 1984.

[GT96] C. Gulcu, G. Tsudik. Mixing Email with BABEL. In *Proceedings of the 1996 Symp. on Network and Distributed System Security* ISOC, pages 2-16, 1996.

[MB95] D. Mitchell, M. Blaze. truerand.c, AT&T Laboratories, 1995.

[Mc89] J. McAfee. Computer Viruses, Worms, Data Diddlers, Killer Programs, and other Threats to Your System. St. Martin's Press, page 29, 1989.

[RSA78] R. Rivest, A. Shamir, L. Adleman. A method for obtaining Digital Signatures and Public-Key Cryptosystems. In *Communications of the ACM*, v. 21, n. 2, pages 120–126, 1978.

[Sl94] R. Slade. Robert Slade's Guide to Computer Viruses. Springer-Verlag, page 49, 1994.

[Wh89] S.R. White, Covert Distributed Processing with Computer Viruses. In *Proceedings of the Crypto 89*, pages 616-619.

[WN94] D. Wheeler, R. Needham. Tiny Encryption Algorithm (TEA). In *Fast Software Encryption: second international workshop*, volume 1008 of Lecture Notes in computer science, Dec. 1994. Springer.

[YY96] A. Young, M. Yung. Cryptovirology: Extortion-Based Security Threats and Countermeasures. In *Proceedings of the 1996 IEEE Symp. on Security and Privacy*, IEEE Computer Society Press, pages 129–140, 1996.

[YuYu96] X. Yu, M. Yung. Agent Rendezvous: A Dynamic Symmetry-Breaking Problem. In *Proceedings of the 1996 ICALP*. Lecture Notes in Computer Science, Springer, July 1996.

A Cryptographic Mobile Agents

Currently, Cryptographic Application Programming Interfaces (CAPI) are available or in the process of being developed on many different computing platforms. Mobile agents for such platforms need not have their own multi-precision code and can utilize the node. However, there is evidence that such code can be implemented in compact form [YY96]. They compiled the portions of the GNU MP library that are needed to do 512 bit RSA encryptions with a public exponent of 3. The routines were compiled and run on a Macintosh SE/30. They made two notable optimizations.

1. First, they computed the reciprocal of the originator's public RSA modulus n, and included it within the agent. This allowed them to avoid including a MP division routine in the agent.
2. By choosing the public exponent of 3, a MP modular exponentiation routine was not needed since only two MP multiplications are required to encrypt.

The size of the compiled library, with C code and in-line assembly, is 4,372 bytes (cutting 70% of the general library). Note that using assembly language, it is of course possible to make this even smaller. Their agent also contained the TEA block cipher [WN94] in assembly, and it occupied a mere 88 bytes. TEA encrypted at a rate of 47k bytes/sec and can be employed for pseudorandom generation and hashing mentioned in this work, as well as for hybrid encryption. Thus, there is evidence to suggest that even when the agent has to implement the cryptographic procedures internally, the tools described in this paper can be implemented using relatively small space.