Reducing the Cost of Security in Link-State Routing

Ralf Hauser*
McKinsey Consulting
Zürich, Switzerland
hauser@acm.org

Tony Przygienda
Fore Systems
Bethesda, MD
prz@fore.com

Gene Tsudik†
USC Information Sciences Institute
Marina Del Rey, CA
gts@isi.edu

Abstract

Security in link state routing protocols is a feature that is both desirable and costly. This paper examines the cost of security and presents two techniques for efficient and secure processing of link state updates. The first technique is geared towards a relatively stable internetwork environment while the second is designed with a more volatile environment in mind.

1 Introduction

It is hardly necessary to motivate the need for security in a large, distributed internetwork environment. Since routing is a critical internetwork function, security of routing protocols is of the utmost importance. This has been widely recognized by the designers of many existing routing protocols. Sound security, however, often comes at a high price. Naturally, it is desirable to minimize the associated costs.

This paper concentrates on reducing the costs of security in link state routing protocols. We begin with a brief discussion of current security methods (Section 2) and an overview of the general security requirements (Section 3). After a brief description of the necessary building blocks (Section 4), the two techniques are presented in Sections 5 and 6, respectively.

2 Secure Link-State Routing

Network routing has received a lot of attention in the last three decades as evidenced by the enormous amount of literature in the field. Dijkstra's Shortest Path [10] and Ford/Fulkerson's Max Flow algorithms [9] gave rise to *link state* and *distance vector* routing protocols, respectively. In distance vector protocols, nodes keep tables of the best paths and associated metrics for all possible destinations and periodically exchange the contents of the tables with neighbors. In contrast, link state protocols are characterized by every node keeping a "map" of the entire network which is used to compute shortest paths to all destinations. Each node contributes to this global view by periodically distributing (via flooding) link state updates (LSUs); an LSU reflects the current status of all incident links of a given node.

The *state-of-the-art* in link state routing security is exemplified by *Network-layer Protocol with Byzantine Robustness* (NPBR) proposed by Perlman [6]. NPBR comes in two flavors: flooding and link state. Flooding NPBR is a highly-robust protocol where communication between any two nodes is guaranteed as long as there exists a non-faulty path between them. This robustness is achieved at the expense of: (1) flooding data packets, (2) per-packet public key encryption at every hop, and, (3) significant state in routers. The value of this protocol is largely theoretical as it illustrates the limits of achievable network-layer robustness and security.

Link state NPBR is slightly less robust. It guarantees communication between two nodes as long as there exist n node-disjoint paths between a given pair of nodes, and at most *(n-1)* node failures exist simultaneously in the network. Reduced robustness in link state

*Names appear in alphabetical order.

†Contact author.

0

NPBR is counter-balanced by the use of link state routing in conjunction with source routing. LSU dissemination is performed using highly-robust flooding NPBR method, whereas data packets are source-routed. This design, while still quite costly, demonstrates some very useful techniques:

- Public key signatures for link state updates to defend against tampering and repudiation of origin.
- Per-node nonwrapping sequence numbers to prevent replay detection and packet reordering.
- End-to-end and hop-by-hop packet delivery acknowledgements to determine dynamically the status of links and nodes traversed by a packet.

Nowadays, designers of link-state routing protocols have little choice but to consider similar techniques, however costly. (The use of digital signatures in OSPF [22, 3] is a case in point.) While public key technology is still lagging in terms of performance the only alternative – conventional, shared-key cryptography – is awkward and unsuitable for link state routing.[1]

For routers, the bulk of security-related activity is spent in composing and processing LSU-s which translates into generating and verifying public key signatures, respectively. Our goal in this paper is to come up with more efficient techniques for the these tasks.

3 Requirements

Before attempting to construct solutions we briefly examine the requirements for LSU security:

1. Origin Authentication
2. Non-repudiation[2]
3. Data integrity
4. Timeliness and Ordering

The first two go hand-in-hand: *every LSU must be unambiguously associated with, and, traceable to, the router that generated it.* Data integrity is required to detect any tampering with the LSU-s. Finally, timeliness and ordering are necessary to prevent stale LSU-s from propagating.

