Tracing Faults in MANETs Using Symmetric Authentication Chains*

Eric Hokanson, *Department of Computer Science, Florida State University, Tallahassee, Florida 32306, USA*

***Abstract*—Before Mobile Ad Hoc Networks (MANETs) become a mainstay in computing applications, many security issues need to be addressed. One such issue is the insider threat. How does one find and effectively neutralize a malicious node? This paper outlines a plan for improving a proposed Byzantine tracing algorithm by using keyed message authentication codes (HMACs) in place of digital signatures. An HMAC is computationally more efficient and therefore less of a strain on a node's limited resources. Tracing algorithms allow intermediate nodes to be more proactive in finding and hopefully neutralizing malicious nodes.**

***Index Terms*—Ad hoc networks, Byzantine faults, fault-tracing.**

I. INTRODUCTION

Technological advances have brought ad hoc networks closer to reality and next generation computing applications are expected to rely heavily on this new type of network infrastructure. Unfortunately, before ad hoc networks can be successfully deployed, many security issues must be addressed. One such issue includes insider threats. A malicious node on a routing path may try to redirect packets, perform a denial of service attack by engaging a node in resource consuming activities such as routing packets in a loop, or may simply just drop the packet altogether. The problem: how does one find such malicious faults and then how does one effectively neutralize them?

Tracing malicious (insider) nodes in ad hoc networks is not as easy it appears. One such algorithm proposed in [1] is clever but computationally expensive; finding the fault in log(n) time. Additionally, the algorithm uses past performance to find the insider and assumes that nodes do not conspire. This Bayesian approach will not work with nodes that exhibit Byzantine behavior. The tracing algorithms in [3], [4] address the weaknesses of [1] much more efficiently thus making it hard for a malicious insider to avoid detection by acting non-maliciously when the insider "knows" it is being investigated.

The goal of this paper is to improve the performance of the tracing algorithm presented in [3] by using a keyed message authentication code (HMAC) instead of a digital signature scheme. In particular, we wish to emulate asymmetric authentication in a purely symmetric setting similar in scope to [11] but at a cheaper cost. Symmetric key authentication is computationally more efficient than digital signing and therefore less of a drain on a node's resources. Some public key systems may require longer key lengths than symmetric systems for an equivalent level of security (e.g. see [12]) thereby making the computation expensive by an order of ten times. Weaker sensor devices that can only use hash functions but no encryption are not able to take advantage of the optimal tracing algorithm in [3]. Never the less, we show that weaker ad hoc devices may still employ a Byzantine tracing program using symmetric authentication chains.

II. A PROPOSED FAULT TRACER USING SYMMETRIC AUTHENTICATION CHAIN

We propose to build on the optimized Byzantine tracing algorithm outlined in [3] by replacing the digital signatures of the packets, *acks* (acknowledgements) and the *frpts* (fault reports) with HMACs [2] in the following way: the source will send a packet (pkt_{sd}) authenticated with a keyed message authentication code, using a shared key between the source and the destination. It is assumed a routing algorithm such as [7] or [10] is used, but most routing algorithms used in ad hoc networking can be extended to use our proposed scheme. Furthermore, this tracer is used during the communication phase over an established path (established during the route discovery phase) such that two honest nodes adjacent to one another are neighbors. There is no such assumption for dishonest nodes.

The Destination, when receiving the packet from source *s*, constructs an acknowledgement (ack_{sd}) also using HMACs. The ack_{sd} contains important information regarding the transaction between the source and destination including a session number and the hash of the packet. The destination then sends ack_{sd} to the intermediate nodes for delivery back to the source.

The intermediate node(s), in addition to forwarding packets from source to destination, keep a record of the contents of pkt_{sd} and ack_{sd} (their hashed values) for future validations in the event of a fault.

If the source's timer expires before receiving a valid ack_{sd}, then the source constructs a probe to find the origin of the fault. This probe is *chain authenticated* and contains a payload identifying the correct pkt_{sd} and ack_{sd}, which the

* This material is based on work partly supported by the NSF grant DUE 0243117.

intermediate nodes use to compare with their record of the transaction between the source and destination. The aim of the probe is to hone in on the guilty party and set off a chain of events whereby the nodes on the path will issue a fault report (*frpt*), which is also chain authenticated in a similar fashion as the probe, naming the malicious node. How the probe and the chain authentication scheme works will be described in the next section.

