Automated Verification of Security Policies in Mobile Code[⋆]

Chiara Braghin[1], Natasha Sharygina[2,3], and Katerina Barone-Adesi[2]

[1] DTI, Università Statale di Milano, Crema, Italy
[2] Faculty of Informatics, Università della Svizzera Italiana, Lugano, Switzerland
[3] School of Computer Science, Carnegie Mellon University, Pittsburgh, USA

Abstract. This paper describes an approach for the automated verification of mobile programs. Mobile systems are characterized by the explicit notion of locations (e.g., sites where they run) and the ability to execute at different locations, yielding a number of security issues. We give formal semantics to mobile systems as Labeled Kripke Structures, which encapsulate the notion of the location net. The location net summarizes the hierarchical nesting of threads constituting a mobile program and enables specifying security policies. We formalize a language for specifying security policies and show how mobile programs can be exhaustively analyzed against any given security policy by using model checking techniques.

We developed and experimented with a prototype framework for analysis of mobile code, using the SATABS model checker. Our approach relies on SATABS's support for unbounded thread creation and enhances it with location net abstractions, which are essential for verifying large mobile programs. Our experimental results on various benchmarks are encouraging and demonstrate advantages of the model checking-based approach, which combines the validation of security properties with other checks, such as for buffer overflows.

1 Introduction

Despite the promising applications of mobile code technologies, such as web services and applet models for smart cards, they have not yet been widely deployed. A major problem is security: without appropriate security measures, a malicious applet could mount a variety of attacks against the local computer, such as destroying data (e.g., reformatting the disk), modifying sensitive data (e.g., registering a bank transfer via a home-banking software), divulging personal information over the network, or modifying other programs.

Moreover, programming over a wide area network such as the Internet introduces new issues to the field of multi-threaded programming and analysis. For example, during the execution of a mobile program, a given thread may stop executing at a site, and continue executing at another site. That is, threads may jump from site to site while retaining their conceptual identity. The following issues distinguish mobile systems from a more general case of multi-threaded programs:

[⋆] This work was done when the first author was staying at the Faculty of Informatics, Università della Svizzera Italiana, Lugano, Switzerland.

J. Davies and J. Gibbons (Eds.): IFM 2007, LNCS 4591, pp. 37–53, 2007.

- threads may run in different locations (e.g., administrative domains, hosts, physical locations, etc.);
- communication among threads and threads migration take into account their geographical distribution (e.g., migration can only occur between directly linked net locations).

To protect mobile systems against security leaks, *security policies* are defined, i.e., rules or conditions that state which actions are permitted and which are prohibited in the system. The rules may concern access control or information flow, and are usually verified on the fly during the system execution. The dynamic approach has several drawbacks: it slows down the system execution, there is no formal proof that the dynamic checks are done properly, and the checks are not exhaustive.

This paper describes an approach for modeling and verifying mobile programs. We give formal semantics to mobile systems as Labeled Kripke Structures (LKSs), which encapsulate the notion of location and unbounded thread creation typical to mobile systems. We define the semantics of mobile programs where thread locations are hierarchically structured, where threads are always confined to locations and where threads may move within the Internet. The LKS notation allows modeling both data and communication structures of the multi-threaded systems. Consequently, it outperforms the traditional process algebra approach that captures only the communication behavior.

We formalize a language for specifying general-purpose and application-dependent security policies, and we show how mobile programs can be statically and exhaustively analyzed against those security policies by using model checking techniques. A policy configuration file, specifying what permissions (i.e., which types of system resource access) to deny, is given as an input to the model checker together with the program to be verified. To support features of mobile systems, the policy specification language defines rules for expressing and manipulating the code location.

We implemented a prototype framework for modeling and verifying mobile programs written in C. In our approach, a mobile program is annotated with information related to the security policy in such a way that if and when the security policy is violated, the model checker returns a counter-example that led to such an error. In such a way, we are able to discover both implementation and malicious errors. Our framework uses the SATABS model checker [1], which implements a SAT-based counterexample-guided abstraction refinement framework (CEGAR for short) for ANSI-C programs.

To cope with the computational complexity of verifying mobile programs, we define *projection* abstractions. Given a path of a multi-threaded program, one can construct projections by restricting the path to actions or states satisfying certain conditions. We exploit the explicit notion of locations and define location-based projections, which allow efficient verification of location-specific security policies.

In summary, our approach to modeling and verifying mobile programs has several advantageous features:

- it explicitly models thread location, location distribution and thread moving operations, which are essential elements of mobile programs;
- it preserves both data and communication structures of mobile systems;
- it defines a specification language for specifying security policies of mobile code;

- it integrates model checking technologies to support exhaustive analysis of security policies;
- it defines location-specific abstractions which enable the efficient verification of large mobile code applications.

We experimented with a number of mobile code benchmarks by verifying various security policies. The results of verifying security policies, dealing with both access permissions of system actions and tracing the location net with respect to permissible location configurations, were encouraging.

2 Related Work

The use of mobile systems raises a number of security issues, including access control (is the use of the resource permitted?), user authentication (to identify the valid users), data integrity (to ensure data is delivered intact), data confidentiality (to protect sensitive data), and auditing (to track uses of mobile resources). All but the first category are closely coupled with research in cryptography and are outside of the scope of this paper. Our techniques assume that the appropriate integrity checking and signature validation are completed before the security access policies are checked.

Trust management systems (TMS) [2] address the access control problem by requiring that security policies are defined explicitly in a specification language, and relying on an algorithm to determine when a specific request can be allowed. An extensive survey of trust management systems and various authorization problems can be found in [3,4,5]. The major difference from our work is that these techniques rely on encryption techniques or proof-carrying code certification. For example, in the SPKI/SDSI framework, all principals are represented by their public keys, and access control is established by checking the validity of the corresponding public keys. In contrast, our security analysis reduces access control problems to static reachability analysis.

