Empowering Mobile Software Agents

Volker Roth*

Fraunhofer Institute for Computer Graphics
Dept. Security Technology
Fraunhoferstr. 5, 64283 Darmstadt, Germany
vroth@igd.fhg.de

Currently at
International Computer Science Institute
University of California, Berkeley
1947 Center Street, Suite 600, Berkeley CA 94704-1198, USA
roth@icir.org

Abstract. Recent work has shown that several cryptographic protocols for the protection of free-roaming mobile agents are vulnerable by means of protocol interleaving attacks. This paper presents equivalent protocols meant to be robust against this type of attack. Moreover, it describes the required processes and data structures at a level of detail that can be translated to an implementation in a straightforward way. Our aim is to demonstrate how cryptographic processing can be implemented transparently for agent programmers, thereby reducing the risks of human error in (secure) mobile agent programming.

Keywords: mobile agent security, malicious host, protocol interleaving attack, applied cryptography.

1 Introduction

One class of mechanisms for the protection of data in free-roaming mobile software agents against malicious hosts (as opposed to e.g., protecting the confidentiality of an agent's computation) is based on cryptographic protocols. In general, the objectives are twofold: first, an agent carries confidential information that is revealed only while the agent is on a trusted host, and second, the agent transports partial results back to its origin in a way that assures the integrity (and optionally the confidentiality) of the partial results. Furthermore, the owner of the agent shall be able to derive the identity of the host on which a given partial result was acquired.

Recently, several of these protocols were shown to be vulnerable to *interleaving attacks* [1]. An interleaving attack [2, §10.5] is "an impersonation or other deception involving selective combination of information from one or more previous or simultaneously ongoing protocol executions *(parallel sessions)*, including possible origination

* This research was supported by the DAAD *(German Academic Exchange Service).* Views and conclusions contained in this document are those of the author and do not necessarily represent the official opinion, either expressed or implied, by the DAAD.

of one or more protocol executions by an adversary itself." Figure 1 illustrates how interleaving attacks may be mounted in the domain of mobile agents: the adversary receives an agent, and copies protocol data back and forth between this agent and agents she sent herself.

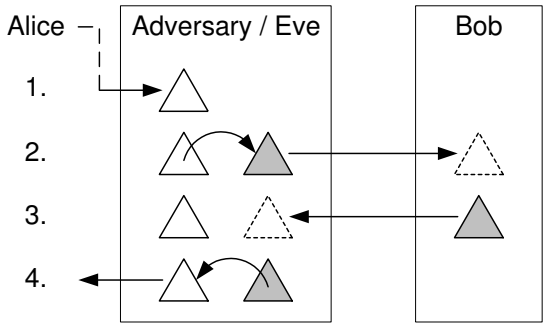


Fig. 1. Basic scheme of attacks we mount against various protocols. Triangles denote agents. Triangles shaded in gray denote agents created by the adversary Eve.

In this paper, we put forward protocols that are designed to be robust against this type of attack.[1] We apply techniques from [1, 3, 4] and define their application at a level of abstraction that can be translated to an implementation in a straightforward way. We have the following additional objectives:

- Cryptographic protocol data shall be produced only if a partial result is generated.
- The implementation shall be independent of a particular agent programming language.
- There shall be a security policy that is simple and easy to understand.
- Cryptographic processing shall be transparent for agent programmers.

The last two objectives take into account the potential for human error. Security is often added as an afterthought rather than at design time. On top of that, it is customary that (agent) programmers need to "get the application going" within a tight schedule, and security features are likely sacrificed close to a deadline. Hence, it is prudent not to leave the management and implementation of cryptographic functions up to agent programmers – a task for which they are hardly ever trained in the first place.

We chose to represent a mobile agent as a ZIP [5] archive. Our previous approach [3] was based on the JAR [6] format. As it turned out, that choice offered little benefits to us and complicated the implementation. JAR Manifest sections do not require a canonical ordering and may contain additional (even custom) attributes. Therefore, said sections would require caching and complicated re-ordering so that automatic regeneration of

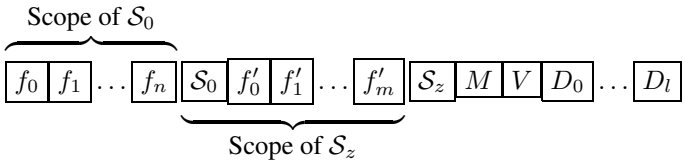
[1] We do not take into account interleaving attacks based on malicious modification of an agent's mutable state. This needs to be addressed by other means.

Manifests does not invalidate e.g., digital signatures that are computed on them. For the sake of simplicity, we changed to a format that has a more concise set of features, contains less redundancy, and is easier to parse and recreate in a cryptographically unambiguous way. Our current mobile agent structure still bears strong resemblance to a JAR. In particular, we keep most of the JAR terminology, although the structures to which we refer have a different format. Please note, though, that the archive format is purely an implementation choice. The abstract mechanisms that we describe can be applied to other representations of a mobile agent as well e.g., a representation based on XML [7].

The remainder of the paper is structured as follows. An overview of the mechanisms we describe is given in sect. 2. Section 3 gives the definitions that we use throughout the paper. The details of our work are at the focus of sect. 4. A subset of the functionality that we describe in sect. 4 was implemented and sect. 5 summarizes the results of our performance measurements. We describe related work in sect. 6, and give our conclusions in sect. 7.

2 Overview and Security Rationale

We implemented security funtions on top of the ZIP format by means of meta files with a well-defined cryptographic interpretation. A schematic overview is given below:



Files f_i are static files, and f'_i stands for mutable files in the agent. The static files are signed by the agent's owner, which yields signature $\mathcal{S}_0$. The message digest $h(\mathcal{S}_0)$ uniquely identifies the protocol run that is represented by that particular agent instance. Each host E_z signs the mutable files including the file with owner's signature, which yields signature $\mathcal{S}_z$, thereby replacing the signature of the previous host E_{z-1}. We assume that a signer's identity can be derived from the signature. Signatures are computed on Manifest sections of the files. The Manifest sections are kept current in file M.