[1]There are essentially two options with conventional cryptography: i) a single globally-shared key used to "sign" all LSUs, or, ii) pairwise keys. The former provides a very low level of security while the latter results in key *explosion* as every router has to maintain n keys. The only other alternative is hop-by-hop LSU re-authentication; it is discussed in Section 3.

[2]Non-repudiation is not always considered a primary requirement in routing security.

Data secrecy is conspicously absent from the list. There is no inherent reason for this other than *folklore.* Routing information (network topology) is not usually treated as sensitive. Where the need may arise, a group of routers can share a secret key to be used for LSU encryption. However, this approach does not scale. Alternatively, LSUs can be encrypted on a hop-by-hop basis. This method scales better since only adjacent routers need to share keys, but the overhead can be prohibitive for a network with a large diameter (encrypt/decrypt at every hop.)

4 Building Blocks

We use two basic tools in our construction of secure LSU distribution: public key-based digital signatures and one-way hash functions. Signatures can be computed using any well-known public key cryptosystem: RSA, DSS, El Gamal, Schnorr, etc. A number of (conjectured) strong one-way functions have been proposed, e.g., MD4, MD5, 8-pass SNEFRU and SHA [2, 1, 8, 17].

We also take advantage of a simple and elegant technique of constructing hash chains similar to that utilized in S/KEY one-time authentication [20]. Originally developed by Merkle, Lamport [18], and Winternitz, it is currently being used in a number of application domains, such as micro-payments [15], conditional anonymity [14], non-repudiation [13] and certificate revocation [16].

In brief, hash chains are utilized as follows:

A party (Alice) generates a secret quantity R and computes a hash chain of length n:

$$\mathcal{H}^1(R),\ ...,\ \mathcal{H}^i(R),\ ...\ \mathcal{H}^n(R)$$

where $\mathcal{H}^0(R) = R$ and $\mathcal{H}^i(R) = \mathcal{H}(\mathcal{H}^{i-1}(R))$ for $0 < i < n$ and the hash function itself – $\mathcal{H}()$ – can be, for example, MD5 or SHA.

We assume that, initially, the verifyer (Bob) receives $\mathcal{H}^n(R)$ and is assured of its genuineness. When Alice wants to authenticate herself to Bob, she releases $\mathcal{H}^{n-1}(R)$ and Bob simply checks that $\mathcal{H}(\mathcal{H}^{n-1}(R))$ matches $\mathcal{H}^n(R)$. This assures Bob that only Alice could have produced $\mathcal{H}^{n-1}(R)$. This procedure (i.e., one-time authentication) can be repeated n times – once for each step *in the hash chain.*

5 Stable Link State

In this section we describe a simple technique for reducing cryptographic costs associated with LSU processing. This scheme, referred to as SLS (Stable Link State), is effective when network links and nodes are fairly stable and LSUs are triggered either by time or by explicit requests.

It has been observed that a large percentage (50%, by some estimates) of LSU-s are simply re-statements of previous LSU-s. In other words, an LSU often carries no new information other than its timing since links and nodes go up and down infrequently.

We assume that, whenever a link's state changes, the incident nodes will each generate a fresh, signed LSU. Such an LSU is called ALSU.[3] Each ALSU must be sequenced, timestamped and digitally signed by its originator thus allowing each receiving node to establish unambiguously the origin and the data integrity of the ALSU. Signing is done with the node's private half of the public key-pair in a manner similar to Perlman's NBPR.

In addition to the digital signature and all the usual information carried in an LSU, an ALSU carries an *anchor* value and an index. We denote the anchor value as as $\mathcal{H}^n(R)$ and the index as n. As mentioned in the previous section, $\mathcal{H}^n$ represents n successive computations of a strong one-way hash function $\mathcal{H}$ starting with a unique, randomly-chosen quantity R. For example, $\mathcal{H}^{10}(R)$ denotes 10 computations starting with R.

