Mobile Agent Route Protection through Hash-Based Mechanisms*

Josep Domingo-Ferrer

Universitat Rovira i Virgili, Dept. of Computer Engineering and Mathematics,
Av. Països Catalans 26, E-43007 Tarragona, Catalonia, Spain,
jdomingo@etse.urv.es

Abstract. One approach to secure mobile agent execution is restricting the agent route to trusted environments. A necessary condition for this approach to be practical is that the agent route be protected. Previous proposals for agent route protection either offer low security or suffer from high computational costs due to cryptographic operations. We present two fast, hash-based mechanisms for agent route protection. The first solution relies on hash collisions and focuses on minimizing the computational cost of route verification by hosts along the route; the cost is shifted to the stage of route protection by the agent owner. The second solution uses Merkle trees and minimizes the cost of route protection by the agent owner, so that a single digital signature suffices to protect the whole route; for hosts along the route, the verification cost is similar to the cost of previous schemes in the literature, namely one digital signature verification per route step. The first solution is especially suitable for agent routes which go through heavily loaded hosts (to avoid denial of service or long delay). The second solution is more adapted to mitigating the bottleneck at agent owners who are expected to launch a great deal of agents. Both solutions provide independent protection for each route step and can be extended to handle flexible itineraries.

Keywords: Mobile agent security, agent route protection, hash collisions, Merkle trees.

1 Introduction

It is increasingly difficult for individuals to take advantage of the exponentially growing wealth of information on the Internet. Mobile agents can be very helpful, as they are programs that roam the network searching for products that fit best buyer requirements. The mobility property raises important security issues: (i) it is important to protect network hosts against malicious agents; (ii) agents should also be protected against malicious hosts. The first problem is analogous to anti-viral protection, and has thus profusely been studied. The second problem consists of attacks to modify the route or the code of the mobile agent by a

* This work is partly supported by the Spanish CICYT under project no. TEL98-0699-C02-02.

C. Pandu Rangan, C. Ding (Eds.): INDOCRYPT 2001, LNCS 2247, pp. 17–29, 2001.

malicious host which may or may not be in the initial agent route. Only a few (non-exclusive) approaches to protecting agents against malicious hosts have been proposed:

- *Encrypted functions.* In [16], the agent code is identified with the function it computes and a solution is proposed based on computing with encrypted functions (an extension of computing with encrypted data, [3]). This only works for a restricted class of functions.
- *A posteriori detection.* Attacks are detected after they have happened (which may be too late), and upon detection, information can be retrieved and used to accuse the malicious host. Examples are [1,10,21]. Code watermarking [18] would also fall in this category.
- *Obfuscation of agent code.* This is an alternative which reduces code readability and thus makes attacks unlikely. Examples can be found in [5,7,9,17].
- *Trusted environments.* Agents only visit trusted execution environments. A necessary condition to restrict mobility to trusted environments is that the agent route be protected.

This paper contributes to the last aforementioned approach by showing efficient ways to protect agent routes. Section 2 discusses previous work. Section 3 describes a mechanism based on hash collisions which has an extremely low cost in terms of verification by hosts along the route, but is more costly for the agent owner. Section 4 describes a solution based on Merkle trees which has the same verification cost for hosts along the route than previous schemes, but offers a lower computational cost for the agent owner. Section 5 contains some conclusions and shows how the proposed schemes can be used to protect flexible itineraries with alternative paths.

2 Previous Work on Agent Route Protection

In [19] a general concept of an agent route, called itinerary, is given. Flexible agent travel plans can be specified which allow dynamic adaptation and expansion during the execution of the agent. A shortcoming of this scheme is that it is vulnerable to corruption of hosts specified in the itinerary; a corrupted host can modify the itinerary or attack other hosts in the itinerary to cause denial of service to the agent. In [2], a partial solution to the above problems is outlined, but no countermeasures are described to prevent host addition or removal.