Whenever the agent produces a partial result, that result is encrypted for the entity who produced $\mathcal{S}_0$, and the ciphertext is added to the mutable files. If the previous host added a partial result, then that host's signature $\mathcal{S}_{z-1}$ is saved in the agent as well. Additionally, the current host adds the difference D_l between the previous version of M and the version it signs so that the input to the verification of $\mathcal{S}_{z-1}$ can be recovered later. If the previous host did not add a partial result then D_l is merged with D_{l-1}.

Signature $\mathcal{S}_z$ serves a dual purpose. It protects the overall integrity of the agent during transport and binds partial results to it for later verification. The indirection by means of the M file and the differential Manifests D_i assures that saved signatures can be verified on a set of files where some files are allowed to change and others are not. Thus, the chain of (encrypted) partial results $\mathcal{P}_1, \ldots, \mathcal{P}_n$ is protected qualitatively as

given below:

$$\{h(\mathcal{S}_0), \ldots \{h(\mathcal{S}_0), \{h(\mathcal{S}_0), \mathcal{P}_1\}_{S_1^{-1}}, \mathcal{P}_2\}_{S_2^{-1}}, \ldots, \mathcal{P}_n\}_{S_n^{-1}}$$

Signing $h(\mathcal{S}_0)$ along with $\mathcal{P}_i$ prevents that a $\mathcal{P}_i$ is used in a context of an agent other than the agent with signature $\mathcal{S}_0$. Without further precautions, this scheme still allows arbitrary truncation of partial results, where a malicious host strips off an *arbitrary number* of outer signatures and grows a fake stem. This risk is reduced by means of a variant of *Partial Result Authentication Codes* (PRAC) [4] and encryption.

Each encrypted partial result $\mathcal{P}_i$ is equivalent to $(\{m_i\}_{N_i}, \{N_i\}_{K_0})$ where K_0 is the public encryption key of the entity who produced $\mathcal{S}_0$, and m_i is the plaintext of the partial result, and N_i is a random secret key chosen by the host that produced $\mathcal{P}_i$. Key N_i is reused as input to a hash function that is computed on N_i, S_i, and the previous partial result authentication code V_{i-1}. The new value $V_i = h(N_i \mid\mid V_{i-1} \mid\mid S_i)$ replaces the previous value V_{i-1} in the agent. This construction serves two purposes:

- The PRAC can be predicted neither into the past nor into the future based on the current value. The agent's owner has access to the secret keys and thus can validate the sequence of computations that led to the final value of the PRAC.
- The input of S_i and N_i into the PRAC computation generally prevents that the adversary signs and adds cipher texts for which he does not know the corresponding secret key (e.g., cipher texts that the adversary collected elsewhere).

These measures particularly prevent that the adversary strips off the *last* signature in the (truncated) chain of encrypted partial results, adds a signature of his own, and thus claims to be the origin of the encrypted partial result (albeit not knowing the plain text).

Adversaries may still truncate the partial results at a position for which they know a valid PRAC. Particularly, any host on a loop in the agent's itinerary can replace the agent by a previously saved copy, thus undoing all state changes of that agent between consecutive visits. There does not seem to be a way around this problem unless there is a notion of agent *freshness*, or an external state is used (e.g., based on co-operating agents [8]).

Readers may notice that at this point we did not include a forward reference to the next hop of the agent into the PRAC computation, as is occasionally proposed in related work [9–11]. The reasons for this decision go beyond the scope of this paper and will probably be described elsewhere.

We describe our security mechanisms in greater detail in sect. 4. For the sake of completeness, this includes mechanisms for binding encrypted data to a mobile agent that is made available to it only on trusted hosts. We covered that material already in [3, 1], see also def. 7 for details.

3 Definitions and Data Types

Table 1 gives an overview over the meta-structure of a mobile agent. Almost all data types are defined in terms of ASN.1 [12], and instances thereof are encoded by means

of the *Distinguished Encoding Rules* (DER) which are defined in [13]. Each data type is associated with a file extension, the mapping is given in tab. 1. For the sake of brevity and precision, we also define some data types in an abstract way, including the operations that we require on these types.

We will write encryption of some *plaintext* into a *ciphertext* symbolically as $c = \{m\}_K$, where K is the *key* being used. A digital signature will be written as an encryption with a private signing key S^{-1}. We will write $S^{-1}(m)$ when we refer to the bare signature rather than the union of the signature and the signed data. We assume that the identity of the signer can be extracted from her signature. A cryptographic hash of some input will be written $h(m)$. Unless noted otherwise, we assume that h is *preimage resistant* and *collision resistant* [2, §9.2.2].

Definition 1 (Collection). *A* collection $\mathcal{C} = \{(s_i, v_i) \mid i = 0, \ldots, n\}$ *is an ordered set of key/value pairs where s_i (the key) is a name and v_i is the value. Each name occurs at most once in $\mathcal{C}$, so $s_i = s_j \Leftrightarrow i = j$. The sort order is the lexicographic order of the keys. Let* zip *and* unzip *be two functions so that for a given collection $\mathcal{C}$:*

$$\mathrm{zip}(\mathcal{C}) \quad \textit{is a value, and} \quad \mathrm{unzip}(\mathrm{zip}(\mathcal{C})) = \mathcal{C}$$

We assume that zip *adds some redundancy to the value (to be verified by* unzip*) so that the probability that a random value is accepted by* unzip *is negligible.*

Below, we define additional operations on collections. The first operation selects all pairs from a collection $\mathcal{C}$ whose keys have a given prefix p (the prefix is removed from the keys). The second operation prepends a given prefix to the keys of a collection. The third operation removes values from the collection whose keys have a given prefix. The fourth operation returns the subset of a selection based on a list of keys. The fifth operation returns the keys of a given collection. The sixth operation returns the value that is associated with a given key.