Having successfully verified the signature, each receiver stores the anchor value along with the rest of a ALSU. Subsequently, if the originating node needs to send out a new LSU (because, the old one expired or because it is programmed to do so every so often) it releases $\mathcal{H}^{n-1}(R)$ as the new LSU which we refer to as $CLSU_1$.[4] Every receiver verifies $CLSU_1$ by checking that:

$$\mathcal{H}(\mathcal{H}^{n-1}(R)) = \mathcal{H}^n(R)$$

As long as no link adjacent to the originator changes state, any subsequent $CLSU_i$ can be represented as $\mathcal{H}^{n-i}(R)$ where i is the number of updates after the original signed update ALSU that included the anchor value.

With the present method, no expensive signature computations or signature verifications are necessary for the construction and dissemination of LSU-s as long as the state of the links adjacent to the originator stays unchanged. The cost of generating a CLSU is negligible if the originator stores all $\mathcal{H}^{n-i}(R)$ values. If the originator chooses to store only the anchor value $R = \mathcal{H}^0(R)$ (in order to conserve space), the cost of generating a CLSU amounts to $n - i$ hash function computations. The cost of CLSU verification is one hash function computation: checking that $\mathcal{H}(\mathcal{H}^{n-i}(R)) = \mathcal{H}^{n-i-1}(R)$.

[3] "A" for *anchor.*

[4] "C" for *chained.*

5.1 Timeliness/Freshness

The SLS method, as described thus far, does not provide timeliness of the CLSU-s. (This is different from CLSU *sequencing* which is given by the nature of the hash chain construction.) Timeliness can be obtained by incorporating timestamps into the hash chain explicitly, e.g., by setting:

$$\mathcal{H}^i = \mathcal{H}(T_i, \mathcal{H}^{i-1})$$

where T_i is the timestamp of the i-th step in the hash chain. (Note that $T_i > T_{i+1}$ for $0 \leq \; i \leq \; n$.) The interval between successive timed updates, $t = (T_i - T_{i+1})$, can be selected as either a system-wide or node-specific, protocol parameter. (In the latter case the t must be included in each ALSU.)

As an aside, we note that if each ALSU is timestamped and the inter-LSU interval is established (either as a system-wide parameter or as part of ALSU) it is not absolutely necessary to include timestamps in the hash chain computation.[5]

5.2 Missing CLSU-s

If a node misses some (say, j) CLSU-s from a given originator, it can easily catch up. Suppose that the last CLSU received is $CLSU_i = \mathcal{H}^{n-i}(R)$ and it now receives $CLSU_k = \mathcal{H}^{(n-k)}(R)$ (where $n > k > i \; and \; j = k - i$.) Then, the receiving node simply computes $\mathcal{H}^j(\mathcal{H}^{(n-k)}(R))$ and checks that it equals $\mathcal{H}^{n-i}(R)$.

It is also possible that a node misses an ALSU which contains the anchor value and the "real" signature. In order to accommodate this case, each $CLSU_i$ can be distributed with a copy of its "ancestor" ALSU. This way, a receiving node can re-initialize by verifying the signature on the old ALSU, obtaining $\mathcal{H}^n(R)$ and then verifying $CLSU_i$ based on the anchor value as described above. (Catching up would require i hash function computations.) Alternatively, a node can request a copy of a missing ALSU from its neighbors.

5.3 Storage Requirements

Storage requirements are not much greater than in a normal link state environment. A router needs to store the latest ALSU for each one of its peer routers. The only additional burden is having to store the latest CLSU alongside each ALSU. (A typical CLSU contains little more than a timestamp T_i and $\mathcal{H}^{(i)}(R)$.)

[5] Since the appropriate time for a $CLSU_i$ is given as the sum of ALSU timestamp and $(t * i)$.

6 Fluctuating Link State

The SLS scheme is not very effective if links and nodes fluctuate. However, it is precisely under such conditions when the cost of processing LSUs is particularly felt. In this section we outline a technique called FLS (Fluctuating Link State) aimed at a more dynamic routing environment.

