CardS4: Modal Theorem Proving on Java Smartcards

Rajeev Prabhakar Goré[1] * and Phuong Thê Nguyên[2] **

[1] Automated Reasoning Group, Computer Sciences Laboratory,
Res. Sch. of Inf. Sci. and Eng.
Institute of Advanced Studies, Australian National University
rpg@arp.anu.edu.au
arp.anu.edu.au/~rpg
[2] Formal Methods Group, Dept. of Computer Science, Fac. of Eng. and Inf. Tech.
The Faculties, Australian National University
ntp@cse.unsw.edu.au
www.cse.unsw.edu.au/~ntp

Abstract. We describe a successful implementation of a theorem prover for modal logic **S4** that runs on a Java smart card with only 512 KBytes of RAM and 32KBytes of EEPROM. Since proof search in **S4** can lead to infinite branches, this is "proof of principle" that non-trivial modal deduction is feasible even on current Java cards. We hope to use this prover as the basis of an on-board security manager for restricting the flow of "secrets" between multiple applets residing on the same card, although much work needs to be done to design the appropriate modal logics of "permission" and "obligations". Such security concerns are the major impediments to the commercial deployment of multi-application smart cards.

Keywords: security of mobile code, modal deduction

1 Introduction

Smart cards are credit-card sized pieces of plastic with an embedded silicon chip. Smart cards are either memory cards, which cannot be programmed, or microprocessor cards, which contain a small amount of RAM and disc (EEPROM) on the card itself. A card reader/writer is required to provide power to the card, to provide a clock signal, and to act as an interface between the card and the terminal (a PC, an ATM machine, a public telephone, or even a mobile telephone).

Java cards are smart cards that contain a (downsized) Java platform, installed by the manufacturer, thus allowing users to download Java applets and run them on the card. Java cards can therefore provide multiple applications such as

* Supported by a Queen Elizabeth II Fellowship from the Australian Research Council.
** Supported by an RSISE Summer Research Scholarship.

I. Attali and T. Jensen (Eds.): E-smart 2001, LNCS 2140, pp. 111–123, 2001.

electronic purse, credit card, passport, loyalty programmes, all residing on the same card.

The Java cards used in this project were the GemXpresso RAD Protyping card, containing a 32-bit microprocessor with 512 bytes of RAM, 32 KBytes of Flash EEPROM and 8 KBytes of ROM.

Current Java cards are preprogrammed to contain applets by the manufacturer for the card vendor, typically a bank (for credit and debit cards), or an airline (for frequent flyer cards). But if Java cards are to succeed then a card carrier must be able to down-load new applets onto an existing card "just in time", or even merge existing cards into one card. This would mean that multiple applets from different vendors would reside on the same card.

The single biggest problem with this scenario is that of security. How can we guarantee that a simple query to the drivers licence section of the card for identification purposes (say) will not steal money from the card's electronic purse? If new applets are to be down-loaded then how can the vendor of applet A ensure that a competing vendor's applet will not be down-loaded at a later stage and steal information from applet A? Alternatively, applets A and B may trust each other to some extent, and therefore share some information. But if applets B and C enjoy a similar trust relationship, how can A be sure that B will not tell C information which it has obtained from A [Gir99,PB00] ?

Many methodologies for guaranteeing such security have been investigated, but almost all of them involve a trusted "third" party. For example, the bank applet may be signed using a digital signature obtained from the government that certifies that the applet really did originate from the bank in question. The digital certificate is decoded by the card's on-board digital signature chip and the applet is allowed to access the card's electronic purse. But the need for a certification agency and a certification procedure makes this avenue cumbersome.

An alternative methodology that involves no third parties is for card owners to implement a personal security policy using some international standard "language for security". The electronic purse applet installed on the card may come with such a built in security policy which the user is prompted to tailor to his or her needs. Another applet which wishes to access the electronic purse must now pass a challenge determined by the level of security chosen by the card user.

As new applets are added to the card, they are slotted into this set up either explicitly by the card user, or by some implicit default method. The simplest method is to use some form of access control list as is done by the Smart Card for Windows system (`http://www.microsoft.com.smartcard`), which uses simple propositional logic in its access control lists. A more sophisticated approach is to use a hierarchy with the "public" applets at the bottom, the "private" applets at the top, and the others in between these two extremes in some partial order [Gir99,PB00]. But this work does not address the problem of the dynamics of this partial order when a new applet is down-loaded. In particular, how can we be sure that all the previously checked "shared secrecy" conditions are satisfied by the new hierarchy ?