III. THE OPTIMISTIC SYMMETRIC KEY TRACING ALGORITHM

In this section we define and discuss more precisely how the Optimistic Symmetric Key Tracing (O-SKT) algorithm works. Before doing so, it is important to note that when a node has been located and reported as malicious by another node, it is not possible to tell which node is actually faulty. In our tracer, each time a node is reported as malicious, both the reporting node and the reported are treated as malicious and eliminated. In the worst case, two innocent nodes are sacrificed for every traitor; otherwise one non-faulty node is forfeit for one faulty node.

The adversary can redirect, drop, corrupt or inject packets into a MANET sending them to any node in the network $G = (V, E)$. He may eavesdrop on all communications no matter the network structure; he may even request any node under his control to perform any action. The adversary has complete knowledge of the MANET making her more powerful than any other adversary models for ad hoc networks, including the Byzantine adversarial model [8]. The adversarial threats model assumed in our proposed solution is based on [6]. For this, the adversary can corrupt any node set belonging to a set $\Gamma \subseteq 2^V$, that is monotonic, i.e., for which, if $X \in \Gamma$ and $X' \subset X$ then $X' \in \Gamma$. Γ is called an *adversary structure*. This model clearly extends the Byzantine threats model for which $\Gamma = \{X \subset V : |X| \leq k\}$.

A. Formal Definition of the O-SKT Protocol

The O-SKT is an optimistic[1] malicious node tracing program based on the work done in [3]. With this algorithm there is no additional cost when there are no faults. When a fault occurs, the cost to locate that fault is one tracing round and two *n*-chained hash message authentication codes (HMAC), where *n* is the number of symmetric keys. This process resembles onion layered chaining [13], however for HMACs we have chained authentication.

The same assumptions in [3] and [4] hold for this scheme, i.e., we assume that all HMACs are unforgeable and the adversary is polynomial bounded in the security parameter of the HMACs. We also assume that any security associations between the nodes have been established using some external Trusted Third Party (TTP) and that all keys required for our protocol have been distributed amongst the nodes. The network is subject to medium constraints such as weak synchrony (the time for a single transmission to be received is bounded by a constant) and promiscuity (a packet transmitted by a node will be received by all its neighbors). Additionally the source and destination are trusted, and the route generated by the route discovery phase of the routing protocol, is a sequence of nodes $s = x_1, x_2, \ldots, x_n = d$ for which x_i, x_{i+1} are neighbors if both are not faulty for $i = 0, 1, \ldots, n$. There are no assumptions for pairs x_j, x_{j+1} if one is faulty.

The impossibility of dealing with *man-in-the-middle relay attacks* (e.g. see invisible node attack [9]) during the route discovery phase of any routing algorithm is well known in the literature. To the best of our knowledge, this attack can only be addressed with out-of-system mechanisms such as temporal or location certification [3]. Determining if two adjacent nodes are real neighbors is not as simple as it appears and to date, there is no known route discovery algorithm that can guarantee immediate delivery of a packet in the presence of a general adversarial model [3].

Our proposed tracing program will only work during communications over an established route – routes for which honest nodes are actual neighbors. The following notation will be used:

$pkt_{sd} = [s, d, sn, seq_s, data\ [s, d, sn, seq_s, data]_{sd}]$, a packet that has been authenticated by s with an HMAC [2] using the shared secret key of s and d. The packet's payload contains identifiers s, d, a session number sn for the tracing algorithm (unique to each session), the sequence number seq_s for pkt_{sd}.

seq_s and $timer_{sd}$ are counters for s; *timeout* depends on the time taken for a round trip from s to d.

$ack_{sd} = [s, d, sn, seq_s\ [s, d, sn, seq_s]_{sd}]$, an acknowledgement of receipt of pkt_{sd} authenticated by d with an HMAC whose key is the shared secret key of s and d.

$probe_s$ is a chained HMAC probe using the keys $x_0 x_i$ that the source s (node x_0) shares with some i^{th} node (x_i) on route $s = x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_n = d$; it is defined recursively as:

$$H_s(z, 1) = z, h_{0,n}, h_{0,n-1}, \ldots, h_{0,1} \quad (1)$$
$$H_s(z, 2) = z, h_{0,n}, h_{0,n-1}, \ldots, h_{0,2} \quad (2)$$
$$\vdots$$
$$H_s(z, i) = z, h_{0,n}, h_{0,n-1}, \ldots, h_{0,i} \quad (i)$$
$$\vdots$$
$$H_s(z, n) = z, h_{0,n} \quad (n)$$

where z is the payload containing: (s, d, sn, seq_s, $hash(pkt_{sd})$, $hash(ack_{sd})$), and *hash* is a cryptographic hash, i.e. MD-5 or SHA-1 [12]. The probe is created by s, hashed in a chained manner as follows:

$h_{0,i} = h_{x_{0,i}}(z, h_{0,n}, \ldots, h_{0,i+1})$, $i = 1, \ldots, n-1$ where $h_{0,i}$ is an HMAC with the shared key of x_0 and x_i, and $h_{0,n} = h_{0,n}(z)$.