Certified code [6] is a general mechanism for enforcing security properties. In this paradigm, untrusted mobile code carries annotations that allow a host to verify its trustworthiness. Before running the agent, the host checks the annotations and proves that they imply the host's security policy. Despite the flexibility of this scheme, so far, compilers that generate certified code have focused on simple type safety properties rather than more general security policies. The main difficulty is that automated theorem provers are not powerful enough to infer properties of arbitrary programs and constructing proofs by hand is prohibitively expensive. Unable to prove security properties statically, real-world security systems such as the Java Virtual Machine (JVM) have fallen back on run-time checking. Dynamic security checks are scattered throughout the Java libraries and are intended to ensure that applets do not access protected resources inappropriately. However, this situation is unsatisfying for a number of reasons: 1) dynamic checks are not exhaustive; 2) tests rely on the implementation of monitors which are error-prone; and 3) system execution is delayed during the execution of the monitor.

Modeling of Mobile Systems. The most common approach to modeling mobile programs is the process algebra-based approach. Various location-aware calculi, with an explicit notion of *location*, have arisen in the literature to directly model phenomena such as the distribution of processes within different localities, their migrations, or their failures [7,8,9,10].

The π calculus [11] is often referred to as *mobile* because it features the ability to dynamically create and exchange channel names. While it is a de facto standard for modeling concurrent systems, the mobility it supports encompasses only part of all the abstractions meaningful in a distributed system. In fact, it does not directly and explicitly model phenomena such as the distribution of processes within different localities, their migrations, or their failures. Moreover, mobility is not expressed in a sufficiently explicit manner since it basically allows processes only to change their interconnection structures, even if dynamically. Indeed, name mobility is often referred to as a model of labile processes or as link mobility, characterized by a dynamic interaction structure, and distinguished from calculi of mobile processes which exhibit explicit movement.

Seal [12] is one of the many variants spawned by π calculus. The principal ingredient added, the seal, is the generalization of the notions of agents and locations. Hierarchical locations are added to the syntax, and locations influence the possible interaction among processes. As in the π calculus, interaction takes place over named channels. Communication is constrained to take place inside a location, or to spread over two locations that are in a parent-child relationship. Locations are also the unit of movement, abstracting both the notions of site and agent: a location, together with its contents, can be sent over a channel, mimicking mobility of active computations.

Djoin [10] extends the π calculus with location, migration, remote communication and failure. The calculus allows one to express mobile agents roaming on the net, however, differently from the Mobile Ambient calculus, the details of message routing are hidden.

The most famous one is the Mobile Ambient calculus [13,7]: this specification language provides a very simple framework that encompasses mobile agents, the domains where agents interact and the mobility of the agents themselves. An ambient is a generalization of both agent and place notions. Like an agent, an ambient can move across places (also represented by ambients) where it can interact with other agents. Like a place, an ambient supports local undirected communication, and can receive messages (also represented by ambients) from other places [14]. The formal semantics we give to mobile systems draws many ideas from the ambient calculus.

The disadvantages of process algebra-based approaches, however, is that they model only limited details of the systems (they are restricted only to communication structures and do not preserve any information about data). This restricts the set of properties that can be analyzed to a set of control-specific properties. Additionally, process-algebraic techniques usually deal with coarse over approximations during the analysis of mobile systems. Over-approximations are useful to reduce the analysis complexity and guarantee that, if no errors are found in the abstract system, then no errors are present in the actual system. However, if errors are found, the verification techniques developed for process algebra fail to guarantee that they are real. In contrast to the process algebraic approach, our techniques not only model both data and communication structures but also (in the context of the abstraction-based model checking) simulate the errors on the actual system and, if the errors are found to be spurious, the approximated programs are refined. To the best of our knowledge, there are no abstraction-refinement techniques that would support the process algebraic analysis techniques.

3 Formal Semantics of Mobile Programs

3.1 Mobile Programs

This section gives the syntax of mobile programs using a C-like programming language (which we believe is one of the most popular general-purpose languages). We extend the standard definition of multi-threaded programs with an explicit notion of *location* and moving actions[1]. The syntax of a mobile program is defined using a finite set of *variables* (either local to a thread or shared among threads), a finite set of *constants*, and a finite set of *names*, representing constructs for thread synchronization, similar to the Java `wait` and `notify` constructs. It is specified by the following grammar:

LT ::=		location-aware threads
	$\mid \ell[T]$	single thread
	$\mid LT_1 \parallel LT_2$	parallel composition
T ::=		threads
	$T_1 \mid T_2$	parallel comp.
	$\mid$ *Instr*	sequential exec.
Instr ::=		instructions
	Instr$_1$ `;` *Instr*$_2$	sequential exec.
	$\mid$ `x := e`	assignment
	$\mid$ `if` (*Expr* `!= 0`) *Instr*	condition
	$\mid$ `while` (*Expr* `!= 0`) *Instr*	loop
	$\mid$ `skip`	skip
	$\mid$ `m`	sync. call
	$\mid$ `fork`	thread creation
	$\mid M\_Instr$	moving action
Expr ::=		expressions
	`c`	constant
	$\mid$ *Expr*$_1$ $(+ \mid - \mid * \mid /)$ *Expr*$_2$	arith. operation
M_Instr ::=		moving actions
	$\mid$ `go_in(`ℓ`)` $\mid$ `go_out(`ℓ`)`	move in/out

In the grammar, `x` ranges over variables, `c` over constants, and `m` over the names of synchronization constructs. The meaning of the constructs for expressions and instructions is rather intuitive: an expression can be either a constant or an arithmetic operation (i.e., sum, difference, product and division). The instruction set mainly consists of the standard instructions for imperative languages: a sequential composition operator (`;`), the assignment instruction, the control flow instructions `if` and `while`, and the `skip` statement. The instructions specific to the threads package are the `fork` instruction, which spawns a new thread that is an exact copy of the thread executing the `fork` instruction, and the call to a synchronization method `m`.