In [22] nested encryptions and signatures are used to provide increased security to the agent route. The basic idea is to sign and encrypt host addresses iteratively to attain a high level of security. Let PK_i be the public key of the i-th host and let $S_0(\cdot)$ be the signature function of the agent owner H_0. Then the whole route r is coded as

$$r = E_{PK_1}[H_2, S_0(H_1, m_1, H_2, t, E_{PK_2}[\cdots]), E_{PK_2}[\cdots]]$$

where, for $i = 1$ to $n - 1$,

$$E_{PK_i}[\cdots] = E_{PK_i}[H_{i+1}, S_0(H_i, m_i, H_{i+1}, t, E_{PK_{i+1}}[\cdots]), E_{PK_{i+1}}[\cdots]]$$

and $E_{PK_n}[\cdots] = E_{PK_n}[H_0, S_0(H_n, m_n, H_0, t)]$. In the above expressions, H_i is the IP-address of the i-th host, m_i is the method (code) to be run at H_i, H_0 is the IP-address of the agent owner (host which originates the agent) and t is a timestamp (IP stands for Internet Protocol).

The above coding allows the route to be enforced as follows. The first host H_1 receives r and decrypts it using its private key to obtain $S_0(H_1,m_1,H_2,t,E_{PK_2}[\cdots])$, H_2 and $E_{PK_2}[\cdots]$. Using the public key of the agent owner, H_1 can verify that the next host address H_2 and the value of the rest of the route $E_{PK_2}[\cdots]$ were included in the route by the agent owner. The inclusion of his own address H_1 and method m_1 in the signature allows H_1 to detect that he was also included in the route by the agent owner and that the method m_1 being enclosed with the route is what the agent owner meant to be run at H_1 (this is actually an enhancement over the original proposal [22], which did not include the methods in the signatures). The timestamp t is used to include an expiration date to prevent re-use of older routes of the same agent by malicious hosts. Beyond these validations, H_1 cannot obtain any additional knowledge on the rest of the route since all remaining information is encrypted under the public key PK_2 of H_2. Then H_1 sends the agent to H_2 together with

$$r_1 = E_{PK_2}[H_3, S_0(H_2, m_2, H_3, t, E_{PK_3}[\cdots]), E_{PK_3}[\cdots]]]$$

The above decryption and verification process is repeated by H_2 and so on up to H_n. After n steps, the agent is returned by H_n to H_0. The dark point of proposal [22] is the high processing cost of nested encryptions and signatures at each host along the route (one decryption and one signature verification are needed).

The approach described in [11] is similar to [22] in that it uses one encryption and one signature for each step of the route. In this case, encryptions are nested, whereas signatures are not. This scheme can be generalized to accomodate alternative itineraries.

In the next two sections, two mechanisms are described which substantially reduce the computational overhead of the above proposals, while still preserving the feature that route verification at H_i does not require any information on previous route steps (in particular methods m_j for $j < i$ can be discarded).[1]

3 Reducing the Computational Cost of Route Verification

In comparison with proposals recalled in Section 2, the mechanism proposed in this section focuses on reducing the cost of route verification at the expense of making route protection more costly for the agent owner. This is especially useful if some of the hosts in the route are usually overloaded with computation

[1] Note that a single signature on the concatenation of all route steps is very efficient from a computational viewpoint, but forces the agent to convey all route steps (with their methods m_i) until the route is finished.

(verifications have to be fast to avoid long delay or denial of service). A realistic case where this may happen is when the agent route goes through hosts containing massively accessed Internet or database search engines.

The scheme proposed here borrows from the MicroMint micropayment system [15] the idea of replacing digital signatures with collisions of a hash function; even if not explicitly mentioned, all hash functions used in what follows are assumed to be one-way and collision-free (see Appendix). For implementation purposes, one-way hash functions like SHA [13], MD5 [14] or DES [12] are reasonable choices. The main advantage of replacing digital signatures with hash functions is the speed gain. According to figures by Rivest, if computing an RSA signature takes time t, verifying an RSA signature with low public exponent can be done in $t/100$ and, more important, evaluating a hash function can be done in $t/10000$.

Based on the above ideas, we propose to extend the application of hash collisions from micropayments to agent route protection. Basically, the problem is the same: instead of fast verification of coins by a payee, we need fast verification of route steps by the hosts along the route.