$$\begin{aligned}
\mathrm{children}(\mathcal{C}, p) &= \{(s, v) \mid (p \circ s, v) \in \mathcal{C}\} \\
\mathrm{move}(\mathcal{C}, p) &= \{(p \circ s, v) \mid (s, v) \in \mathcal{C}\} \\
\mathrm{remove}(\mathcal{C}, p) &= \{(s, v) \in \mathcal{C} \mid p \circ s' \neq s\} \\
\mathrm{select}(\mathcal{C}, L) &= \{(s, v) \in \mathcal{C} \mid s \in L\} \\
\mathrm{keys}(\mathcal{C}) &= \{s \mid \exists v : (s, v) \in \mathcal{C}\} \\
\mathrm{value}(\mathcal{C}, s) &= \begin{cases} v & \textit{if } (s, v) \in \mathcal{C} \\ \textit{error} & \textit{else} \end{cases}
\end{aligned}$$

where '$\circ$' is the string concatenation. We furthermore define a binary copy operator '$\sqcup$' so that for two collections $\mathcal{C}_1$ and $\mathcal{C}_2$:

$$\mathcal{C}_1 \sqcup \mathcal{C}_2 = \{(s, v) \in \mathcal{C}_1 \mid s \notin \mathrm{keys}(\mathcal{C}_2)\} \cup \mathcal{C}_2$$

Thus, $\mathcal{C}_2$ "overwrites" the values in $\mathcal{C}_1$.

Definition 2 (Aliases). *A mobile agent $\mathcal{A}$ is a collection. A* Manifest *is a collection. The elements of a Manifest are also called* Manifest sections.

Definition 3 (Differential Manifest). *Let M_1, M_2 be two Manifests. We define a difference operator* diff *and a* differential Manifest *D as given below:*

$$\text{retain} = M_1 \backslash M_2 \tag{1}$$

$$\text{delete} = \text{keys}(M_2) \backslash \text{keys}(M_1) \tag{2}$$

$$\Rightarrow D = \text{diff}(M_2, M_1) = (\text{retain}, \text{delete}) \tag{3}$$

We define the substraction *of a differential Manifest D from a Manifest M as follows:*

$$\text{sub}(M, D) = \{(s, v) \in M \mid s \notin \text{keys}(\text{retain}) \cup \text{delete}\} \cup \text{retain} \tag{4}$$

Differential Manifests represent a series of incremental changes of a Manifest in a compact way. This is used in sects. 4.6 and 4.7 to compute and verify cryptographic "checkpoints" of an agent's state.

Theorem 1 (Manifest reconstruction). *Let M_2, M_1 be two Manifests with identical message digest algorithm identifiers. Then the following equation holds*

$$\text{sub}(M_2, \text{diff}(M_2, M_1)) = M_1$$

Proof of Theorem 1. Follows by a simple argument over defs. (1) – (4).

Definition 4 (Meta, regular, static, and mutable files). *The* meta files *of a given mobile agent $\mathcal{A}$ are the elements whose names start with "META-INF/". The* regular files $\text{regular}(\mathcal{A})$ *of $\mathcal{A}$ are all elements that are not meta files. The* static files *of the agent are the regular files marked • in tab. 1, plus any additional files that shall not be modified during the agent's lifetime. The* mutable files *are all regular files that are not static files.*

Definition 5 (Entities, groups, and associated key pairs). *Let $\mathcal{E} = \{E_0, \ldots, E_n\}$ be a set of entities. Any nonempty subset $G \subseteq \mathcal{E}$ is a* group. *Let $(S_i, S_i^{-1}), (K_i, K_i^{-1})$ be key pairs of E_i for use with a digital signature algorithm and a public key encryption scheme. For simplicity, we do not distinguish between entities and their identities. The difference should be clear from the context.*

Definition 6 (Encrypted archives and seals). *Let $G = \{E_0, \ldots, E_n\}$ be a group of entities with corresponding key pairs $K_i, K_i^{-1}, 0 \leq i \leq n$ that are suitable for encryption. Let N be a randomly chosen group key, and let v be a value. Then $\{v\}_N$ is an encrypted archive and*

$$P = \{(E_0, \{N\}_{K_0}), \ldots, (E_n, \{N\}_{K_n})\}$$

is a collection where the keys are the identities of the group members and the values are the group keys encrypted with the keys of the group members. P is also called a seal, *and the group members are called* authorized recipients.

Definition 7 (Install file). *Let $\mathcal{A}$ be a mobile agent, E_0 be the agent's* owner, *and let S_0 be the owner's verification key. Let $G_0, \ldots, G_m$ be groups of authorized recipients.*

Furthermore, let $N_0, \ldots, N_m$ be a list of randomly chosen group keys where N_i corresponds to group G_i. Then an install file *I is defined as:*

$$\begin{aligned} \text{acl} &= \{(p_i, j_i) \mid i = 0, \ldots, n\} \\ \text{proofs} &= \{(k, h(N_k \mid\mid S_0)) \mid k = 0, \ldots, m\} \\ \Rightarrow I &= (\text{acl}, \text{proofs}) \end{aligned}$$

where $j_i \in \{0, \ldots, m\}$ and all p_i are prefix-free. The first set of I is the access control policy *of $\mathcal{A}$ and its interpretation is as follows: p_i is the path name prefix of a collection of files in $\mathcal{A}$, k is the identifier of the encrypted archive in which that file collection is kept, and j_i is the index of the group whose group key the archive is encrypted with. The second set of I contains per group* proofs of knowledge *of the group key. They are used in order to ascertain that a given encrypted archive actually belongs to a given mobile agent.*