We further assume a set of location names *Loc*, and we let $\ell, \ell_1, \ell_2, \ldots$ range over *Loc*. A thread is $\ell[T]$, with ℓ being the location name of thread T. More than one

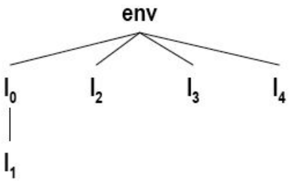
[1] For detailed discussion on programming languages for mobile code and their syntax the reader can refer to [15].

thread may be identified by the same location, that is $\ell[\![T_1 \mid T_2]\!]$ (examples will be shown later). A mobile program is defined by the parallel composition of multiple threads. A location can thus be seen as a bounded place, where mobile computation happens.

Conceptually, thread locations represent the geographical distribution of the Web. To capture this fact, we use a special structure, called a *location net*, which encapsulates the hierarchical nesting of the Web. We define the location net as a tree, whose nodes are labeled by unique location names, and the root is labeled by the special location name env, representing the external environment of the system under analysis. A tree t_ℓ is identified with the set of its paths.

Example 1. As a running example consider a *shopping agent* program, where several agents are sent out over the network to visit airline Web-sites to find the best airfare. Each agent is given various requirements, such as departure and destination time restrictions. After querying the airline database, it reports back the information to the user who made the request.

For simplicity, let's assume that the system is composed of threads $T_1 ... T_6$ which are distributed among various locations: *Loc* $= \{env, \ell_0, \ell_1, \ell_2, \ell_3, \ell_4\}$ and that a single thread is sent out. Here, ℓ_2, ℓ_3, ℓ_4 are the locations of various websites; ℓ_1 is the location of the agent, ℓ_0 is the program sending out the agent, and env the generalized environment location. Clearly, some of the locations are nested, and the location net corresponds to a tree, which can be defined by the set of its paths, i.e., $t_\ell = \{env.\ell_0.\ell_1, env.\ell_2, env.\ell_3, env.\ell_4\}$, or can be depicted as follows.



In the rest of the paper, when referring to nodes of the location net, we borrow standard vocabulary to define the relationship among tree nodes, such as father, child and sibling. For instance, in our example, ℓ_2 and ℓ_3 are siblings, whereas ℓ_0 is the father of ℓ_1 (and ℓ_1 is the child of ℓ_0). $\diamond$

The location net represents the topology of thread locations. In fact, it implicitly represents the distribution of threads. Location-aware threads can perform moving actions to change this distribution. These actions are the moving instructions, `go_in` and `go_out`. The explicit notion of location and the existence of moving actions affect the interaction among concurrent threads as follows (the formal definition will be given in Section 3.2):

- There are two types of composition: the parallel composition among threads identified by the same location (i.e., $\ell[\![T_1 \mid T_2]\!]$), and the parallel composition among threads identified by different locations (i.e., $\ell_1[\![T_1]\!] \parallel \ell_2[\![T_2]\!]$) - see the example below.
- The execution of moving actions changes the location net, i.e., mobility can be described by updates of the location net.

- The execution of moving actions is constrained by the structure of the location net, i.e., moving actions can be performed only if the thread location and the target location has the father-child or siblings relationship.

Example 2. For example, if threads T_1 and T_2 represent a mail server and a browser running at site l_0 and threads $T_3 ... T_6$ are each running at sites $l_1 ... l_4$, then the shopping agent program of Example 1 can be formalized as follows:

$$\ell_0[\![T_1 \mid T_2]\!] \;\parallel\; \ell_1[\![T_3]\!] \;\parallel\; \ell_2[\![T_4]\!] \;\parallel\; \ell_3[\![T_5]\!] \;\parallel\; \ell_4[\![T_6]\!]$$

In this program, threads T_1 and T_2 are running in parallel *locally* since $\ell_0[\![T_1 \mid T_2]\!]$. On the contrary, T_3 and T_4 are running *remotely* since $\ell_1[\![T_3]\!] \;\parallel\; \ell_2[\![T_4]\!]$. $\diamond$

3.2 The Computational Model

In this section we formalize the semantics of mobile programs. We first define the semantics of a single thread, and then extend it to the case of a multi-threaded system. As done in the examples of the previous section, when discussing about multi-threaded systems consisting of n threads, we will use i, with $1 \leq i \leq n$, as a unique identifier of each thread T (i.e., we will write T_i).

Definition 1 (Location-aware Thread). *A thread is defined as a Labeled Kripke Structure $T = (S, Init, AP, \mathcal{L}, \Sigma, \mathcal{R})$ such that:*
- *S is a (possibly infinite) set of states;*
- *$Init \in S$ is the initial state;*
- *AP is the set of atomic propositions;*
- *$\mathcal{L} : S \rightarrow 2^{AP}$ is a state-labeling function;*
- *Σ is a finite set (alphabet) of* actions*;*
- *$\mathcal{R} \subseteq S \times \Sigma \times (S \cup \{\, S \times S \,\})$ is a total* labeled *transition relation.*

A state $s \in S$ of a thread is defined as a tuple $(V_l, V_g, pc, \varphi, \eta)$, where V_l is the evaluation of the set of local variables, V_g is the evaluation of the set of global variables, pc is the program counter, $\varphi : Loc \hookrightarrow Loc$ is a partial function denoting the *location net* (where *Loc* is the set of location names as defined in Section 3.1), and $\eta : \mathbb{N} \hookrightarrow Loc$ is a partial function denoting the *thread location*. More specifically, φ describes the location net at a given state by recording the father-child relationship among all nodes of the net ($\perp$ in the case of env), whereas $\eta(i)$ returns the location name of T_i (i.e., the thread identified by i).