The basic technique in FLS is similar to that in SLS. However, instead of one, each router generates $(n \times\ k \times\ 2)$ distinct hash chains forming a *hash table.* (Where n is the number of steps in the chain and k is the number of incident links.) For each link two hash chains are generated: one for UP state and one for $DOWN$ state. All $DOWN$ chains are generated with a hash function $\mathcal{G}$ and all UP chains – with $\mathcal{H}$. ($\mathcal{G}$ and $\mathcal{H}$ must be distinct but need to have the same security properties. An alternative is to use the same hash function for both UP and $DOWN$ chains, but, with *different* starting values.)

All starting values: R_j $(1 \leq\ j \leq\ k)$ need to be unique, although they can all can be generated from a single common R, e.g., $R_j = \mathcal{F}(j, nodeID, R)$ where $\mathcal{F}$ is some (not necessarily one-way or hash) function and $(j, nodeID)$ uniquely identifies the link/node pair.

The layout of a hash table is illustrated in Table 1. Upon its computation, each originating node composes an *anchor* LSU (ALSU). It contains a set of signed anchor values taken from the bottommost row of the hash table:
$[nodeID, T_n, \mathcal{H}^n(R_1), \mathcal{G}^n(R_1), ...,\ \ \mathcal{H}^n(R_j), \mathcal{G}^n(R_j), ..., \mathcal{H}^n(R_k), \mathcal{G}^n(R_k)]^{SK}$
where T_n is the current time and SK is the node's secret key.

Upon receipt of an ALSU from a peer, a router first checks the signature computed over the anchor values. If the signature checks out and the timestamp T_n is considered fresh, the entire ALSU is stored.

Subsequently, when it is time for an update (either because of time or a change in some link's state), the originating node composes a $CLSU_i$ (i is index of the update or its distance from ALSU.)

For each link L_j, $(1 \leq\ j \leq\ k)$ and for each $CLSU_i$, $(1 \leq\ i <\ n)$ link state flags (LSF_i) is defined as:

$$LSF_i = [LF_i(1), ..., LF_i(k)]$$

such that:

$$LF_i(j) = \begin{cases} 1 & \text{if } L_j \text{ is UP} \\ 0 & \text{if } L_j \text{ is DOWN} \end{cases}$$

Similarly, for each link L_j, $(1 \leq\ j \leq\ k)$ and for each $CLSU_i$, $(1 \leq\ i <\ n)$ link state vector (LSV_i) is defined as:

$$LSV_i = [LS_i(1), ..., LS_i(k)]$$

where:

$$LS_i(j) = \begin{cases} \mathcal{H}^{n-i}(R_j) & \text{if } LF_i(j) = 1 \\ \mathcal{G}^{n-i}(R_j) & \text{if } LF_i(j) = 0 \end{cases}$$

The i-th CLSU following the ALSU contains $[nodeID, i, T_i, LSF_i, LSV_i]$. Every receiving node is assumed to have in its possession an earlier CLSU, $CLSU_p$. In most cases $p = i - 1$ which means that the receiving node has not missed any CLSUs since $CLSU_p$. The case of $i - p > 1$ indicates that the receiving node missed $(i - p - 1)$ CLSUs and, if $i = p$, $CLSU_p$ is a duplicate.

A receiving node processes $CLSU_i$ in the following manner:[6]

1. Looks up the current entry for *nodeID*

2. Validates T_i and i:

   Checks that T_i is reasonably close to current time, $i > p$ and $T_i > T_p$ (last stored timestamp from $CLSU_p$.)