[1] Optimistic algorithms have optimal performance when there are no faults.

The authentication chain, $H_0(z, 1)$, is passed to node x_1 who will verify the probe, strip off tag $h_{0,1}$ and send the remaining chain (2) to node x_2 where the process repeats until the destination $d = x_n$ gets $H_0(z, n-1)$ and strips it off to get $H_0(z, n)$.

$frpt_y$ is a fault report created by some node y who observes a fault and is hashed in a similar recursive fashion as the probe: $H_y(Z, t) = Z, h_{0,y}, h_{1,y}, \ldots, h_{t,y}$ where z is the payload containing the identifiers: (s, d, y, $succ(y)$, $[x_i$ *or NULL]*, sn, seq_s); $succ(y)$ is the successor node blamed for failure by y. The *[x_i or NULL]* field is used in the event a node creates a malicious fault report to cause some other node further up stream to falsely accuse an innocent neighbor. A node who receives an invalid or corrupt fault report would create a new fault report naming the creator of the malicious fault report (x_i) in addition to itself and its successor.

$timer_{xy}$, bound on time taken for a packet going round trip from node x to y.

B. The O-SKT Protocol Description

In this section, we offer a simple example with several scenarios to illustrate how the O-SKT algorithm will work in a mobile ad hoc network and how it adapts to various situations when an adversary is present. Let *source = s, a, b, c, x, y, d = destination* be the path discovered during the route discovery phase of a routing protocol. Note that all non-faulty nodes in this path should be neighbors. We acknowledge that not all adversary nodes may be traceable; however, if the route discovery phase produces a legitimate route as described in the previous section, then our tracer will succeed in either sending a packet to a destination or trace at least one faulty node.

1) Case 1 -- Everyone follows the rules (The optimistic round): Source s sends a packet authenticated by using an HMAC with the symmetric key it shares with the destination d (pkt_{sd}) and sets its timer ($timer_{sd}$). The timer is set to allow for enough time for a round trip: the number of hops times an upper bound, τ, on the time allocated for each hop (figure 1). Each node along the path to d stores a record of the transaction, i.e. the hash of the contents of pkt_{sd}, and forwards pkt_{sd} along to its successor. These nodes only keep the records for a brief period of time, expunging them when either the communication phase between the source and destination have concluded successfully, or after a faulty node has been traced. When destination d successfully receives pkt_{sd}, it constructs ack_{sd} – an acknowledgement to send to s. Each intermediate node between s and d stores a hash of the ack_{sd} contents before passing ack_{sd} to its predecessor. The ack_{sd} makes it back to s before $timer_{sd}$ lapses, thus ensuring s its packet reached d without incident.

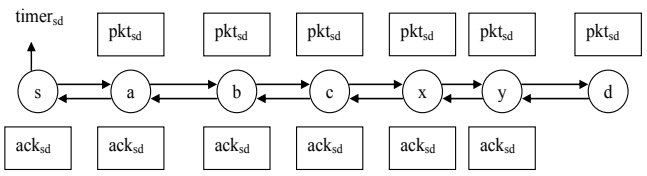

Fig. 1. A round of the Optimistic Symmetric Key Tracing algorithm when there are no faults. Source s sends a packet to destination d successfully. On receipt, destination d creates and returns an acknowledgement to s.

2) Case 2 -- Source receives neither a valid ack nor frpt from destination (The tracing round): source s sends an authenticated HMAC packet pkt_{sd} to destination d but along the way a malicious node (c) drops pkt_{sd} (figure 2) or corrupts it. The source's timer will expire since it will not receive a valid ack_{sd} from the destination d. As a result, s constructs $probe_s$ (the construction process is explained later in this section) with the payload [s, d, sn, seq_s, $hash(pkt_{sd})$, $hash(ack_{sd})$]. Note that the source has all the ingredients necessary to construct by itself the acknowledgement it should have received for the dropped packet.

Upon receipt of the probe, each intermediate node validates and compares the contents of the probe with its own record of pkt_{sd} and ack_{sd}. If pkt_{sd} matches but the ack_{sd} does not, which is the case in our example because pkt_{sd} was sent but no ack_{sd} was received, the intermediate node sets its timer (round-trip from its position on the route to the destination) and forwards the probe to its successor who repeats the validation and examination process. In this protocol the nodes set timers with decreasing expiration times so that a fault report is only issued by a non-faulty node when its successor is faulty.