Example 3. Consider again the shopping agent program and its location net as defined in Example 2. In this case, the location net function is $\varphi(\ell_0) = env, \varphi(\ell_1) = \ell_0, \varphi(\ell_2) = env, \varphi(\ell_3) = env, \varphi(\ell_4) = env$. In addition, the thread location function for threads $T_1 \cdots T_6$ is defined as $\eta(1) = \ell_0, \eta(2) = \ell_0, \eta(3) = \ell_1, \eta(4) = \ell_2, \eta(5) = \ell_3, \eta(6) = \ell_4$. $\diamond$

The transition relation $\mathcal{R}$ is labeled by the *actions* of which there are four types: *moving*, *synchronization*, *thread creation*, and τ actions, which are contained in the mutually disjoint sets Σ^M, Σ^S, Σ^T, Σ^τ, respectively. We use Σ to identify the set of all actions. τ represents a generic action such as an assignment, a function call, etc. We write

$s \xrightarrow{a} s'$ to mean $(s, a, s') \in \mathcal{R}$, with $a \in \Sigma$. Moreover, we write $s \xrightarrow{a}_i s'$ to specify which thread performed the action. Note that, since we allow thread creation, if thread T_i performs a `fork` action, s' can be defined as a pair of states s.t. $s \xrightarrow{\texttt{fork}}_i (s', \overline{s})$, where s' is the next state of s, and $\overline{s} = Init_i$ is an initial state of the newly created thread (which corresponds to the initial state of T_i).

Table 1 gives the inference rules for the labeled transition relation in the case of moving actions $(\texttt{go\_in}(\ell), \texttt{go\_out}(\ell))$, thread creation action, `fork`, and the synchronization action `m`. For the rules corresponding to the generic operations the reader is referred to [16]. The premises of the rules presented in Table 1 represent guarded conditions for the execution of the actions. All rules check the value of *Instr*$(s.pc)$, which determines the instruction to be executed by the running thread. Then, depending on the type of the action, they check further guarding conditions. In the consequences of the inference rules, we describe (within square brackets) the updates of the thread state caused by the execution of an action. We use the standard notation $\varphi \cup \{\ell_1 \mapsto \ell_2\}$ (with $\ell_1, \ell_2 \in \mathit{Loc}$) to indicate the update to function φ, i.e., the updates to the location net.

In the case of a "`fork`" action, thread T_i spawns a new thread that is an exact copy of itself. As a consequence, the program counter of T_i is updated, and a new thread is created with an initial state $\overline{s}$. The initial state is a copy of the initial state of T_i.

In the case of a "$\texttt{go\_in}(\ell)$" action, if ℓ is a sibling location to thread T_i location (i.e., $s.\varphi(s.\eta(i)) = s.\varphi(\ell)$), then the thread makes a transition and changes the state accordingly: the program counter *pc* is incremented, and the location net is updated (ℓ is now the father location of T_i location). If ℓ is not a sibling location, then the action is not performed because the guard does not hold.

In the case of a"$\texttt{go\_out}(\ell)$" action, if ℓ is the father location to thread T_i location (i.e., $s.\varphi(s.\eta(i)) = \ell$), then the thread makes a transition and changes the state accordingly: the program counter *pc* is incremented, and the location net is updated (ℓ is now a sibling location of T_i location). If ℓ is not the father location, then the action is not performed because the guard does not hold.

Note that the subtle features of mobile programs (namely, location, location net and unbounded thread creation) are modeled explicitly.

Let $T_1, \cdots, T_n$ be a set of threads initially present in the mobile program $\mathcal{P}$, then $\mathcal{P} = T_1 \parallel \cdots \parallel T_n$. The parallel composition operation is defined as follows.

Definition 2 (Mobile Program). *Let thread* $T_1 = (S_1, Init_1, AP_1, \mathcal{L}_1, \Sigma_1, \mathcal{R}_1)$ *and thread* $T_2 = (S_2, Init_2, AP_2, \mathcal{L}_2, \Sigma_2, \mathcal{R}_2)$ *be two Labeled Kripke structures. Then their composition is defined as follows:* $T_1 \parallel T_2 = (S_1 \times S_2, Init_1 \times Init_2, AP, \mathcal{L}, \Sigma_1 \cup \Sigma_2, \mathcal{R})$ *with the labeled transition relation defined in Table 2.*

In Table 2, a single state belonging to thread T_i is denoted by s^i, i.e., with i as superscript to indicate the thread number. When needed, we also use a subscript (and variable j) to indicate the position of an element in the path. For example, s^i_1 is the initial state of thread T_i. Given a state $s^i \in S_i$ of thread T_i, $s^i.V_l$, $s^i.V_g$, $s^i.pc$, $s^i.\varphi$ and $s^i.\eta$ are the values of local variables V_l, of global variables V_g, of program counter *pc*, of φ and of η, respectively. Moreover, *Instr*$(s^i.pc)$ denotes the instruction pointed by *pc* in thread T_i at state s^i. Note that $\forall i, j, i \neq j, \Sigma_i \cap \Sigma_j = \Sigma^S$, that is threads share only synchronization actions. In other words, threads proceed independently on local actions

Table 1. Inference rules for the labeled transition relation $\mathcal{R}$ for thread T_i

(FORK-ACTION)

$$\frac{\mathit{Instr}(s.pc) = \texttt{fork}}{s \xrightarrow{\texttt{fork}}_i (s', \overline{s}) \;\; [s'.pc = s.pc + 1; \overline{s} = Init_i]}$$

(**in**-ACTION)

$$\frac{\mathit{Instr}(s.pc) = \texttt{go\_in}\,(\ell) \wedge (\exists \ell_1 . \ell_1 := s.\eta(i) \wedge s.\varphi(\ell_1) = s.\varphi(\ell))}{s \xrightarrow{\texttt{go\_in}(\ell)}_i s' \;\; [s'.pc = s.pc + 1; s'.\varphi = s.\varphi \cup \{\ell_1 \mapsto \ell\}]}$$

(**out**-ACTION)

$$\frac{\mathit{Instr}(s.pc) = \texttt{go\_out}\,(\ell) \wedge (\exists \ell_1 . \ell_1 := s.\eta(i) \wedge s.\varphi(\ell_1) = \ell)}{s \xrightarrow{\texttt{go\_out}(\ell)}_i s' \;\; [s'.pc = s.pc + 1; s'.\varphi = s.\varphi \cup \{\ell_1 \mapsto s.\varphi(\ell)\}]}$$

(SYNC-ACTION)

$$\frac{\mathit{Instr}(s.pc) = \texttt{m}}{s \xrightarrow{\texttt{m}}_i s' \;\; [s'.pc = s.pc + 1]}$$

and synchronize on shared actions ($m \in \Sigma^S$), or on shared data (by definition of S_i, $S_1 \cap S_2 \neq \emptyset$). This notion of composition is derived from CSP [17].