3. For each link L_j reflected in $CLSU_i$ $(0 < j < k)$:

   a) if state unchanged $(LF_i(j) = LF_p(j))$, compute:

   $$\begin{array}{ll} \mathcal{G}^{i-p}(LS_i(j)) & \text{if } LF_i(j) = 0 \\ \mathcal{H}^{i-p}(LS_i(j)) & \text{if } LF_i(j) = 1 \end{array}$$

   and compare to $LS_p(j)$; reject upon mismatch.

   b) if state changed $(LF_i(j) \neq LF_p(j))$, compute:

   $$\begin{array}{ll} \mathcal{G}^{i}(\mathcal{G}^{n-i}(R_j)) & \text{if } LF_i(j) = 0 \\ \mathcal{H}^{i}(\mathcal{H}^{n-i}(R_j)) & \text{if } LF_i(j) = 1 \end{array}$$

   and compare to $LS_n(j)$; reject upon mismatch.

After the entire LSV_i is authenticated, the previous link state vector (LSV_p) is replaced by LSV_i.

Remark: In the event that the incoming $CLSU_i$ indicates a state change for a link L_j, the process of validating $LS_i(j)$ can be optimized. The description above (Step 3b) involves i hash function operations to compute either $\mathcal{G}^i(\mathcal{G}^{n-i}(R_j))$ or $\mathcal{H}^i(\mathcal{H}^{n-i}(R_j))$, i.e., the computation starts with the anchor value. This can be improved if, for each L_j, a receiving node keeps not only the last $LF_i(j), LS_i(j)$ pair but also the pair $LF_q(j), LS_q(j)$ where $LF_q(j) \neq LF_i(j)$, $q < i$ and, for all s, $(q < s < i) LF_s(j) = LF_i(j)$. In other words, $LF_q(j), LS_q(j)$ represent the state of the link L_j before the next-to-last link state change occurred. This time/space trade-off saves $(i - q)$ hash function operations for each link that had a state change.

[6] As in SLS, $CLSU_i$ can optionally include the set of current anchor values in order to help routers that either lost state or somehow missed the original ALSU.

	L_1		…	L_j		…	L_k	
	up	down	…	up	down	…	up	down
1	$\mathcal{H}^1(R_1)$	$\mathcal{G}^1(R_1)$	…	$\mathcal{H}^1(R_j)$	$\mathcal{G}^1(R_j)$	…	$\mathcal{H}^1(R_k)$	$\mathcal{G}^1(R_k)$
.	.	.	…	.	.	…	.	.
.	.	.	…	.	.	…	.	.
.	.	.	…	.	.	…	.	.
i	$\mathcal{H}^i(R_1)$	$\mathcal{G}^i(R_1)$	…	$\mathcal{H}^i(R_j)$	$\mathcal{G}^i(R_j)$	…	$\mathcal{H}^i(R_k)$	$\mathcal{G}^i(R_k)$
.	.	.	…	.	.	…	.	.
.	.	.	…	.	.	…	.	.
.	.	.	…	.	.	…	.	.
n	$\mathcal{H}^n(R_1)$	$\mathcal{G}^n(R_1)$	…	$\mathcal{H}^n(R_j)$	$\mathcal{G}^n(R_j)$	…	$\mathcal{H}^n(R_k)$	$\mathcal{G}^n(R_k)$

Table 1. Hash table computed by a router

6.1 Security

The security of both SLS and FLS is dependent on four factors:

1. Strength of the underlying signature function (used to sign ALSUs)
2. Strength of the underlying hash function
3. Randomness of the starting values
4. Loose clock synchronization

The first three are not specific to techniques described above. We assume that tools such as RSA or DSS (for signatures), SHA or MD5 (for one-way hash function) and pseudo-random number generators[7] will be used. Clock synchronization is briefly addressed below. (It is important to note that this requirement is not unique to our approach; any non-interactive scheme for secure LSU distribution would need loose clock synchronization.)