We assume in what follows that, for each host H_i, the agent owner H_0 has set up a symmetric encryption key k_i for secret communication from H_0 to H_i. A way to do this is for H_0 to generate k_i and send $E_{PK_i}(k_i)$ to H_i, where PK_i is the public key of host H_i[2]. The key k_i can be used by H_0 for all routes she schedules through H_i. This allows public-key encryption to be replaced with symmetric encryption; an advantage is that, with the higher speed of symmetric encryption, we can afford to encrypt the code methods to be run at each host, which results in higher confidentiality.

Also, one-way hash functions F_m and F_t are used whose outputs are, respectively, strings of lengths m and t. Standard hash functions, such as SHA, MD5 or DES, may have more output bits than the required m and t; in that case, take the m, resp. t, lower bits as output.

Algorithm 1 (Route protection with hash collisions)

1. [**Notation**] *The agent owner chooses n hosts represented by their IP-addresses $H_1, \cdots H_n$. Let H_0 be the address of the agent owner. Let k_i be the symmetric encryption key set up for secret communication from H_0 to H_i. Let m_i be the code to be run at H_i.*
2. [**Encryption**] *The agent owner computes $U_i = E_{k_i}(H_{i-1}, H_i, H_{i+1}, m_i)$ for $i = 1, \cdots, n-1$. For $i = n$, compute $U_n = E_{k_n}(H_{n-1}, H_n, H_0, m_n)$.*
3. [**Collision computation**] *For $i = 1$ to n, H_0 computes a k-collision $(x_{i,1}, \cdots, x_{i,k})$ such that*

$$F_m(x_{i,1}) = F_m(x_{i,2}) = \cdots = F_m(x_{i,k}) = y_i$$

[2] We do not require that H_0 authenticate itself to H_i when setting up k_i. Although authenticating the origin of k_i would certainly render route step authentication straightforward, it would burden H_i with something like a signature verification, which is against the main goal of the scheme discussed here (minimizing computation at route hosts).

where y_i is an m-bit string such that its t high-order bits match $F_t(U_i)$.

Algorithm 2 (Route verification with hash collisions)

1. [**Start of route**] *The agent is sent by H_0 to the first host of the route, namely H_1, together with the n k-collisions (one for each route step) and U_i for $i = 1, \cdots, n$.*
2. [**Operation at host** H_i] *H_i takes the i-th k-collision and checks that all of its k values actually hash to the same y_i. Then H_i checks that $F_t(U_i)$ matches the t high-order bits of y_i. If the check is OK, H_i decrypts U_i and obtains $(H_{i-1}, H_i, H_{i+1}, m_i)$ for $i < n$ or (H_{n-1}, H_n, H_0, m_n) for $i = n$. At this moment, H_i runs m_i and, after that, forwards $U_{i+1}, \cdots, U_n$ along with the k-collisions corresponding to the remaining route steps to H_{i+1} (or to H_0 if $i = n$).*
3. [**End of route**] *The route ends at the agent owner H_0.*

3.1 Computational Cost

For a host H_i along the route, the computational cost of route verification following Algorithm 2 is reduced to k hash computations and one symmetric decryption.

For the agent owner, the cost is dominated by the computation of k-collisions. Objections have been raised against the high cost of k-collision computation in the case of large-scale MicroMint [20]. We next give a quantitative cost analysis and illustrate the practicality of using k-collisions for our application with a realistic example.

Lemma 1. *If N hash values are computed, the probability of obtaining at least a k-way collision of length m bits with the t high-order bits fixed is*

$$1 - [e^{-N2^{-m}} \sum_{i=0}^{k-1} \frac{(N2^{-m})^i}{i!}]^{2^{m-t}} \tag{1}$$

Proof. Computing a hash value $y = F_m(x)$, where the length of y is m bits, is analogous to the problem of tossing a ball x at random into one of 2^m bins (the possible values of y). If we call a ball x "good" when the t high-order bits of y match a fixed pattern, then N hash computations will yield an expected number $N' := N2^{-t}$ of good balls to be tossed at random into one of 2^{m-t} bins (the possible values of the $m - t$ low-order bits of y). The probability of obtaining at least one k-way collision is 1 minus the probability of all bins getting $k - 1$ or less balls. The probability of a given bin getting a ball in a given toss is $p := 2^{t-m}$. If $N'p = N2^{-m} < 5$, $p < 0.1$ and $N' > 30$ (equivalently, $N > 2^t 30$), the probability that a bin gets $k-1$ or less balls can be computed using a Poisson approximation:

$$P(k-1 \text{ or less balls in a bin}) = e^{-N'p} \sum_{i=0}^{k-1} \frac{(N'p)^i}{i!} \tag{2}$$