The definition of a path of a mobile program reflects the possibility of unbounded thread creation during the execution of the `fork` instruction.

Definition 3 (Path). *A path* $\pi = \langle (s_1^1, s_1^2, \ldots, s_1^{n_1}), a_1, (s_2^1, s_2^2, \ldots, s_2^{n_2}), a_2, \ldots \rangle$ *of a mobile program is an alternating (possible infinite) sequence of tuples of states and events such that:*

(i) $n_j \in \mathbb{N}$ *and,* $\forall i, j \geq 1, s_1^i = Init_i$, $s_j^i \in S_i$, *and* $a_j \in \cup_i \Sigma_i$*;*

(ii) *either* $s_j^i \xrightarrow{a_j} s_{j+1}^i$ *or* $s_j^i = s_{j+1}^i$ *for* $1 \leq i \leq n_{j+1}$*;*

(iii) *if* $a_j = \texttt{fork}$*:*

- *then* $n_{j+1} = n_j + 1$ *and* $s_{j+1}^{n_{j+1}} = Init_k$ *with* $s_j^k \xrightarrow{a_j} s_{j+1}^k$
- *else* $n_{j+1} = n_j$.

A path includes *tuples* of states, rather than a single state. The reason for that is that when a `fork` operation is executed, the state of the newly created thread must be recorded. Our notation indicates each state s_i^j by two indices, i and j, one to indicate the thread number, the other one to indicate the position in the path, respectively. The size of the tuple of states (i.e., the number of the currently existing threads) increases only if a `fork` is executed, otherwise it remains unchanged (case (iii)). In case of a `fork`, index k identifies the thread that performed the action. Thus, the state of the newly created thread is a copy of the initial state of thread T_k. Moreover, depending on the type of action (i.e., shared or local) one or more threads will change state, whereas the others do not change (case (ii)).

Table 2. The labeled transition relation for the parallel composition of two threads

(SYNC-ACTION)

$$\frac{a \in \Sigma_1^S \wedge s^1 \xrightarrow{a}_1 s'^{1} \wedge a \in \Sigma_2^S \wedge s^2 \xrightarrow{a}_2 s'^{2} \wedge s^1.\eta(1) = s^2.\eta(2)}{(s^1, s^2) \xrightarrow{a} (s'^{1}, s'^{2})}$$

(L-PAR)

$$\frac{a \in \Sigma_1^M \wedge s \xrightarrow{a}_1 s'^{1}}{(s^1, s^2) \xrightarrow{a}_1 (s'^{1}, s^2)}$$

(R-PAR)

$$\frac{a \in \Sigma_2^M \wedge s^2 \xrightarrow{a}_2 s'^{2}}{(s^1, s^2) \xrightarrow{a}_2 (s^1, s'^{2})}$$

4 Specifying Security Policies of Mobile Programs

In order to support features of mobile systems, we devised a policy specification language that defines rules for expressing also the code location. This security language primarily works at the level of method calls and variable accesses. Methods may be disallowed to an agent, either in general, or when invoked with specific arguments. (Global) variables may be marked as having a high security level, and they cannot be assigned to variables of a lower level; it is also possible to specify methods that may not be accessed within or passed to (no Read Up, no Write Down). In this way, it is possible to express both information flow and access control policies with the same language .

The BNF specification of the language follows, where terminals appear in Courier, non terminals are enclosed in angle brackets, optional items are enclosed in square brackets, items repeated one or more times are enclosed in curly brackets, and alternative choices in a production are separated by the | symbol. A policy might contain: (i) the definition of *security levels*; (ii) a list of *operation* definitions; and (iii) a list of *deny* statements, each one including one or more *permission* entries. In the definition of security levels, we enumerate the high level variables to specify a multi-level security policy. The definition of an *operation* collects together functions with the same meaning or side-effect (e.g., scanf and fread). A deny statement specifies which types of actions are not allowed to entities. By default, in the absence of deny statements, all actions are allowed to every possible user.

The *entities* to deny permissions to consist of processes (e.g., agents), identified by their current location. The keyword `public` means that the permission is denied to all entities. As we are dealing with mobile systems, an entity can also be identified by the host address (via `codeBase`), or by the location (via `codeOrigin`) it came from. The keyword `remote` identifies non-local locations.

A *permission* entry must begin with the keyword `permission`. It specifies *actions* to deny. An action can be a function (either user-defined or from the standard library), or an operation (a collection of functions). If it is a function, it is possible to also specify (i) formal parameters (variable names), (ii) actual parameters (the value of the arguments passed), (iii) an empty string, denying access to the function regardless of the arguments to it, or (iv) the keyword `high` (no high variables can be passed as arguments to this function). Notably, an actual parameter may be a location (a trailing $\star$ prevents not only the location, but all sub-locations too).