When the probe is forwarded to a node that has no record of both pkt_{sd} and ack_{sd} (neither the packet nor the acknowledgement will match the contents of the probe), it does not pass the probe on and remains silent (see figure 2) causing an upstream node's timer to expire ($timer_{bd}$ in our example). Upon timeout, node b constructs an $frpt_b$ blaming c as the packet dropper and passes the fault report back to s. On the way to s, each intermediate node validates the $frpt_b$ before passing it on to its predecessor. If the *frpt* is

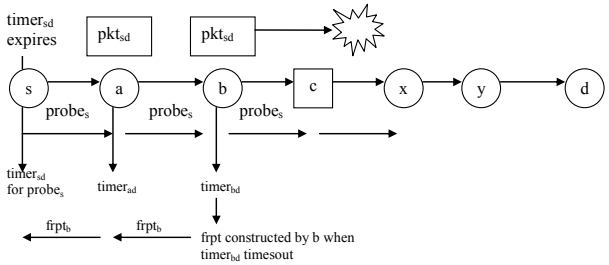

Fig. 2. A tracing round in which malicious node c drops a packet en route to d. When the $timer_{sd}$ lapses, it constructs a probe to flesh out the faulty node. Each subsequent node on the path validates and examines the probe before forwarding it on. The node whose record neither matches the probe's packet and acknowledgement, holds the probe and remains silent. Eventually the $timer_{bd}$ expires and it creates the $frpt_b$ blaming c for the fault.

found to be invalid, then the node who first "notices" the invalid report constructs an *frpt* of its own naming its successor and the creator of the suspect *frpt* as malicious or faulty. The next case will demonstrate what happens when a malicious node attempts to frame an innocent node.

Once the fault report reaches *s*, the source will know a problem exists with either node *b* or *c*. Both nodes are thrown out because non-repudiation is not possible with a symmetric system (*b* could be lying).

Note that if node *c* had simply dropped the probe instead of passing it on, *c* would have signed its own "death warrant" as the timer of its predecessor would still elapse causing a *frpt* to be constructed and sent back to *s*. The above example (figure 2) illustrates that even if a malicious node "acts" innocent and cooperative when it is "aware" of an investigation, it still will be found out. A similar scenario will play out if node *c* corrupted the packet instead of dropping it. In particular, node *x* (and every remaining node on the route to *d*) would not have the correct record of the packet so it would hold the probe and remain silent (see table 1).

The authentication chain of the probe is illustrated in figure 3 and is based on the routing example depicted in figure 2. When *s* receives no valid *ack* or *frpt*, it builds the probe

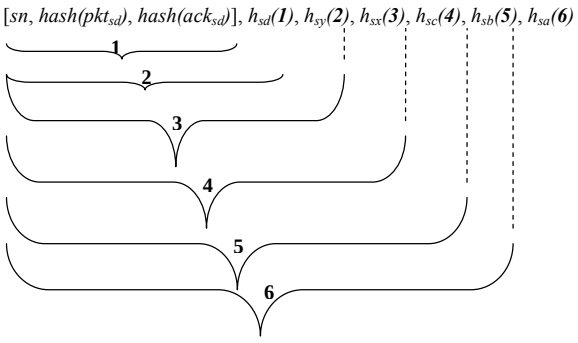

Fig. 3. The authentication chain linking of the probe based on the example of figure 2. The payload is abbreviated to save space in this illustration and is authenticated in a chained fashion.

using successive symmetric key HMACs starting with the shared key between *s* and *d*, and working back by hashing the resulting payload with node *s* and *y*'s shared key, followed by hashing that result with key *sx* and so on until the entire probe is hashed with *s*'s immediate successor's shared symmetric key (node *a* in the illustrated example).

Once the probe is constructed, *s* sends it to its successor (*a*) who takes section 6 (see figure 3) and hashes it with the shared key (*sa*). If the result matches *a*'s tag, $h_{sa}(6)$, node *a* strips away its tag and forwards the remaining probe to its successor only when the probe's contents match pkt_{sd} and not ack_{sd}. Each subsequent node will validate and peel away a layer of the probe before forwarding it on. As mentioned above, the probe stops when both the packet and the acknowledgement of the probe do not match a node's record of the packet and acknowledgement. A node will also not forward a probe if both the packet and acknowledgement match its record to trace node(s) colluding with a malicious node and to deal with situations when an acknowledgement is dropped, corrupted, or forged. Table 1 outlines when a node forwards or does not forward a probe.