Now the probability of getting at least one k-collision is

$$P(\text{at least one } k\text{-collision}) = 1 - (P(k-1 \text{ or less balls in a bin}))^{2^{m-t}} \quad (3)$$

By substituting Expression (2) in Expression (3), we obtain Expression (1). □

Example 1. In Algorithm 1, let F_m be the low-order m bits of the output of the SHA hash function; formally:

$$F_m(x) = [SHA(x)]_{1\cdots m}$$

Similarly define F_t as $[SHA(x)]_{1\cdots t}$. According to recent figures given by [20], current custom chip technology allows 2^{23} hashes to be computed per second per dollar of chip cost. Assume, as [15], that the agent owner spends $100,000 in custom chips, so that she can evaluate F_m around 2^{39} times per second (we approximate $2^{23} \times 100,000$ by 2^{39} for ease of calculation). Take $k = 4$ and assume the agent owner is ready to spend 2^8 seconds (4 minutes) to compute a 4-collision; in that time, $N = 2^{47}$ hash values can be computed. If $m = 52$ and $t = 21$ are used, Lemma 1 gives the probability of the agent owner getting at least one good 4-collision of F_m (with the t higher-order bits fixed) in four minutes:

$$1 - [e^{-2^{-5}}(1 + 2^{-5} + \frac{(2^{-5})^2}{2} + \frac{(2^{-5})^3}{6})]^{2^{31}} \approx 1 - 0.716 \cdot 10^{-36} \quad (4)$$

Thus, a good 4-collision will be obtained by the agent owner in four minutes with extremely high probability. □

Thus, it can be seen from Example 1 that coming up with a 4-collision with fixed t high-order bits is costly but by no means unaffordable for the agent owner [3]. This is compensated by the cost reduction in route verification (a relevant figure if the route goes through very busy hosts).

3.2 Security of the Scheme

We will show in this section that the following properties for agent route protection identified as relevant for agent route protection in [22] (see Section 2) are fulfilled by the above scheme:

P1. *Hosts should not be able to modify the agent route.*
P2. *Every host should be able to check it was included in the agent route.*
P3. *Every host should only see the previous host and the next host in the route.*
P4. *Every host should be able to authenticate the host the agent is coming from.*
P5. *A host should not be able to replace a route by older routes of the same agent.*

[3] Unlike for MicroMint, finding one good collision at a time is enough in our application, so the storage and sorting costs additional to the chip cost are much lower.

The basic security of the above scheme rests on two defense lines:

- The difficulty of computing hash collisions of F_m with standard hardware
- For fixed U_i, the unfeasibility of finding $U_i' \neq U_i$ such that $F_t(U_i) = F_t(U_i')$ and such that decryption of U_i' under k_i' yields H_i' as the second of the three IP addresses obtained, where k_i' is the key shared between the agent owner and the host with IP address H_i'.

With proper parameter choice, the difference between the custom hardware of the agent owner and the standard hardware of a typical user is enough to guarantee efficient computation of m-bit k-collisions for the former and difficult computation for the latter. The following example illustrates this point.

Example 2. In [15], it is assumed that a 1995 standard workstation could perform 2^{14} hash operations per second. Using Moore's law (computer hardware gets faster by a factor of 2 every eighteen months), a more realistic figure for a 2001 standard workstation is that it can perform $2^{14} \cdot 2^4 = 2^{18}$ hash operations per second.