$$
\begin{aligned}
\langle\text{policy}\rangle &\longrightarrow \{\langle\text{sec\_levels}\rangle \mid \langle\text{operation\_def}\rangle \mid \langle\text{deny statement}\rangle\}\\
\langle\text{deny statement}\rangle &\longrightarrow \texttt{deny\textvisiblespace to}\,\langle\text{deny\_target}\rangle\ [\langle\text{code base}\rangle]\ \ [\langle\text{code origin}\rangle]\\
&\qquad \texttt{\{}\ \langle\text{permission entry}\rangle\ \{\texttt{,}\ \langle\text{permission entry}\rangle\}\ \texttt{\}}\\
\langle\text{deny\_target}\rangle &\longrightarrow \texttt{public} \mid \langle\text{entity list}\rangle\\
\langle\text{entity list}\rangle &\longrightarrow \langle\text{entity\_id}\rangle\ \{\texttt{,}\ \langle\text{entity\_id}\rangle\}\\
\langle\text{entity\_id}\rangle &\longrightarrow \langle\text{location\_id}\rangle\\
\langle\text{location\_id}\rangle &\longrightarrow \langle\text{identifier}\rangle\\
\langle\text{identifier}\rangle &\longrightarrow (\langle\text{letter}\rangle \mid \langle\text{symbol}\rangle)\ \ \{\langle\text{letter}\rangle \mid \langle\text{digit}\rangle \mid \langle\text{symbol}\rangle\}\\
\langle\text{symbol}\rangle &\longrightarrow \texttt{\_} \mid \texttt{.}\\
\langle\text{code base}\rangle &\longrightarrow \texttt{codeBase}\,\langle\text{IPv4 addr}\rangle\\
\langle\text{code origin}\rangle &\longrightarrow \texttt{codeOrigin}\ (\langle\text{location}\rangle \mid \texttt{remote})\\
\langle\text{location}\rangle &\longrightarrow \langle\text{location\_id}\rangle\ \{:\ \langle\text{location\_id}\rangle\}\\
\langle\text{permission entry}\rangle &\longrightarrow \texttt{permission}\,\langle\text{action}\rangle\\
\langle\text{action}\rangle &\longrightarrow \langle\text{function}\rangle \mid \langle\text{operation}\rangle\\
\langle\text{function}\rangle &\longrightarrow \texttt{function}\,\langle\text{function\_id}\rangle\ \langle\text{parameters}\rangle\\
\langle\text{function\_id}\rangle &\longrightarrow \langle\text{identifier}\rangle\\
\langle\text{parameters}\rangle &\longrightarrow \langle\text{actual par}\rangle \mid \langle\text{formal par}\rangle \mid \texttt{high} \mid \varepsilon\\
\langle\text{actual par}\rangle &\longrightarrow \texttt{"}\ \langle\text{string}\rangle\ \texttt{"}\\
\langle\text{formal par}\rangle &\longrightarrow \texttt{args}\ \langle\text{vars}\rangle \mid \texttt{"}\ \langle\text{location\_id}\rangle\ \texttt{"}\ [\star]\\
\langle\text{vars}\rangle &\longrightarrow \langle\text{identifier}\rangle\ \{\texttt{,}\ \langle\text{identifier}\rangle\}\\
\langle\text{operation\_def}\rangle &\longrightarrow \langle\text{operation}\ \rangle\ \texttt{\{}\ \langle\text{function\_id}\rangle\ \{\texttt{,}\ \langle\text{function\_id}\rangle\}\ \texttt{\}}\\
\langle\text{operation}\rangle &\longrightarrow \texttt{operation}\,\langle\text{operation\_id}\rangle\\
\langle\text{operation\_id}\rangle &\longrightarrow \langle\text{identifier}\rangle\\
\langle\text{receiver}\rangle &\longrightarrow \langle\text{location\_id}\rangle\\
\langle\text{sec\_levels}\rangle &\longrightarrow \texttt{High=\{}\ \langle\text{vars}\rangle\ \texttt{\}}
\end{aligned}
$$

Consider the Java sandbox: it is responsible for protecting a number of resources by preventing applets from accessing the local hard disk and the network. In our language, a sketch of this security policy could be expressed as:

```
operation read_file_system { fread, read, scanf, gets}
deny to public codeOrigin remote
{ permission function connect_to_location,
  permission operation read_file_system }
```

A multi-level security policy could be expressed as:

```
High={confidential_var, x}
deny to public codeOrigin remote
{ permission function fopen high}
```

4.1 Security and Projection

To cope with the computational complexity of verifying mobile programs, we define *projection* abstractions. Given a path of a multi-threaded program $T_1 \parallel \cdots \parallel T_n$, one can construct projections by restricting the path to the actions in the alphabet of threads, or to states satisfying some conditions. We exploit the explicit notion of locations and define the location-based projections, which allow efficient verification of location-specific security policies (security policies in which `codeOrigin` or `codeBase` is present). With a location-specific policy, only processes which run on the indicated location need to be verified.

In the following, we assume only paths of finite length, as they are computed by the symbolic fix-point algorithm to handle verification of systems with an unbounded number of threads. In addition, we write $\langle\rangle$ for the empty path, and we use the dot notation to denote the concatenation of sequences. The concatenation of sequences will be used in the inductive definitions of projections to concatenate subsequences of paths. Notice that . is the concatenation operator for sequences of characters, thus it is not affected by the presence of mismatched parentheses.

Definition 4 (Location Projection, $\pi \downarrow \ell$). *Let $\mathcal{P}$ be $T_1 \parallel \cdots \parallel T_n$ and $\ell \in Loc$ be a location. The projection function $Proj_\ell : L(\mathcal{P})^* \rightarrow L(\mathcal{P})^*$ is defined inductively as follows (we write $\pi \downarrow \ell$ to mean $Proj_\ell(\pi)$):*