TABLE 1
POSSIBLE STATES DURING PROBING ROUND

Packet	Acknowledgement	Action Node Takes
Matches	Matches	Do not forward; remain silent
Matches	No Match	Forward probe to successor
No Match	Match	Can not happen
No Match	No Match	Do not forward; remain silent

Actions taken when a node receives a probe. The probe is only forwarded if its packet matches the node's packet record but the acks do not match.

If for some reason, the probe is found invalid by a node (it has been tampered with), then that node will not forward the probe. The node upstream of the tampering node will construct a *frpt* naming the guilty node once its timer expires.

After a probe stops, the first upstream node whose timer lapses, constructs the *frpt* in a similar authentication chain as the probe was constructed. In our previous example (figure 2), the probe stopped at node *x*. Since node *b* did not receive a valid *frpt* it constructs its own $frpt_b$ as shown in figure 4(a):

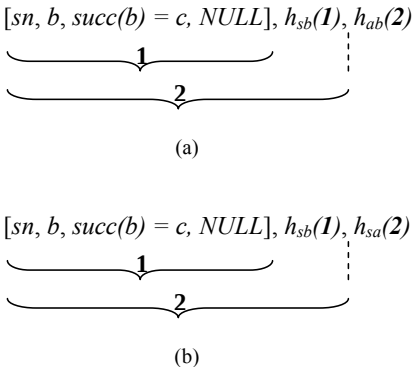

Fig. 4. The authentication chain of fault report $frpt_b$. The payload is abbreviated to save space. (a) the fault report created by node *b*. (b) the fault report after validated by node *a* with *a*'s commitment tag.

Once the *frpt* is constructed, it is then sent back to *s*.

Continuing with our example, node *a* receives and authenticates section 2 of $frpt_b$ (figure 4(a)). If it matches $h_{ab}(2)$, node *a* strips off its tag, hashes the remaining fault report (section 2) using the shared key between *a* and the node preceding *a* (thus committing node *a* to the validity of the *frpt* it received from *b* as shown in figure 4(b)), and sends the new tag along with the remaining *frpt* to its predecessor who repeats the verification process. It is necessary for each node to hash the remaining fault report before passing it on to its predecessor; unless a node authenticates an *frpt* it can never

be certain that the report came from its neighbor downstream (a malicious node could slip in a bad *frpt*). Eventually, the *frpt* wends its way back to *s* who will know the faulty node(s).

On the other hand, if a node receives an invalid *frpt* from its successor, then that node constructs an *frpt* of its own naming its successor and the creator of the received *frpt*. An *frpt* is only forwarded when it is valid; a malicious node would have to tamper with the *frpt* then pass it to its predecessor or create a malicious *frpt* to induce some node upstream into falsely accusing another innocent node as explained in the next case.

3) Case 3 – The Rogue Reporter Attack: Suppose malicious node *c* creates an *frpt* in such a way that node *a* can not validate it upon receipt. For example, node *c* can create a fake tag using the symmetric key between itself and node *a* (see

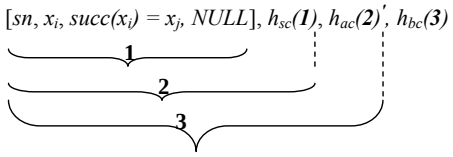

Fig. 5. A fault report $frpt_c$ created by malicious node *c* that purposely created a faulty tag, $h_{ac}(2)'$ When node a receives the *frpt*, it will hash the contents of section 2 with $h_{ac}(2)$ and see it does not equal the tag received from *c*.

figure 5). Node *b* validates the report, hashes section ***2*** using its symmetric key with node *a*, and passes it along to *a*, who will take the contents in section ***2*** and attempt to validate it with its HMAC. Since $h_{ac}(2) \neq h_{ac}(2)'$, node *a* will generate its own fault report but this fault report will contain an extra identifier naming the original creator of the malicious *frpt* (i.e. *s, d, a, succ(a) = b, c, sn,* seq_s). Figure 6 illustrates the rogue reporter scenario.

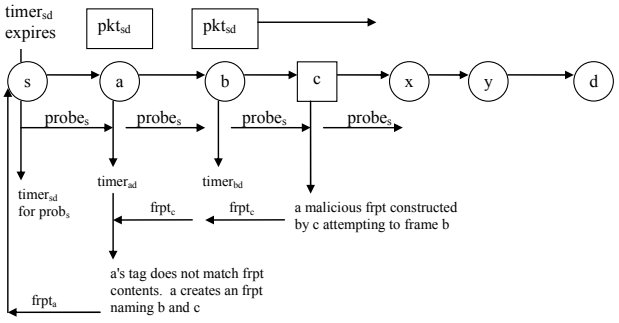

Fig. 6. A case where a malicious node creates a fault report framing an innocent node.