Definition 8 (Useful macros). *Let X be the symbol of an abstract data type as listed in tab. 1. Then $\mathrm{name}(X)$ is the corresponding path name. In order to render our descriptions of the algorithms more compact, we define the "algorithm snippets" given below:*

$$\begin{aligned} \mathrm{store}(\mathcal{A}, X_0, \ldots, X_n) &= \left[\mathcal{A} \leftarrow \mathcal{A} \sqcup \bigcup_{i=0}^{n} \{(\mathrm{name}(X_i), X_i)\}\right] \\ \mathrm{load}(\mathcal{A}, X_0, \ldots, X_n) &= \begin{bmatrix} X_0 \leftarrow \mathrm{value}(\mathcal{A}, \mathrm{name}(X_0)) \\ \vdots \\ X_n \leftarrow \mathrm{value}(\mathcal{A}, \mathrm{name}(X_n)) \end{bmatrix} \\ \mathrm{check}(\mathcal{C}, \mathcal{C}', X_0, \ldots, X_n) &= \bigwedge_{i=0}^{n} (\mathrm{value}(\mathcal{C}, \mathrm{name}(X_i)) = \mathrm{value}(\mathcal{C}', \mathrm{name}(X_i))) \end{aligned}$$

where $\mathcal{A}$ is an agent, $\mathcal{C}, \mathcal{C}'$ are two collections, and $X_0, \ldots, X_n$ are abstract data types. In other words, store *writes files to the meta-structure of agent $\mathcal{A}$,* load *initializes abstract data types from data that is stored in agent $\mathcal{A}$, and* check *compares corresponding sections of two collections for equality. Encoding and decoding is done according to the file types given in tab. 1.*

4 Processes

In this section we define the cryptographic transformations of a mobile agent. These transformations are meant to implement the following security policy:

1. On creation of his mobile agent $\mathcal{A}$, the owner E_0 defines a path p and a group G of authorized recipients. If $\mathcal{A}$ is at some $E \in G$ then the files in path p are made available to $\mathcal{A}$. Please note that the opposite is not true. If some file is available in path p then this does not necessarily mean that the agent is at some $E \in G$. Extend this to n paths and corresponding groups.

path/name		*mark*	*symbol*
`META-INF/`	`manifest.mf`	$\times$	M
	`owner.sf`	$\times$	L_0
	`owner.p7s`	$\circ$	$\mathcal{S}_0$
	`sender.p7s`	$\times$	$\mathcal{S}_i, i > 0$
	`prac.bin`	$\times$	V
	i`.dmf`	$\times$	D_i
`SEAL-INF/`	`owner.cert`	$\bullet$	K_0
	`install.cfg`	$\bullet$	I
	i`.p7m`	$\bullet$	P_i
	i`.ear`	$\circ$	A_i
`VAR-INF/`	i`.ear`	$\circ$	R_i
	i`.p7m`	$\circ$	Q_i
	i`.p7s`	$\circ$	T_i
name`.class`		$\bullet$	

Extension	Formatting
`mf`	ManifestFile‡
`sf`	SignatureFile‡
`dmf`	ManifestSections‡
`p7s`	PKCS#7 SignedData†
`p7m`	PKCS#7 EnvelopedData†
`ear`	raw encrypted ZIP file
`cfg`	InstallFile‡
`bin`	binary data
`cert`	X.509v3 Certificate

Table 1. The meta-structure of a mobile agent (left table). The symbols refer to the definitions of abstract data types to which the files correspond. The *'i'* in file names is replaced by the string value of the index of the corresponding symbol. Files marked $\bullet$ are *static* files and must be signed by the owner of the agent, files marked $\circ$ are signed by the agent's sender, and files marked $\times$ are not signed. File extensions (right table) denote content that is encoded according to distinct formatting rules; the mapping from extensions to formatting rules is given above. Data structures marked with a † are wrapped into a PKCS#7 ContentInfo. Data structures marked with a ‡ are defined in this paper.

2. Let p be the path "spool/". All files in that path are partial results acquired at the current host E_c. These results are transported back to the agent's owner E_0 in confidentiality and integrity. Whenever $\mathcal{A}$ resumes execution at some host E_c it finds path "spool/" empty.
3. Upon return of $\mathcal{A}$ to E_0, the agent finds the n'th partial result in path "results/*n*/". Please note that again the opposite is not true. If an agent finds partial results then it need not necessarily be at the host of its owner.

This security policy gives meaning to certain paths in a mobile agent. It is enforced by the cryptographic processes that we describe below. Dealing with the security policy, rather than dealing with cryptographic detail, simplifies the task of agent programmers.

4.1 Inflation and Deflation

Several processing steps of a mobile agent require that as a subprocess:

- cipher text is decrypted and installed in the meta-structure of the agent;
- clear text is encrypted into cipher text, and deleted subsequently from the meta--structure.

We refer to these processes as *inflation* and *deflation* of the agent. Both processes take as their arguments the encrypted archives of type either A or R, the paths where the

clear text is installed, and the secret keys required to encrypt or decrypt the clear text.

$$\begin{aligned} \text{acl} &= \{(p_i, j_i) \mid i = 0, \ldots, n\}, j_i \in \{0, \ldots, m\} \\ \text{secrets} &= \{(i, N_i) \mid i = 0, \ldots, m\} \end{aligned}$$

Both algorithms are given below, where X is a placeholder for an encrypted archive of type either A or R:

$\underline{\text{deflate}_X(\mathcal{A}, \text{acl}, \text{secrets})}$

Require: Agent $\mathcal{A}$, acl, secrets
1: **for all** $(p_i, j_i) \in$ acl **do**
2: $N \leftarrow \text{value}(\text{secrets}, j_i)$
3: $X_i \leftarrow \{\text{zip}(\text{children}(\mathcal{A}, p_i))\}_N$
4: $\mathcal{A} \leftarrow \text{remove}(\mathcal{A}, p_i)$
5: **end for**
6: $\text{store}(\mathcal{A}, X_0, \ldots, X_n)$

$\underline{\text{inflate}_X(\mathcal{A}, \text{acl}, \text{secrets})}$

Require: Agent $\mathcal{A}$, acl, secrets
1: **for all** $(p_i, j_i) \in$ acl **do**
2: **if** $\text{children}(\mathcal{A}, p_i) \neq \emptyset$ **then**
3: **error** {Agent not clean.}
4: **end if**
5: **if** $j_i \in \text{keys}(\text{secrets})$ **then**
6: $\text{load}(\mathcal{A}, X_i)$
7: $N \leftarrow \text{value}(\text{secrets}, j_i)$
8: $\mathcal{C} \leftarrow \text{unzip}(\{X_i\}_N)$
9: $\mathcal{A} \leftarrow \mathcal{A} \sqcup \text{move}(\mathcal{C}, p_i)$
10: **end if**
11: **end for**

The assertion $\text{children}(\mathcal{A}, p_i) \neq \emptyset$, which is verified in line 2 of the inflation algorithm, prevents unauthorized recipients from merging data into the agent. Assume that the assertion is not verified. Then an unauthorized recipient may add $(p_i \circ s, x)$ to the agent for some s, x. During inflation at an authorized recipient of A_i, that data would be merged with the plain text and subsequently be copied into A_i by the deflation algorithm.