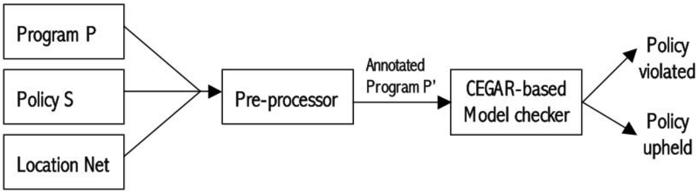
1. $\langle\rangle \downarrow \ell = \langle\rangle$
2. *If* $s^i.\eta(i) = \ell$ *then* $(\langle s^i\rangle.\pi) \downarrow \ell = \langle s^i\rangle.(\pi \downarrow \ell)$
3. *If* $s^i.\eta(i) \neq \ell$ *then* $(\langle s^i\rangle.\pi) \downarrow \ell = \pi \downarrow \ell$
4. *If* $a \in \Sigma_i$*, with* i *s.t.* $s^i.\eta(i) = \ell$*, then* $(\langle a\rangle.\pi) \downarrow \ell = \langle a\rangle.(\pi \downarrow \ell)$
5. *If* $a \notin \Sigma_i$*, with* i *s.t.* $s^i.\eta(i) \neq \ell$*, then* $(\langle a, (s^1, s^2, \ldots, s^n)\rangle.\pi) \downarrow \ell = \pi \downarrow \ell$

This projection traces the execution of threads for a particular location. The following information is collected: (i) states of threads whose location is ℓ (i.e., threads T_i such that $s^i.\eta(i) = \ell$), and (ii) actions that are performed by the threads whose location is ℓ (i.e., actions a such that $a \in \Sigma_i$, with $\ell[\![T_i]\!]$). Here, the concatenation is done on each state element of the path, since each single state is examined to satisfy condition (i) (rules 2-3). On the contrary, once an action does not satisfy condition (ii), the next tuple is erased (rule 4).

With respect to what happens at a particular location during execution of a mobile program, there is no loss of precision in this projection-based abstraction. The projection removes only states and actions which are irrelevant to the particular location. Moreover, since security policies are defined in terms of a single location, this abstraction does not introduce spurious counterexamples during the verification of security policies using the `codeOrigin` entry.

5 A Model Checking Framework for Verification of Security Policies

A prototype framework for security analysis of mobile programs is shown in the picture below. A mobile program, P, and a security policy, S, are provided as an input to the model checking engine.



These inputs are processed, creating a new program, P', annotated with the security invariants. It has the following property: an assertion `assert(0)` (a security invariant) is not reachable in P' if and only if P enforces the security policy S. Thus, it is sufficient to give P' as an input to a model checker to statically determine whether or not an `assert(0)` is reachable in P.

The procedure for annotating the program with security invariants is a multi-step process. First, the intersection of methods in the security policy and methods used within the agent to verify is found. Then, a wrapper method is created for each of these methods. This wrapper contains an `assert(0)`, either unconditionally, or within a guard, based on the policy (this may check where the agent came from, and/or the arguments being passed to the method). The handling of high variable access is more complex (due to scoping and syntax), but analogous. This annotating procedure, as in SLIC [18], is the implementation of Schneider's security automata [19]. In fact, the annotated program P' consists of program P with inlined the reference monitor that enforces the security policy S.

Our framework uses a model checking toolset, SATABS [1]. Applying model checking to the analysis of mobile and multi-threaded systems is complicated by several factors, ranging from the perennial scalability problems to thread creation that is potentially unbounded and that thus leads to infinite state space. *Predicate abstraction* is one of the most popular and widely applied methods for systematic state-space reduction of programs. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The resulting Boolean program is an over-approximation of the original program. One starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found. The actual steps of the loop follow the *abstract-verify-refine* paradigm [20]. SATABS implements the abstraction refinement loop by computing and refining abstract programs. The procedure for the location-specific projections can be seen as the extension of SATABS's abstraction procedures. Among various techniques employed by SATABS, there is a model checker for Boolean programs (computed by the SATABS abstraction engine), BOPPO[16] that handles unbounded thread creation. The execution of a `fork` action corresponds to the migration of the code to the new sites and potentially leads to the creation of an unbounded number of new threads. SATABS implements a symbolic algorithm for over-approximating reachability in Boolean programs to support arbitrary thread creation which is guaranteed to terminate [21]. The devised algorithm is used as the underlying reachability engine in the CEGAR framework and is efficient. The SATABS ability to handle programs with arbitrary thread creation was the key reason for using it as a model checking engine of our security framework.

An initial configuration file can also be provided, to test whether the policy is upheld in specific network configurations (where the agent came from, both on the underlying network and the location network, and where it's currently running). Several functions exist within the mobile code framework to check these values; there is a dynamic version to be used at run-time, and a static version which is generated from the specified initial configuration. To check whether the policy holds under all possible conditions, it suffices to not provide these function definitions to SATABS, which then treats the results as non-deterministic; this can be accomplished by telling SATABS to use the run-time version of the definitions, not providing an initial configuration, or by not providing the generated location file as an input to SATABS.

SATABS supports C programs, thus our benchmarks have a C-base mobile language. Since serialization is not possible in C, we only allow code mobility (i.e., applet sending); running code cannot migrate. It is straightforward to extend our approach to benchmarks using other programming languages (e.g., Java, Telescript, etc.) by implementing a different front-end targeted to the language of choice.

5.1 Experimental Results

To validate the theoretical concepts presented in this paper, an experimental mobile code framework was developed, for which a number of examples of mobile code agents were generated. The mobile code agents were a shopping agent [22] and an updating agent [23].

The shopping example deals with a shopping query client, which sends several agents out to query simulated airline services in order to find available airfares. The agent is run on a simulated airline server, which is a distinct location on the location net from the original query client, and may be on a remote host. When the agent receives a reply, or fails to, it then tries to report back to the shopping query client.

The updating example specifies a central update server and several clients. The clients contact the server, and updates are sent, as an executable agent, whenever an update is available. This represents a way to keep the client software up to date, without forcing the client to poll the update server.

We verified a number of security policies ranging from file access control to policies that conditionally allowed the use of mobile code APIs based on the `codeOrigin`. The examples have been tested against different security policies, some general and some application dependent, as well as different initial location configurations. Both contain a ”malicious” action (opening a connection to the location named ”bad” and opening /etc/passwd, respectively), and one of the security policies for each checks this. The results of the experiments, with a location projection (where ℓ=the agent's location) on the whole system, are reported in Table 5.1.