6.2 Clock Synchronization

Loose clock synchronization is necessary since time is one of the inputs to the hash function in each hash chain computation. Every step in the hash chain (in SLS) or row in the hash table (in FLS) corresponds to a specific time interval. The maximum clock drift between any two nodes is $(2 \times t)$ where t is the time between successive CLSUs. Hash table entries that are not distributed as part of some CLSU simply expire. In the following example $\mathcal{G}^{i+1}(R_j)$ expires:

1. Suppose that at time T_i a router distributes $CLSU_i$ such that for some incident link L_j: $LS_i(j) = 1, \mathcal{H}^{n-i}(R_j)$

[7] See Internet RFC 1750 for details.[21]

2. Then, a time T_{i+1}, L_j goes down and the router distributes $LS_{i+1}(j) = 0, \mathcal{G}^{n-i-1}(R_j)$

An observer who records $CLSU_i$ and $CLSU_{i+1}$ learns $\mathcal{H}^{n-i}(R_j)$ and $\mathcal{G}^{n-i-1}(R_j)$, respectively. However, since at time T_i link L_j was UP, $\mathcal{G}^{n-i}(R_j)$ was never distributed. But, it can be easily computed as $\mathcal{G}(\mathcal{G}^{n-i-1}(R_j))$. If our observer is malicious, he might be tempted to create some confusion by distributing a forged copy of $CLSU_i$ containing: $LF_i(j) = 0 \ and \ LS_i(j) = \mathcal{G}^{n-i}(R_j)$
Surprisingly, this would not cause any disruption since:

- most receiving nodes would consider the timestamp T_i expired
- most nodes already received a genuine copy of $CLSU_i$ and would ignore the forgery

  and:

- even if a node's clock is off by at most $(2 \times t)$ and it has not received the genuine $CLSU_i$, the "current" status (DOWN) being communicated by the forged LSU is actually correct!

It is important to note that allowing clock drift greater than $(2 \times t)$ is insecure as the following example demonstrates:

1. As before, at time T_i, $CLSU_i$ contains: $LS_i(j) = 1, \mathcal{H}^{n-i}(R_j)$
2. Then, at time T_{i+1}, $CLSU_(i+1)$ reflects a status change: $LS_{i+1}(j) = 0, \mathcal{G}^{n-i-1}(R_j)$
3. Finally, at time T_{i+2}, $CLSU_(i+2)$ is distributed where $LS_{i+2}(j) = 1, \mathcal{H}^{n-i-2}(R_j)$

In summary, the state history of link L_j is: UP, DOWN, UP.

We further assume that there exists at least one node with a clock drift of at least $(3 \times t)$ with respect to the CLSU originator.

A malicious observer records all three CLSUs and, as in the previous example, computes $\mathcal{G}(\mathcal{G}^{n-i-1}(R_j))$. He then distributes a forged copy of $CLSU_i$ containing:

$$LS_i(j) \ = \ 0, \mathcal{G}^{n-i}(R_j)$$

This time, the forged $CLSU_i$ can cause disruption since a node with a clock drift of at least $(3 \times t)$ will accept the forgery as both timely and genuine. This would result in the link L_j marked as DOWN while it is actually UP.

6.3 Limitations

Although the FLS method is designed as a general solution to LSU distribution, there are certain conditions that can make it impractical or, at least, unappealing:

- Very frequent state oscillations
  If the environment is such that link and node outages occur very often, FLS becomes unworkable since hash chains are computed to take into account pre-set time intervals.

- No clock synchronization
  Obviously, if routers do not synchronize clocks (even loosely), freshness of CLSUs can not be established. Moreover, attacks of the type discussed in section 6.1 become possible.

- Multiple-valued (or continuous) link state
  So far, we have made an assumption of the link state being binary: UP or DOWN. While this is often the case, routing protocols sometimes express link state as a function of available bandwidth. For example, link state may be expressed as a percentage of the link's current capacity, i.e., anywhere between 0 and 100%. In order to support this type of *multiple-valued* link state, each node's hash table has to grow by a factor of m – the set of all possible values of the link state. In other words, each node will need to make $(n \times \ k \times \ m)$ hash function computations as part of building its hash table. Also, taking into account the increased storage costs, it may be more efficient to resort to full-blown digital signatures.