4.2 Instantiation of a Mobile Agent

Each mobile agent has an owner E_0 with two different key pairs S_0, S_0^{-1} and K_0, K_0^{-1} for signing and encryption. The owner creates a file collection $\mathcal{A}$ and fills it with the data that is required by the mobile agent e.g., the code, itinerary, and serialized object graph of the agent. This data shall not interfere with the definition of special files in tab. 1. E_0 defines the groups $G_0, \ldots, G_m$ of authorized recipients and the access control policy $\text{acl} = \{(p_i, j_i) \mid i = 0, \ldots, n\}, j_i \in \{0, \ldots, m\}$. Then, E_0 computes the seals and the install file, and deflates the agent.

Require: Agent $\mathcal{A}$, groups, S_0
1: secrets $\leftarrow \emptyset$
2: proofs $\leftarrow \emptyset$
3: **for** $j = 0, \ldots, m$ **do**
4: Compute random key N_j
5: secrets $\leftarrow$ secrets $\cup \{(j, N_j)\}$
6: proofs $\leftarrow$ proofs $\cup \{(j, h(N_j \mid\mid S_0))\}$

7: Compute P_j based on G_j, N_j according to def. 6
8: **end for**
9: $I \leftarrow \{\text{acl}, \text{proofs}\}$
10: $\text{store}(\mathcal{A}, I, P_0, \ldots, P_m)$
11: $\text{deflate}_A(\mathcal{A}, \text{acl}, \text{secrets})$

Next, E_0 decides which files of the agent shall be static. All program code of the agent must be static, as well as all files marked as being static in tab. 1. Let L_0 be the names of these files. E_0 computes the Manifest M of his agent, and signs it. The signature is saved to the agent, as well as a secret initial partial result authentication code which is chosen randomly.

Require: Agent $\mathcal{A}$, list of static files L_0, private key S_0^{-1}
1: $M \leftarrow \{(s, h(v)) \mid (s, v) \in \text{regular}(\mathcal{A})\}$
2: $\mathcal{S}_0 \leftarrow S_0^{-1}(\text{select}(M, L_0), t_0), t_0$
3: Compute random number V_0
4: $\text{store}(\mathcal{A}, \mathcal{S}_0, L_0, V_0)$

The owner of the agent is also its first sender. Therefore, E_0 also signs the entire agent. More precisely, E_0 updates the Manifest so that sections for the added files are included, and signs the sections of all mutable files as well as a digest of $\mathcal{S}_0$. This binds the mutable files of $\mathcal{A}$ to the agent's static kernel, which prevents protocol interleaving attacks of the form described in [1].

Require: Agent $\mathcal{A}$, list of static files L_0, private key S_0^{-1}, signature $\mathcal{S}_0$
1: $L \leftarrow \text{keys}(\text{regular}(\mathcal{A})) \backslash L_0$
2: $M \leftarrow \{(s, h(v)) \mid (s, v) \in \text{regular}(\mathcal{A})\}$
3: $\mathcal{S}_1 \leftarrow S_0^{-1}(\text{select}(M, L), h(\mathcal{S}_0), t_1), t_1$
4: $\text{store}(\mathcal{A}, M, \mathcal{S}_1)$

E_0 stores $(h(\mathcal{S}_0), V_0)$ for the purpose of verifying $\mathcal{A}$ after its return. This completes the instantiation of the agent. $\text{zip}(\mathcal{A})$ is now sent to the first host. We do not consider replay of $\mathcal{A}$ to E_0. This has to be detected by E_0 by separate means.

4.3 Security Invariants of Migration

At each miration of a mobile agent, certain *invariants* must hold for the agent to be deemed valid. It is the responsibility of each host to verify these invariants upon receiving the agent. A failed verification means that the agent is invalid, and that it is not admitted to the host. Let $\mathcal{A}$ be an agent, let M be the Manifest as read from $\mathcal{A}$, and let I be the installation file as read from $\mathcal{A}$.

Invariant 1 *All regular files in $\mathcal{A}$ must have a valid Manifest section with a matching message digest.*

Invariant 2 *All Manifest sections must be signed by either the owner or the sender of the agent. The Manifest sections signed by the owner are the sections of the static files.*

Invariant 3 *The sender must have signed the owner's signature along with the Manifest sections of the mutable files.*

Invariant 4 *No two mobile agents with the same owner signature are admitted to the agent server.*

Invariant 5 *Whenever the current host E is an authorized recipient in the seal P_j of a group j there must be a proof of knowledge (j, x) in the install file of $\mathcal{A}$ so that $x = h(N_j \mid\mid S_0)$ where N_j is the secret group key and S_0 is the public key that verifies the owner's signature. Since E is an authorized recipient, E can recover N_j from P_j.*

Invariant 6 *Let p be a path prefix (see def. 7) in I then there must not be a file in $\mathcal{A}$ whose name has p as its prefix.*

Invariant 7 *Code is loaded only from files whose Manifest sections are signed by the owner of the agent (files can also be loaded from remote code sources as long as the digest matches the one in the corresponding Manifest section).*