C. Formal O-SKT Algorithm and Proof

In this section we formally demonstrate that the O-SKT tracing algorithm, illustrated in figure 7, will succeed in either sending a packet to a destination or trace at least one faulty node.

Let the routing path from source *s* to destination *d* be: $s = x_0, x_1, \cdots, x_{n-1}, x_n = d$ with the following notations and definitions:

- Let $H(a_k,t) = a_k^t, h_{0,k}^t, h_{1,k}^t, \cdots, h_{t-1,k}^t, h_{t-1,t}^t$ for $1 \le t < k < n$,
- $H(a_k,0) = a_k^0$ and
- $H(a_k,k) = a_k^k, h_{0,k}^k, h_{1,k}^k, \cdots, h_{k-1,k}^k, 1 \le k < n.$
- We have:
  - $H(a_k,t)$ is *a failreport* of node x_k if $a_k^t = x_k b$ where b is either x_{k+1} or $x_{k+1}x_q$ for some $1 \le q < t$.
  - $H(a_k,t)$ is *authenticated* by node x_t if: $h_{t-1,t}^t = h_{t-1,t}(a_k^t h_{0,k}^t h_{1,k}^t \cdots h_{t-1,k}^t)$.
  - $H(a_k,t)$ is a *valid failreport* if: it is a *failreport* of node x_k with $h_{t-1,k}^t = h_{t-1,k}(a_k^t h_{0,k}^t h_{1,k}^t \cdots h_{t-2,k}^t)$; otherwise it is *not valid*.
- $H(a_k,k)$ is *correct* if: $a_k^t = x_k b$, where b is either x_{k+1} or $x_{k+1}x_q$ and $H(a_k,t)$ is valid for all $t = 1,\ldots,k$; otherwise it is *incorrect*.

Source *s*. Set $seq_s = 0$. While connection to *d* has not terminated do:

1. Set $timer_s$ and send pkt_{sd} to *succ(s)*.
2. If ack_{sd} is received before timeout then set $seq_s = seq_s + 1$
3. Otherwise:
   a. Set $timer_{sd}$ and send $prob_s$ to *succ(s)*.
   b. If a valid $frpt_y$ is received before timeout then *y*, *succ(y)*, or creator of a previous bad $frpt_{x_i}$, x_{i_3} is malicious.
   c. Else *succ(s)* is malicious.

Intermediate node *x*. When pkt_{sd} is received:

1. Send pkt_{sd} to *succ(x)*.
2. If a matching ack_{sd} is received then send ack_{sd} to *prec(x)*.
3. Else if a valid $prob_y$ is received then
   a. Set $timer_{sd}$ and send $prob_x$ to *succ(x)*.
   b. If a valid $frpt_y$ is received before $timer_{sd}$ timeout then send $frpt_y$ to *prec(x)*.
   c. Else construct and send $frpt_x$ to *prec(x)*.

Destination *d*. When a valid pkt_{sd} is received:

1. Construct and send ack_{sd} to *prec(d)*.

Fig. 7. The optimistic symmetric key tracing algorithm.

In the routing protocol's tracing round:

1. If an intermediate node x_t does not receive an

authenticated *failreport* from x_{t+1} before timeout, then x_t computes and sends to x_{t-1} a correct *failreport:* $H(a_t,t)$ with $a_t^t = x_t x_{t+1}$.

2. If an intermediate node x_t receives an authenticated and valid *failreport* $H(a_k,t+1)$ from x_{t+1} before timeout, then x_t computes and sends to x_{t-1} the authenticated *failreport:* $H(a_k,t)$ with $a_k^t = a_k^{t+1}$ and $h_{i,k}^t = h_{i,k}^{t+1}$, $i = 0,\ldots,t-1$.
3. If an intermediate node x_t receives an authenticated *failreport* $H(a_k,t+1)$ from x_{t+1} that is not valid before timeout, then x_t computes and sends to x_{t-1} a correct *failreport:* $H_t(a_t,t)$ with $a_t = x_t x_{t+1} x_k$.[2]

Theorem *For any Γ-adversary structure, the optimistic O-SKT communication routing algorithm succeeds in either sending packet pkt_{sd} to destination d, or tracing at least one faulty node.*

Proof. It is obvious that when all nodes obey the protocol, destination *d* will receive pkt_{sd} and *s* will receive ack_{sd}. Secondly, if *s* gets a valid ack_{sd}, then because we assume that HMACs are unforgeable and because *d* will only construct a valid matching ack_{sd} if the received pkt_{sd} is valid, *d* must have received pkt_{sd}.