7 Future Work

While both FLS and SLS methods presented in this paper appear attractive in some respects, the magnitude of their efficiency (as compared to fully-signed LSUs *a la* [6]) remains to be validated by experiments with actual routing protocols; for example, extending the work of Murphy and Badger [22] on digital signatures in OSPF [3].

Another potential item of interest is the combination of SLS and FLS into a unified technique.

8 Summary

In conclusion, this paper presented two techniques for efficient and secure generation and processing of updates in link state routing protocols. The first technique (SLS) is geared towards relatively stable network environments where node and link outages are infrequent. The second, more involved, technique (FLS) is designed with more volatile environments in mind. However, neither technique is particularly suitable for the worst-case scenario of arbitrary failures. Work is on-going to determine how to best merge traditional solutions (such as Perlman's NBPR) with SLS and FLS to achieve the optimal balance between security, runtime performance and storage requirements.

9 A Word of Thanks

The authors would like to acknowledge Michael Steiner, Hilarie Orman, Kannan Varadhan, Michael Waidner and clandestine referees for their valuable comments on the earlier drafts of this paper.

References

[1] R. Rivest, *The MD5 Message-Digest Algorithm*, RFC 1321, April 1992.

[2] R. Rivest, *The MD4 Message-Digest Algorithm*, RFC 1320, April 1992.

[3] J. Moy, *OSPF Version 2*, RFC 1583, March 1994.

[4] M. Steenstrup, *IDPR as a Proposed Standard*, Internet RFC 1477, July 1993.

[5] M. Steenstrup, *An Architecture for Inter-Domain Policy Routing*, Internet RFC 1478, June 1993.

[6] R. Perlman, *Network Layer Protocols with Byzantine Robustness*, Ph.D. Dissertation, MIT LCS TR-429, October 1988.

[7] R. Rivest, A. Shamir and L. Adleman, *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*, Communications of the ACM, February 1978.

[8] R. Merkle, *One-way Hash functions and DES*, Proceedings of CRYPTO 89, August 1989.

[9] L. Ford and D. Fulkerson, *Flows in Networks*, Princeton, NJ:Princeton University Press, 1962.

[10] E. Dijkstra, *Self-Stabilization in Spite of Distributed Control*, Communications of the ACM, November 1974.

[11] T. El Gamal, *A Public Key Cryptosystem and a Signature Based on Discrete Logarithms*, IEEE Transaction on Information Theory, Vol. 31, July 1985.

[12] W. Diffie and M. Hellman, *New Directions in Cryptography*, IEEE Transactions on Information Theory, Vol. 22, No. 11, November, 1976.

[13] N. Asokan, G. Tsudik and M. Waidner *Server-Supported Signatures*, 1996 European Symposium on Research in Computer Security, September 1996.

[14] R. Hauser, G. Tsudik *On Shopping* Incognito, 2nd USENIX Workshop on Electronic Commerce, November 1996.

[15] R. Hauser, M. Steiner and M. Waidner, *Micropayments based on* iKP, SECURICOM'96, May 1996. (Also available as IBM Research Report.)

[16] S. Micali, *Enhanced Certificate Revocation System*, Technical Memo MIT/LCS/TM-542, November 1995.

[17] U. S. National Institute of Standards and Technology (NIST), *Secure Hash Standard.* Federal Information Processing Standards Publication (FIPS PUB) 180, May 1993.

[18] L. Lamport, *Password Authentication with Insecure Communication*, *Communications of the ACM*, 24(11):770–772, November 1981.

[19] B. Kumar, *Integration of Security in Network Routing Protocols*, SIGSAC Reviews, 11(2):18–25, 1993.

[20] N. Haller, *The S/Key One-Time Password System*, ISOC Symposium on Network and Distributed Systems Security, February 1994.

[21] D. Eastlake, S. Crocker and J. Schiller, *Randomness Recommendations for Security*, Internet RFC 1750, December 1994.

[22] S. Murphy and M. Badger, *Digital Signature Protection of the OSPF Routing Protocol*, ISOC Symposium on Network and Distributed Systems Security, February 1996.