4.4 Processing of an Incoming Agent

Let E be a host and let $\mathcal{A}$ be a mobile agent that is received by E. E first checks the invariants 1-4. Then it determines the groups of authorized recipients to which it belongs and checks invariant 5 as given below:

Require: Agent $\mathcal{A}$, public key S_0 of the owner E_0 of $\mathcal{A}$
1: secrets $\leftarrow \emptyset$
2: $\mathrm{load}(\mathcal{A}, I, P_0, \ldots, P_n)$
3: **for all** $(j, x) \in$ proofs **do**
4: **if** $E \in \mathrm{keys}(P_j)$ **then**
5: $N_j \leftarrow \{\mathrm{value}(P_j, E)\}_{K_E^{-1}}$
6: **if** $x \neq h(N_j \mid\mid S_0)$ **then**
7: **error** {Ciphertext not owned by agent, invariant 5 violated}
8: **end if**
9: secrets $\leftarrow$ secrets $\cup \{(j, N_j)\}$
10: **end if**
11: **end for**
12: $\mathrm{inflate}_A(\mathcal{A}, \mathrm{acl}, \mathrm{secrets})$

This completes the verification and the setup of incoming agents. Based on the established signing keys and host-specific policies the agent is either authorized and executed, or it is rejected.

4.5 Execution of a Mobile Agent

For all code that is loaded into the code segment of the agent, invariant 7 must be assured. The agent is provided read access to its meta-structure by means of a suitable abstraction, and write access to all files in its meta-structure but those listed in tab. 1.

4.6 Processing of an Outgoing Agent

Let the current host E_z be the z'th sender of the agent with $z > 1$. The mobile agent $\mathcal{A}$ might have modified its meta-structure in a way that violates invariant 6. Therefore, this invariant must be assured, and all files that are subject to the agent's access control policy must be re-encrypted with the group key.

Require: Agent $\mathcal{A}$, public key S_0 of the owner E_0 of $\mathcal{A}$
1: Compute acl and secrets as in sect. 4.4 (do not $\mathrm{inflate}$ $\mathcal{A}$ again)
2: Deflate $\mathcal{A}$ based on acl, secrets, and archive type A as described in sect. 4.1

The next step is to seal off any partial results the agent wishes to take home in confidentiality, if there are any. The agent must store these results in a spooling area in the meta-structure which is reserved for this purpose. First, E has to find the smallest unused index in the sequence of encrypted archives. Then E creates a seal with the owner of the agent as the authorized recipient, and an encrypted archive with the contents of the spool area. The plain text is deleted subsequently.

Require: Agent $\mathcal{A}$, agent owner identity E_0, public key S_z of current host
1: $n \leftarrow \min\{i \in \mathbb{N}_0 \mid \mathrm{name}(Q_i) \notin \mathrm{keys}(\mathcal{A})\}$
2: $\mathrm{load}(\mathcal{A}, K_0, V)$
3: **if** $\mathrm{children}(\mathcal{A}, \text{“spool/”}) \neq \emptyset$ **then**
4: Compute random key N
5: $Q_n \leftarrow \{(E_0, \{N\}_{K_0})\}$
6: $V \leftarrow h(V \mid\mid N \mid\mid S_z)$
7: $\mathrm{store}(\mathcal{A}, Q_n, V)$
8: $\mathrm{deflate}_R(\mathcal{A}, \{(\text{“spool/”}, n)\}, \{(n, N)\})$
9: **end if**

If any encrypted archive of type R was added by the previous host then this fact shall be recorded by storing the signature of the agent's most recent sender in the agent. The chain of signatures is verified by the agent's owner after the agent's return. Its purpose is not to provide non-repudiation of origin. Rather the purpose is to testify that the partial results have not been tampered with, and on which host the results were accumulated.

10: $\mathrm{load}(M, L_0, \mathcal{S}_{z-1})$
11: **if** $n > 0$ **then**
12: **if** $\mathrm{name}(T_{n-1}) \in \mathrm{keys}(\mathcal{A})$ **then**
13: $\mathrm{load}(\mathcal{A}, D_{n-1})$
14: $M \leftarrow \mathrm{sub}(M, D_{n-1})$
15: **else**
16: $T_{n-1} \leftarrow \mathcal{S}_{z-1}$
17: $\mathrm{store}(\mathcal{A}, T_{n-1})$
18: **end if**
19: $M' \leftarrow \{(s, h(v) \mid (s, v) \in \mathrm{regular}(\mathcal{A})\}$
20: $D_{n-1} \leftarrow \mathrm{diff}(M', M)$
21: $\mathrm{store}(\mathcal{A}, D_{n-1})$

22: **end if**
23: $L \leftarrow \text{keys}(\text{regular}(\mathcal{A})) \backslash L_0$
24: $\mathcal{S}_z \leftarrow S_z^{-1}(\text{select}(M, L), h(\mathcal{S}_0), t_z), t_z$
25: $\text{store}(\mathcal{A}, M, \mathcal{S}_z)$

The purpose of the algorithm given above is to compress a record of changes to the meta-structure that occurred between to additions of an encrypted archive. This means that cryptographic protocol data is added to the agent only if the agent generated partial results. Given a valid Manifest M_z and sender signature $\mathcal{S}_z$ the application of a differential Manifest D_{n-1} to the given Manifest shall yield a Manifest M_{z-i} that can be validated with the previously recorded signature $\mathcal{S}_{z-i} = T_{n-1}$ where no encrypted archives were added for $i-1$ hops. Manifest M_{z-i} can be used to verify the integrity of data that was added by the $(h-i)$'th sender and has not been modified subsequently.

4.7 Final Processing of a Returning Mobile Agent

Upon the return of an agent that is owned by the receiving entity the accumulated partial results must be retrieved and their integrity must be verified based on the cryptographic protocol data contained in the mobile agent. This takes place after regular processing of an incoming agent as is described in sect. 4.4.

Verification takes place in two passes. The first pass verifies the chain of signatures and the integrity of the encrypted archives. The initial Manifest has already been verified successfully at this stage and thus the sections of these Manifests must bear the correct message digests of the agent's contents.