Suppose the source $s = x_0$ has not received a valid acknowledgement ack_{sd} from x_1 before timeout and that *s* has sent a $probe_s$ downstream. We distinguish two cases.

Case A: *s* does not receive an authenticated *failreport* from x_1 by timeout. Then x_1 is faulty.

Case B: *s* did receive an authenticated *failreport* $H(a_t,1) = a_t^1 h_{0,t}^1$ from x_1, for some $1 \le t < n$.

Case B1: $H(a_t,1)$ is valid. Then $a_t^1 = x_t b$ with *b* either x_{t+1} or $x_{t+1}x_q$ and $h_{0,t}^1 = h_{0,t}(a_t^1)$. Now only one node (other than x_0) that could have validated $a_t^1 h_{0,t}^1$ is x_t. Therefore one of x_t, x_{t+1}, x_q is faulty.

Case B2: $H(a_t,1)$ is not valid. If x_1 were not faulty, the *failreport* $H(a_t,2)$ it received from x_2 would have been authenticated and valid (otherwise x_1 would have sent a correct *failreport*; this is not the case since we are assuming that $H(a_t,1)$ is not valid). This means that: $a_t^2 = x_t b$ for some string *b* and $h_{1,t}^2 = h_{1,t}(a_t^1 h_{0,t}^1)$. Now only x_t could have validated $a_t^1 h_{0,t}^1$. So x_t must have validated a *failreport* with an invalid component. This means that x_t is faulty. Therefore either x_1 is faulty or x_t is faulty.

We conclude that if *s* does not receive an acknowledgement before timeout for a packet pkt_{sd} is sent to *d*, then depending on the *failreport* that *s* has received, either x_1 is faulty (Case A), or one of x_t, x_{t+1}, x_q is faulty (Case B1), or one of x_1, x_t is faulty (Case B2). □

Remark. Observe that O-SKT is not vulnerable to malicious timing attacks under our weak synchrony assumption. Suppose node x_i has set its timer t_i but has not received anything at timeout. Then node x_{i+1} must have set its timer at t_i-2τ and there are two cases: x_i has received something before timeout. Then x_{i+1} is faulty. Else x_{i+1} would have issued a fault report. Therefore, we conclude that if the timer of x_i timed out without x_i having received a fault report then x_{i+1} is faulty.

Once again we reiterate that the O-SKT is a tracing algorithm for communication protocols over an established path in which honest nodes are actual neighbors. So far there are no effective mechanisms to trace faulty nodes during a route discovery phase thereby preventing man-in-the-middle relay attacks. Only out-of-system mechanisms (temporal, locational, or a combination of both) can possibly counter this type of attack.

IV. USING TESLA CHAINS

There are other symmetric chain authentication mechanisms employing one-way hash chains such as TESLA [11] that could be incorporated into our proposed tracing algorithm. However, there are no additional savings by doing so. TESLA suffers from the same non-repudiation issues and, in the worst case, will still sacrifice up to three nodes for each malicious attack during the tracing round. The only way to reduce the number of blamed nodes is to use a public key tracer such as [3], and then the number of blamed nodes will be reduced to two.

TESLA was designed for networks that deal with masses of broadcast data, which MANETs typically do not have. The fault report, once created by the originator, is passed to its predecessor who validates and then commits to the validity of the report before forwarding it on. In fact, it may cost more to use TESLA due to the multiple recursive hashing mechanisms necessary to execute the protocol. Additionally, there would be delays in forwarding the fault reports as each intermediate node would have to wait for the originator to reveal the key necessary for validation.

V. CONCLUSIONS AND FUTURE WORK

We propose an optimistic Byzantine tracing algorithm based on an existing and sound protocol. Instead of using digital signatures, we take advantage of the computational efficiency of HMACs. This algorithm is appropriate for ad

[2] Justification: If a non-faulty x_{t+1} did not receive an authenticated *failreport* $H(a_k, t+2)$ that is valid, then x_{t+1} would have sent to x_t a correct *failreport* $H(a_t,t)$. This is not the case. It follows that either x_{t+1} is faulty, or that $H(a_k, t+2)$ is valid and that x_k validated a *failreport* with an invalid component. So either x_{t+1} is faulty, or x_k is faulty.

hoc networks that can not employ encryption or public key mechanisms and uses only hash functions. The algorithm is guaranteed to successfully deliver packets from source to destination or, in the event of a problem, trace at least one faulty or malicious node.