The above policies are of a few forms, best shown by example. The updating agent opens /etc/passwd: Policy 2 (ua) disallows this action if and only if the agent came from a remote location, whereas every other argument to fopen is allowed.

```
deny to public codeOrigin remote
{ permission function fopen "/etc/passwd"}
```

Table 3. Agent benchmarks with deterministic configurations: pv = policy violated, ua = updating agent, sa = shopping agent

policy	time (s)	# iterations	# predicates	pv?	SATABS: pv?
none (ua)	0	1	0	no	no
1 (ua)	10.888	2	11	yes	yes
2 (ua)	34.812	14	18	yes	yes
3 (ua)	0.194	1	3	yes	yes
none (sa)	0.001	1	0	no	no
no_effect (sa)	0	1	0	no	no
1 (sa)	151.644	7	17	yes	yes
2 local (sa)	100.234	5	15	no	no
2 remote (sa)	524.866	12	36	yes	yes
3 codeBase (sa)	340.011	12	22	yes	yes
3 (sa)	108.564	6	16	yes	yes

Policy 3 codeBase in the shopping agent example is a variant on the policy above: it specifies codeBase (an IPv4 origin address) instead of codeOrigin, and is tailored to the ”malicious action” found in the shopping agent.

```
deny to public codeBase 127.0.0.1
{ permission function connect_to_location bad}
```

Other policies are: ”none” (verifying the agent without any security policy), the Java-like policy described in Section 4 (Policy 1 (ua)), and the security-level example policy also described in Section 4 (Policy 3 (ua)).

We were able to validate our technique on systems of different complexities, by changing the number of agents instantiated. Our tools correctly detected every security policy violation with no false positives. We observed that without performing projections the verification was problematic, whereas when using location projection the technique scaled gracefully and the complexity of the verification was highly reduced. Table 1 reports the total verification time (in sec) for the shopping agent and the updating examples; a number of predicates and a number of the CEGAR loop iterations indicate the complexity of the abstracted models.

6 Conclusion

In this paper, we introduced a framework for the modeling and verification of mobile programs. The system semantics were presented in terms of Labeled Kripke Structures, which encapsulated the essential features of mobile programs: namely, location and unbounded thread creation. The explicit modeling of these features enabled the specification of mobile systems security policies, which are otherwise difficult to define. The verification was based on model checking, exploiting abstraction-refinement techniques that not only allowed handling unbounded state space, but also deal effectively with large systems.

Acknowledgments. The authors thank Daniel Kroening for useful discussions and support during the implementation of the prototype framework for verifying security policies.

References

1. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
2. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.D.: The role of trust management in distributed systems security, pp. 185–210 (1999)
3. Weeks, S.: Understanding trust management systems. In: IEEE Symposium on Security and Privacy, pp. 94–105 (2001)
4. Schwoon, S., Jha, S., Reps, T.W., Stubblebine, S.G.: On generalized authorization problems. In: CSFW, pp. 202–217 (2003)
5. Ganapathy, V., Jaeger, T., Jha, S.: Automatic placement of authorization hooks in the linux security modules framework. In: ACM Conf. on Comp.and Comm. Security., pp. 330–339 (2005)
6. Necula, G.C., Lee, P.: Research on proof-carrying code for untrusted-code security. In: IEEE Symposium on Security and Privacy, p. 204 (1997)
7. Cardelli, L., Gordon, A.D.: Mobile Ambients. In: Nivat, M. (ed.) ETAPS 1998 and FOSSACS 1998. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)
8. Hennessy, M., Riely, J.: Resource Access Control in Systems of Mobile Agents. In: HLCL '98. Journal of TCS, pp. 3–17. Elsevier, Amsterdam (1998)
9. De Nicola, R., Ferrari, G., Pugliese, R.: Klaim: a Kernel Language for Agents Interaction and Mobility. IEEE Transactions on Software Engineering 24(5), 315–330 (1998)
10. Fournet, C., Gonthier, G., Lévy, J.J., Maranget, L., Rémy, D.: A Calculus of Mobile Agents. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 406–421. Springer, Heidelberg (1996)
11. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, I and II. Information and Computation 100(1), 1–40, 41–77 (1992)
12. Vitek, J., Castagna, G.: Seal: A Framework for Secure Mobile Computations. In: Bal, H.E., Cardelli, L., Belkhouche, B. (eds.) Internet Programming Languages. LNCS, vol. 1686, pp. 47–77. Springer, Heidelberg (1999)
13. Cardelli, L.: Wide Area Computation. In: Wiedermann, J., van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 10–24. Springer, Heidelberg (1999) (Invited Paper)
14. Cardelli, L., Gordon, A.D.: Mobile Ambients. Theoretical Computer Science 240(1), 177–213 (2000)
15. Braghin, C., Sharygina, N.: Modeling and Verification of Mobile Systems. In: Proc. of TV 06 (2006)
16. Cook, B., Kroening, D., Sharygina, N.: Symbolic model checking for asynchronous boolean programs. In: Valmari, A. (ed.) Model Checking Software. LNCS, vol. 3925, pp. 75–90. Springer, Heidelberg (2006)
17. Roscoe, A.: The theory and practice of concurrency. Prentice-Hall, Englewood Cliffs (1997)
18. Ball, T., Rajamani, S.K.: SLIC: a Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research (2002)
19. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security, vol. 3(1) (2000)

20. Kurshan, R.: Computer-Aided Verification of Coordinating Processes. Princeton University Press, Princeton (1995)
21. Cook, B., Kroening, D., Sharygina, N.: Over-Approximating Boolean Programs with Unbounded Thread Creation. In: FMCAD 06: Formal Methods in System Design, Springer, Heidelberg (2006)
22. White, J.: Telescript technology: The foundation of the electronic marketplace. Technical report, General Magic Inc (1994)
23. Bettini, L., De Nicola, R., Loreti, M.: Software update via mobile agent based programming. In: SAC, pp. 32–36. ACM, New York (2002)