With the above choice $m = 52$, $t = 21$ and $k = 4$, assume 2^{25} seconds (more than one year time) are devoted to compute a 4-collision; in that time, $N = 2^{43}$ hash values can be computed by an attacker owning a standard workstation. Lemma 1 gives the probability of the attacker getting at least one good 4-collision of F_m (with the t higher-order bits fixed) in 2^{25} seconds:

$$1 - [e^{-2^{-9}}(1 + 2^{-9} + \frac{(2^{-9})^2}{2} + \frac{(2^{-9})^3}{6})]^{2^{31}} \approx 1 - 0.9987008 = 0.0012992 \quad (5)$$

Thus, the probability of forging a good 4-collision in one year time is very low. As computer hardware gets faster, slight increases of k may be needed in order for k-collisions to be computable by the agent owner and not by typical users. □

Regarding the second defense line, for a fixed U_i, consider the feasibility of finding $U_i' \neq U_i$ such that $F_t(U_i) = F_t(U_i')$ and such that U_i' decrypts into a valid IP address H_i' when being decrypted by a host H_i' under its key k_i'. Note that it does not make sense for H_i to try to forge a $U_i' \neq U_i$ (this does not cause any deviation in the route); nor does it make sense for a host H_i' to try to replace U_i with a different U_i' which decrypts into H_i' as second IP address when using the key k_i' shared between H_i' and the agent owner. What makes sense is for a host H_j to try to forge $U_i' \neq U_i$, for $j \neq i$; this could alter the i-th step of the planned route. Thus, the attacker does not know the encryption key k_i' that will be used by the host H_i' at the i-th step of the altered route. In this case, the best strategy is to randomly look for a $U_i' \neq U_i$ that satisfies $F_t(U_i) = F_t(U_i')$ and then hope that decryption of U_i' under k_i' will yield the 32 bits corresponding to the IP address H_i' in the correct positions. An attempt to meeting this second condition with a random U_i' will succeed only with probability 2^{-32}. Thus, a huge number of attempts are likely to be needed. On the other hand, each attempt requires finding a U_i' colliding with U_i under F_t, and then decrypting U_i'; with proper parameter choice, finding a colliding U_i' takes a non-negligible computing

time (see Note 1 below), which makes it impractical to perform a huge number of attempts.

Note 1 (On satisfying $F_t(U_i) = F_t(U_i')$). If F_t is one-way, a $U_i' \neq U_i$ such that $F_t(U_i') = F_t(U_i)$ must be looked for at random. The probability of coming up with a good U_i' is analogous to the probability of hitting a fixed bin when randomly tossing a ball into one of 2^t bins; thus this probability is 2^{-t}, which means that 2^t hash computations will be needed on average to find a suitable U_i'. Assuming $t = 21$ and a processing power of 2^{18} hash values per second as above, this means 8 seconds for a standard user to find U_i'.

Now let us check P1 through P5 stated above.

P1. To modify the agent route, at least one step U_i should be modified into a $U_i' \neq U_i$ by some attacker who does not know how to decrypt U_i nor U_i' (see discussion above). This has been shown to be computationally infeasible earlier in this section.

P2. Host H_i decrypts U_i using the key k_i and should obtain three IP N addresses, of which the second one should be its own address H_i. If this is not so, then H_i was not included in the route by H_0.

P3. Decryption of U_i allows H_i to learn the addresses of H_{i-1} and H_{i+1}. The remaining addresses of the route are encrypted in U_j, for $j \neq i$, with keys k_j unknown to H_i. Thus, the rest of hosts in the route remain undisclosed to H_i.

P4. This property can be satisfied only if IP communication between hosts is authenticated. In this case, every host H_i knows which is the *actual* host H_{i-1}' the agent is coming from. On the other hand, decryption of U_i provides H_i with the IP address of the host H_{i-1} they agent *should* come from. In this way, H_i can detect whether $H_{i-1}' \neq H_{i-1}$.

P5. To satisfy this property, a timestamp or an expiration date t should be appended by the agent owner to each tuple $(H_{i-1}, H_i, H_{i+1}, m_i)$ before encrypting the tuple into U_i.

4 Reducing the Computational Cost of Route Protection

The main thrust behind the proposal of Section 3 was to reduce the verification cost. If the agent owner is very busy launching a lot of agents each on a different route, the priority may be to reduce the cost of route protection with respect to previous proposals. This is what is achieved by the mechanism presented in this section: as compared to conventional schemes recalled in Section 2, route protection is faster and route verification requires the same amount of work (one signature verification per step).