In this work we have made several assumptions: namely, that some TTP [12] distributes the secret shared keys among the nodes in the MANET, and that the participating nodes will not divulge their secret keys.

How will our proposed tracing algorithm stand up to a Sybil attack [5]? In our design, the adversary will pay for each successful attack and as long as the adversary is limited to corrupting no more than k nodes such that $k < n$, the total number of nodes in the network, then the MANET should converge to fault free status. However, if the adversary is able to recruit another k nodes, then another set and so on, the adversary will eventually win. More research is needed to see if our symmetric tracer can effectively deal with such attacks. Furthermore, our proposed tracer is at the mercy of the route discovery mechanism employed. There is no such thing as a secure routing algorithm ensuring a route is free from man-in-the-middle relay attacks in the general adversary model [3]. Clearly more work needs to be done in this area to prevent malicious nodes from fabricating their positions to one another and especially prevent them from being transparent in a communications route.

In future works trust models must be developed to deal with fault reports. When the source node receives a fault report, it knows that up to three nodes may be bad. How, then, does the source relay to the rest of the network its adverse experience with these blamed nodes? Do the remaining nodes take the source's information at face value or should there be a mechanism whereby the source must commit to the validity of its report? How do we prevent an adversary from tampering with the reports in order to falsely accuse an otherwise innocent node?

We must also address the issue of what to do with Byzantine nodes once they are fleshed out. Do we neutralize them? Do we revocate them? If so how? Several ideas include: cordoning the bad guys into a separate area much like a "holding cell"; or revealing the malicious node's secret key thus marking the node as untrustworthy and essentially excommunicating it from the network.

VI. ACKNOWLEDGEMENT

I would like to thank Dr. Mike Burmester, my major professor, for his continued guidance and support in this project. Also, I would like to thank Dr. Tri Van Le for his helpful discussions and consideration. Additionally, I want to acknowledge my colleagues in the graduate Mobile Ad Hoc Networks course (Florida State University, spring semester, 2006).

REFERENCES

[1] B. Awerbuch, D. Holmer, C. Nita-Rotaru and H. Rubens. "An On-Demand Secure Routing Protocol Resilient to Byzantine Failures", *ACM Workshop on Wireless Security (WiSe '02)*, September 2002.

[2] M. Bellare, R. Canetti, H. Krawczyk. "Keying Hash Functions for Message Authentication," *Advances in Cryptography – Crypto 96 Proceedings*, June 1996.

[3] M. Burmester and Tri Van Le. "Security Issues of Mobile Ad Hoc Networks," accepted to appear in *Network Security, ed.* S. Huang, D. MacCallum and Ding Zhu Du, Springer, 2006.

[4] M. Burmester, Tri Van Le, and M. Weir. "Tracing Byzantine Faults in Ad Hoc Networks," *Proc. Computer, Network and Information Security 2003, (2003).*

[5] J.R. Douceur, "The Sybil Attack," *Proc. 1st International Workshop on Peer-to-Peer Systems* – IPTPS'02, 2002.

[6] M. Hirt and U. Maurer, "Player Simulation and General Adversary Structure in Perfect Multiparty Computation," *Journal of Cryptology*, Vol 13 No 1, pp. 31-60, 2000.

[7] D. B. Johnson and D. A. Maltz. "Dynamic Source Routing in Ad Hoc Networks," *Mobile Computing, ed.* T. Imielinski and H. Korth, Kluwer Academic Publisher, pp. 152 – 181, 1996.

[8] L. Lamport, R. Shostac, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol 4 No 2, pp. 382-401, 1982.

[9] J. Marshall, V. Thakur and A. Yasinsac, "Identifying Flaws in the Secure Routing Protocol," *Proc. 22nd International Performance, Computing, and Communications Conference* (IPCCC2003), pp. 167-174, April 9-11, 2003.

[10] C. E. Perkins and E. M. Royer. "Ad Hoc On-Demand Distance Vector Routing," *IEEE Workshop on Mobile Computing Systems and Applications*, pp. 90 – 100.

[11] A. Perring, R. Canetti, J.D. Tygar and D. Song. "The TESLA Broadcast Authentication Protocol," *Cryptobytes*, Volume 5, No. 2 (RSA Laboratories, Summer/Fall 2002), pp. 2-13, 2002.

[12] B. Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd Ed., *John Wiley & Sons*, New York, 1996.

[13] P. F. Syverson, D. M. Goldschlag and M. G. Reed. "Anonymous Connections and Onion Routing," *IEEE Symposium on Security and Privacy*, 1997