Require: Agent $\mathcal{A}$, E_0, K_0^{-1}, public key S_{z-1} of the most recent sender
1: $\text{load}(\mathcal{A}, M, L_0, \mathcal{S}_0, V)$
2: $n \leftarrow \min\{i \in \mathbb{N}_0 \mid \text{name}(Q_i) \notin \text{keys}(\mathcal{A})\} - 1$
3: $M' \leftarrow M \backslash \text{select}(M, L_0)$
4: **if** $n \geq 0 \wedge \text{name}(T_n) \notin \text{keys}(M)$ **then**
5: signers $\leftarrow \{(n, S_{z-1})\}$
6: $n \leftarrow n - 1$
7: **else**
8: signers $\leftarrow \{\emptyset, \emptyset\}$
9: **end if**
10: **for** $i = n, \ldots, 0$ **do**
11: $\text{load}(\mathcal{A}, T_i, D_i)$
12: $M' \leftarrow \text{sub}(M', D_i)$
13: **if** $\neg(E_0$ knows $S : S$ verifies $T_i, M', \mathcal{S}_0)$ **then**
14: **error** {Bad signature, results are incomplete.}
15: **end if**
16: **if** $\neg\, \text{check}(M, M', Q_i, R_i)$ **then**
17: **error** {Encrypted data was tampered with.}
18: **end if**
19: **if** $n > 0 \wedge \neg\, \text{check}(M, M', T_{i-1})$ **then**
20: **error** {Previous signature was tampered with.}

21: **end if**
22: signers $\leftarrow$ signers $\cup \{(i, S)\}$
23: **end for**

The second pass verifies the correctness of the partial result authentication code. This prevents some forms of truncation attacks and an attack where a host claims to have be the originator of a previously added partial result. The algorithm continues as follows:

24: $V' \leftarrow V_0$
25: **for** $i = 0, \ldots, n$ **do**
26: $\mathrm{load}(\mathcal{A}, Q_i)$
27: $N_i \leftarrow \{\mathrm{value}(Q_i, E_0)\}_{K_0^{-1}}$
28: secrets $\leftarrow$ secrets $\cup \{(i, N_i)\}$
29: acl $\leftarrow$ acl $\cup \{(\text{“results/”} \circ i, i)\}$
30: $S \leftarrow \mathrm{value}(\text{signers}, i)$
31: $V' \leftarrow h(V' \mid\mid N_i \mid\mid S)$
32: **end for**
33: **if** $V \neq V'$ **then**
34: **error** {Partial result authentication code mismatch.}
35: **end if**

We have to make sure that no unauthorized data is merged into the partial results of the agent by the decryption process.

36: **if** $\mathrm{children}(\mathcal{A}, \text{“results/”}) \neq \emptyset$ **then**
37: **error** {Agent is not clean.}
38: **end if**
39: $\mathrm{inflate}_R(\mathcal{A}, \text{acl}, \text{secrets})$

This completes the final processing of the mobile agent. Please note that the partial results must be deleted from the agent if it migrates after the final processing. Otherwise, the agent might leak the clear text.

5 Implementation Notes

The algorithms given in sect. 4 are not yet entirely implemented. However, we made an implementation of a (functionally equivalent) subset of these processes with the following exceptions and differences:

- Agents are encoded as JAR files; Manifests comply to the JAR standard.
- Partial results are handled differently and no PRACs are computed nor verified.
- Pre-generated encrypted archives may be either static or mutable files so that an agent can modify or store partial results in them while being hosted by an authorized recipient.
- Signatures of previous hosts are not saved.

– Hence there is no final processing of a returning agent (sect. 4.7) on top of the regular processing of an incoming agent.

The remaining set of features allows an agent to carry encrypted data that is transparently revealed to the agent while being hosted by an authorized recipient. Agents can modify the data which are re-encrypted transparently before the agent migrates. Interleaving attacks on the encrypted data are detected as described in sect. 4. However, an adversary may replace an encrypted archive with a previous version of that archive thus possibly rolling back modifications.

We measured the overhead of the cryptographic processing using a setup of four computers. The implementation was based on Java Version 1.3.0_01 *(HotSpot VM, native threads, sunjit)* and ran on Sun Ultra 5/10, UPA/PCI (UltraSPARC-IIi 333Mhz), 111 MHz, Solaris 8, which were connected to a Switched Fast Ethernet (100 MBit/s) that serves more than 200 workstations and PCs which are accessed by more than 320 staff members, researchers and students.. The software was loaded from an Auspex 700 fileserver. The cryptographic processing of agents was implemented by means of security filter plugins for the SeMoA mobile agent server [14]. We used DSA for signing and RSA with Triple DES for encryption.

In our measurements, we let a mobile agent migrate 600 times between these computers. In each test, we varied the size of the payload carried by the agent and set it to 0, 32, 64 und 96 KB respectively. The payload consisted of random data produced by means of the SHA1 pseudorandom number generator that is included in the JDK. Our intention was to produce a payload that yields a low compression rate before encryption takes place. The size of the bare agent without payload was about 23 KB.

Figure 2 summarizes the results of our measurements. The dashed line gives the mean time per migration where neither signatures nor encryption is used. The solid line gives the mean migration time where only signatures are computed on the agent. This means that on each migration two signatures were verified and one signature was generated. The dotted line gives the mean time per migration where the payload is decrypted and re-encrypted on each migration, on top of the signature computation. The reported agent size is the overall size of the agent as it was transported between hosts. Due brevity prevents us from interpreting or comparing the results which needs to be done in a separate work.

6 Related Work

Several protocols of the type on which we focus in this paper were published in the past [9–11, 15] and were found to be vulnerable [1]. The work we present in this paper is meant to "tie some loose ends" that led to the aforementioned attacks. Additional protocols e.g., published in [11, 16] remain to be investigated with regard to the applicability of the attacks described in [1].