The mechanism discussed here uses binary Merkle trees as basic tool. Binary Merkle trees are trees constructed as follows. Each leaf is a statement plus the hash of that statement. The hash values of pairs of siblings in the tree are hashed together to get a hash value for their parent node; this procedure iterates until the hash value RV of the root node of the tree has been obtained. We use Merkle

trees to construct signatures for each step of the route in a similar way they are used in [6] and [4] to construct public-key certificates. The main advantage of Merkle trees is that one signature on the root node of the tree allows independent integrity verification for all leaves, provided that the hash function used is one-way and collision-free.

Suppose the agent owner has to sign the steps of a route or of several routes, where the i-th step of the route is $(H_{i-1}, H_i, H_{i+1}, m_i)$, that is, the IP addresses of three consecutive hosts plus the method to be run at host H_i (just like in the mechanism described in Section 3).

The algorithm below uses a one-way collision-free hash function F and allows the agent owner to sign all the steps corresponding to a route with a single digital signature.

Algorithm 3 (Route protection with Merkle trees)

1. [**Notation**] *Let the IP addresses of the hosts along the route be* $H_1, \cdots, H_n$. *Let* k_i *be a symmetric encryption key set up for secret communication from the agent owner* H_0 *to* H_i [4].
2. [**Encryption**] *The agent owner computes* $U_i = E_{k_i}(H_{i-1}, H_i, H_{i+1}, m_i)$ *for* $i = 1, \cdots, n-1$. *For* $i = n$, *compute* $U_n = E_{k_n}(H_{n-1}, H_n, H_0, m_n)$.
3. [**Merkle tree computation**] H_0 *computes a binary Merkle tree by taking as leaves the statements* U_i *and their hash values* $F(U_i)$, *for* $i = 1$ *to* n.
4. [**Signature**] *After creating the Merkle tree, its root value* RV *is digitally signed into* $S_0(RV)$ *by the agent owner by using her private key.*

Define the *ver-path* for a route step U_i to be the path from the leaf containing $(U_i, F(U_i))$ to the root RV, together with the hash values needed to verify that path (*i.e.*, the hash values of siblings of nodes along that path). Note that the length of the ver-path equals the height of the tree for a leaf and grows only logarithmically with the number of leaves.

Let us now detail the route verification algorithm:

Algorithm 4 (Route verification with Merkle trees)

1. [**Start of route**] *The agent is sent by* H_0 *to the first host of the route, that is* H_1, *together with all route steps* U_i, *for* $i = 1, \cdots, n$, *and the Merkle tree for the whole route with signed* RV.
2. [**Operation at host** H_i] H_i *takes the* i*-th route step* U_i, *extracts its ver-path from the Merkle tree and verifies this ver-path (by recomputing all intermediate hash values starting from* $(U_i, F(U_i))$ *down to the root). Then* H_i *checks whether the root value recomputed using* U_i *and its ver-path are the same* RV *signed by* H_0. *If the check is OK,* H_i *decrypts* U_i *and obtains* $(H_{i-1}, H_i, H_{i+1}, m_i)$ *for* $i < n$ *or* (H_{n-1}, H_n, H_0, m_n) *for* $i = n$. *At this*

[4] As in the previous scheme, we do not require here that H_0 authenticate itself to H_i when setting up k_i (even if this would make route step authentication nearly trivial). The reason is that this scheme aims at minimizing the computation at H_0.

moment, H_i runs m_i and, after that, forwards to H_{i+1} (or to H_0 if $i = n$) $U_{i+1}, \cdots, U_n$ along with the part of the Merkle tree needed to verify the remaining route steps (nodes which do not belong to any ver-path of remaining steps can be pruned).

3. **[End of route]** *The route ends at the agent owner H_0.*

4.1 Computational Cost

For a host H_i along the route, the computational cost of route verification using Algorithm 4 is a number of hash computations equal to the length of the ver-path for step i (typically a one-digit figure), plus one signature verification and one symmetric decryption. According to the figures by Rivest mentioned in Section 3, a signature verification takes as long as 100 hash computations, so the route verification cost is essentially the cost of one signature verification (just like for previous schemes described in Section 2).