The representation of a mobile agent as a collection of files matches the *Briefcase* abstraction in TACOMA [17], and shares the same advantages. Most notably, the representation is independent of a particular agent implementation language. Our contribution is the addition of a security layer that is tailored to the needs of mobile agents.

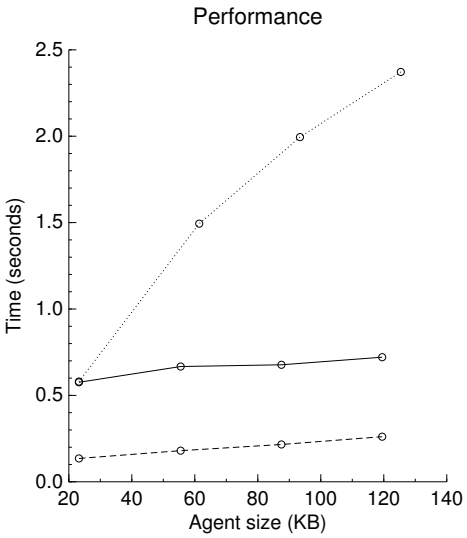


Fig. 2. This figure gives the mean time per migration of a mobile agent with a varying payload.

7 Conclusions

In this paper we presented algorithms and data structures meant to protect free-roaming mobile agents against certain types of malicious host attacks, namely, attacks on the integrity and confidentiality of data that is brought or acquired by a mobile agent on its route. Our approach has the advantage that it:

- is robust against interleaving attacks as described in [1];
- can be implemented transparently for agent programmers by means of cryptographic preprocessing and post processing of the agent meta-structure based on a simple-to-understand security policy;
- simultaneously protects the overall integrity of the agent during transport;
- adds cryptographic protocol data only if a partial result is generated; and
- although our approach is file-based, suitable abstractions for e.g., object-oriented agent programming languages are easily provided.

The level of detail at which we describe our approach eliminates ambiguities that may be introduced by a presentation that focusses on the pure protocol aspects thereby discounting the side effects of a protocol's translation into an implementation. It should be noted, though, that a mobile agent still needs to be programmed carefully so that tampering with its mutable state does not lead to leakage of sensitive data.

References

1. V. Roth, "On the robustness of some cryptographic protocols for mobile agent protection," in *Proc. Mobile Agents 2001*, vol. 2240 of *Lecture Notes in Computer Science*, Springer Verlag, December 2001.

2. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Discrete Mathematics and its Applications, New York: CRC Press, 1996. ISBN 0-8493-8523-7.
3. V. Roth and V. Conan, "Encrypting Java Archives and its application to mobile agent security," in *Agent Mediated Electronic Commerce: A European Perspective* (F. Dignum and C. Sierra, eds.), vol. 1991 of *Lecture Notes in Artifical Intelligence*, pp. 232–244, Berlin: Springer Verlag, 2001.
4. M. Bellare and B. Yee, "Forward integrity for secure audit logs," tech. rep., Computer Science and Engineering Department, University of California at San Diego, November 1997.
5. PKWARE Inc., 9025 N. Deerwood Dr., Brown Deer, WI 53223-2480, *.ZIP File Format Specification*, November 2001. Available at URL `http://www.pkware.com/support/appnote.html`.
6. T. Dell, D. Hopwood, D. Brown, B. Renaud, and D. Connelly, *Manifest Format*. Sun Microsystems Inc. and Netscape Corporation, March 1999. Available at URL `http://java.sun.com/products/jdk/1.2/docs/guide/jar/`.
7. T. Bray, E. Maler, J. Paoli, and C. M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0," w3c recommendation, W3C, October 2000. Available at URL `http://www.w3.org/TR/2000/REC-xml-20001006`.
8. V. Roth, "Mutual protection of co–operating agents," in *Secure Internet Programming: Security Issues for Mobile and Distributed Objects* (J. Vitek and C. Jensen, eds.), vol. 1603 of *Lecture Notes in Computer Science*, pp. 275–285, New York, NY, USA: Springer-Verlag Inc., 1999.
9. A. Corradi, R. Montanari, and C. Stefanelli, "Mobile agents protection in the Internet environment," in *The 23rd Annual International Computer Software and Applications Conference (COMPSAC '99)*, pp. 80–85, 1999.
10. G. Karjoth, N. Asokan, and C. Gülcü, "Protecting the computation results of free–roaming agents," in *Proceedings of the Second International Workshop on Mobile Agents (MA '98)* (K. Rothermel and F. Hohl, eds.), vol. 1477 of *Lecture Notes in Computer Science*, pp. 195–207, Berlin Heidelberg: Springer Verlag, September 1998.
11. G. Karjoth, "Secure mobile agent-based merchant brokering in distributed marketplaces," in *Proc. ASA/MA 2000* (D. Kotz and F. Mattern, eds.), vol. 1882 of *Lecture Notes in Computer Science*, pp. 44–56, Berlin Heidelberg: Springer Verlag, 2000.
12. International Telecommunication Union, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*, December 1997. ITU-T Recommendation X.680, equivalent to ISO/IEC International Standard 8824-1.
13. International Telecommunication Union, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*, December 1997. ITU-T Recommendation X.690, equivalent to ISO/IEC International Standard 8825-1.
14. V. Roth and M. Jalali, "Concepts and architecture of a security-centric mobile agent server," in *Proc. Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, (Dallas, Texas, U.S.A.), pp. 435–442, IEEE Computer Society, March 2001. ISBN 0-7695-1065-5.
15. N. M. Karnik and A. R. Tripathi, "Security in the Ajanta mobile agent system," Technical Report TR-5-99, University of Minnesota, Minneapolis, MN 55455, U. S. A., May 1999.
16. S. Loureiro, *Mobile Code Protection*. Ph.d. thesis, Ecole Nationale Supérieure des Télécommunications, January 2001.
17. D. Johansen, R. van Renesse, , and F. B. Schneider, "An introduction to the TACOMA distributed system version 1.0," Technical Report 95-23, Department of Computer Science, University of Tromsø, June 1995.