For the agent owner, the cost consists of the hash computations needed to create the Merkle tree (or update it, if the same tree is shared by all routes), plus the signature on the root value RV. Since computing a digital signature typically takes as long as 10000 hash computations, the cost is essentially one digital signature for the whole route. This is much lower than the cost for schemes in Section 2, which required one digital signature per route step.

In addition to reducing the number of signatures for route protection, Merkle trees allow mobile agents to convey the protected route in a compact way. Independent protection of each route step would require the agent to initially convey one signature per step, that is $1024n$ bits for an n-step route (assuming 1024-bit RSA signatures). Storing the route as a binary Merkle tree with one leave per step requires one hash value per tree node and a single signature for the whole tree; this makes $1024 + 160 * (2n - 1)$ bits to be conveyed by the agent, assuming SHA is used as hash function. This is substantially less than $1024n$ bits.

4.2 Security of the Scheme

Using Merkle trees to extend a single digital signature to a collection of messages is not new [8,6]. Provided that the hash function used is one-way and collision-free, there is no loss of security with respect to signing messages individually. Let us check for this scheme the security properties P1 to P5 discussed in Section 3.2 for the hash collision scheme.

P1. To modify the i-th step, U_i should be modified into $U_i' \neq U_i$. This would require finding a ver-path for U_i' such that its verification yields the same root value obtained from verification of the ver-path of U_i. If the hash function used is one-way and collision-free, this is computationally infeasible.

P2,P3,P4,P5. Same comments as in Section 3.2.

5 Conclusions and Extension to Flexible Itineraries

One approach to secure mobile agent execution is restricting the agent route to trusted environments. A necessary condition for this solution to be practical is that the agent route be protected. We have proposed hash-based schemes which try to improve computational efficiency without degrading security:

- The mechanism based on hash collisions concentrates on making route verification very lightweight, while route protection stays somewhat costly. This mechanism is very appropriate for agents going through very busy hosts (these could deny service if verification was too time-consuming).
- The mechanism based on Merkle trees aims at reducing the computational work carried out by the agent owner to protect a route. This mechanism is especially suitable when the agent owner is the bottleneck, as it might happen for very busy agent owners who must launch large number of mobile agents each on a different route.

Both mechanisms presented here can be extended to flexible itineraries in the sense of [19,11]. Since each route step is independently coded (either as a hash collision or as a tree leaf), we could think of including several alternative paths going from a host H_i to another host H_j along the route. To do this, code all steps of the alternative paths. When choosing one particular path at a junction, the information (including methods) related to steps in alternative paths does not need to be conveyed by the agent to the next hosts in the route. Coding all alternative path steps substantially increases the route protection work in the scheme based on hash collisions (a new collision is needed for each route step). For the Merkle tree scheme, the computational overhead of including alternative paths is negligible: adding more leaves to the Merkle tree is fast and does not even significantly increase the length of ver-paths.

Acknowledgments

This work was partly supported by the Spanish CICYT under project no. TEL98-0699-C02-02. Thanks go to Francesc Sebé and Marc Alba for reading an earlier draft of this paper and helping to work out some examples.

References

1. S. Y. Bennet, "A sanctuary for mobile agents", in *Foundations for Secure Mobile Code Workshop*. Monterey CA: DARPA, 1997, pp. 21-27.
2. J. Borrell, S. Robles, J. Serra and A. Riera, "Securing the itinerary of mobile agents through a non-repudiation protocol", in *33rd Annual IEEE Intl. Carnahan Conference on Security Technology*. Piscataway NJ: IEEE, 1999, pp. 461-464.
3. J. Domingo-Ferrer, "A new privacy homomorphism and applications", *Information Processing Letters*, vol. 60, no. 5, Dec. 1996, pp. 277-282.
4. J. Domingo-Ferrer, M. Alba and F. Sebé, "Asynchronous large-scale certification based on certificate verification trees", in *IFIP Communications and Multimedia Security'2001*, Boston MA: Kluwer, 2000, pp. 185-196.

5. D. Dyer, "Java decompilers compared", June 1997.
`http://www.javaworld.com/javaworld/jw-07-1997/jw-decompilers.html`
6. I. Gassko, P. S. Gemmell and P. MacKenzie, "Efficient and fresh certification", in *Public Key Cryptography'2000*, LNCS 1751. Berlin: Springer-Verlag, 2000, pp. 342-353.
7. F. Hohl, "Time limited blackbox security: Protecting mobile agents from malicious hosts", in *Mobile Agents and Security*, LNCS 1419. Berlin: Springer-Verlag, 1998, pp. 92-113.
8. C. Jutla and M. Yung, "PayTree: " Amortized-signature" for flexible micropayments", in *Second USENIX Workshop on Electronic Commerce*, Oakland CA, Nov. 1996.
9. D. Libes, *Obfuscated* C *and Other Mysteries*, New York: Wiley, 1993.
10. C. Meadows, "Detecting attacks on mobile agents", in *Foundations for Secure Mobile Code Workshop*. Monterey CA: DARPA, 1997, pp. 50-56.
11. J. Mir, "Protecting flexible routes of mobile agents", private communication, 2001.
12. National Bureau of Standards, "Data Encryption Standard", FIPS Publication 46, Washington DC, 1977.
13. U. S. National Institute of Standards and Technology, *Secure Hash Standard*, FIPS PUB 180-1, 1995.
`http://csrc.ncsl.nist.gov/fips/fip180-1.txt`
14. R.L. Rivest and S. Dusse, "RFC 1321: The MD5 message-digest algorithm", Internet Activities Board, Apr. 1992.
15. R.L. Rivest and A. Shamir, "PayWord and MicroMint: Two simple micropayment schemes", Technical report, MIT Laboratory for Computer Science, Nov. 1995.
16. T. Sander and C.F. Tschudin, "Protecting mobile agent against malicious hosts", in *Mobile Agents and Security*, LNCS 1419. Berlin: Springer-Verlag, 1998, pp. 44-60.
17. K. B. Sriram, "Hashjava - a java applet obfuscator", July 1997.
`http://www.sbktech.org/hashjava.html`
18. J. P. Stern, G. Hachez, F. Koeune and J.-J. Quisquater, "Robust object watermarking: application to code", in *Information Hiding'99*, LNCS 1768. Berlin: Springer-Verlag, 2000, pp. 368-378.
19. M. Strasser, K. Rothermel and C. Maihöfer, "Providing reliable agents for electronic commerce", in *TREC'98*, LNCS 1402. Berlin: Springer-Verlag, 1998, pp. 241-253.
20. N. van Someren, "The practical problems of implementing MicroMint", in *Financial Cryptography'2001*, February 2001 (proceedings still to appear). Available from author `nicko@ncipher.com`.
21. G. Vigna, "Cryptographic traces for mobile agents", in *Mobile Agents and Security*, LNCS 1419. Berlin: Springer-Verlag, 1998, pp. 137-153.
22. D. Westhoff, M. Schneider, C. Unger and F. Kaderali, "Methods for protecting a mobile agent's route", in *Information Security-ISW'99*, LNCS 1729. Berlin: Springer-Verlag, 1999, pp. 57-71.

Appendix. Hash Functions and the MicroMint System

Hash functions are widely used in cryptography to perform digital signatures. A hash function, sometimes called message digest, is a function F that takes a variable-length input string x and converts it to a fixed-length output string y.

A hash function F is said to be computationally one-way if it is easy and fast to compute the hash $y = F(x)$ but, given y, it is hard to compute x such that $y = F(x)$. A hash function F is said to be collision-free if it is hard to find two x, x' such that $x \neq x'$ and $F(x) = F(x')$.

MicroMint [15] is a system for low-value payments (micropayments) where the coins are not digitally signed by the bank (or minting organization). Instead, the bank computes coins as k-way collisions, *i.e.* k values whose hash images collide for a prespecified one-way hash function F. More formally, a coin is represented by $(x_1, \cdots, x_k)$ such that

$$F(x_1) = F(x_2) = \cdots = F(x_k) = y$$

where y is an m-bit string. Increasing k has the dual effect of increasing the computation needed to find the first collision, and also accelerating the rate of collision generation once the first collision has been found; actually, $k = 4$ is recommended by the MicroMint authors.

The verifier of a coin (the payee) accepts it as valid if it is a k-way collision and the t high-order bits of y match a value z advertised by the bank at the start of the current validity period. Thus verification only requires computing k hash values.