One way to define such a security policy is to use formal mathematical logic and to ensure that the permission granted to down-loaded applications meet certain rigorously defined security criteria expressed as formulae of logic. For example, notions like "agent i trusts agent j" are easily encoded as statements of multi-modal propositional logics, which are now well-established in artificial intelligence research as bases for defeasible reasoning [Shv90], logics of agents [RG93], and logics of authentication [BAN90,Mat97]. Multi-modal logics like Propositional Dynamic Logic [Gol87] have also been used to model the changing states of a program. Finally, propositional bi-modal tense logics give a very simple and elegant model of the flow of time [HC96].

Checking that a down-loaded applet meets the security criteria is now reduced to proving, **on-board**, that an appropriate formula is a theorem of the logic used to code the criteria, since this is the only computer that the customer should trust. Multi-modal logics are particularly well-suited to this task as most of them are decidable. Consequently, the ability to perform automated multi-modal deduction on Java smart cards may be of use in electronic commerce.

But surely multi-modal deduction is simply too difficult to perform on a smart card with extremely limited resources ? After all, even classical propositional logic is `NP`-complete, and most multi-modal logics are actually `PSPACE`-complete!

In [GNg00] automated deduction in bi-modal tense logics was shown to be feasible on a Java smart card. It is reasonably straightforward to extend this work to other multi-modal logics, and hence to logics of knowledge and belief, or to logics of authentication and security. But many of these logics (e.g. PDL) contain operators which are inherently transitive, and transitivity can lead to infinite loops. Here we show that transitivity is not insurmountable by implementing a prover for modal logic **S4**. Thus our work is "proof of principle" that a logic-based security policy could be implemented on current Java cards. As the resources and speed of Java cards skyrocket, the task will only become simpler.

The paper is set out as follows. Section 2 describes the logic **S4** and the basics of modal theorem proving using tableaux. Section 3 explains the design of our prover `CardS4`. Section 6 presents test results and Section 7 presents conclusions.

Acknowledgements: We are grateful to Didier Tollé of Gemplus Australia for donating the hardware necessary to carry out this project.

2 Syntax, Semantics and Tableaux for Modal Logic S4

2.1 Syntax and Semantics for S4

Given a denumerably infinite set of atomic formulae $\texttt{PRP} = \{p_0, p_1, p_2, \cdots\}$, a formulae φ of modal logic is defined using the following BNF grammar:

$$p ::= p_0 \mid p_1 \mid p_2 \mid \cdots$$
$$\varphi ::= \top \mid \bot \mid p \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \Diamond\varphi \mid \Box\varphi$$

$w \models \top$		for every $w \in W$	$w \models \bot$		for no $w \in W$
$w \models p$		if $w \in V(p)$	$w \models \neg A$	if	$w \not\models A$
$w \models A \wedge B$	if	$w \models A$ and $w \models B$	$w \models A \vee B$	if	$w \models A$ or $w \models B$
$w \models A \rightarrow B$	if	$w \not\models A$ or $w \models B$			
$w \models \Diamond A$	if	$(\exists v \in R(w))(v \models A)$	$w \models \Box A$	if	$(\forall v \in R(w))(v \models A)$

Fig. 1. Kripke semantics for **S4**

The standard Kripke semantics for **S4** are as follows. A Kripke frame is a pair $\langle W, R\rangle$ where W is a non-empty set (of worlds) and R is a binary relation over W. A Kripke model $\langle W, R, V\rangle$ is a Kripke frame $\langle W, R\rangle$ augmented with a valuation $V : \mathtt{PRP} \cup \{\top, \bot\} \mapsto 2^W$ mapping each atomic formula and atomic constant to the subset of W where they take the value "true". If $w \in V(p)$ we write $w \Vdash p$ and extend this satisfaction relation to arbitrary formulae in the usual way [HC96] as shown in Figure 1 where for any $w \in W$, $R(w) := \{v \in W \mid wRv\}$.

An **S4**-model is a Kripke model where R is both reflexive $(\forall w \in W)[wRw]$ and transitive $(\forall w_1, w_2, w_3 \in W)[w_1Rw_2 \;\&\; w_2Rw_3 \Rightarrow w_1Rw_3]$.

A formula φ is **S4**-satisfiable if there exists some **S4**-model with some $w \in W$ such that $w \Vdash \varphi$. A formula φ is **S4**-valid if $w \Vdash V(\varphi)$ for every $w \in W$ in every **S4**-model $\langle W, R, V\rangle$.

2.2 Proof Search in S4 Using Tableau Calculi

The problem of deciding whether or not a formula is **S4**-satisfiable is known to be PSPACE-complete [Lad77a,Lad77b] although the best known decision procedures use only $O(n^2.logn)$-space [Hud].

The most popular method for implementing theorem provers for **S4** is to use the tableau method [Fit83,Gor99]. In addition to the standard rules for classical propositional logic, we require only the following two rules, and their duals for $\neg\Box$ and $\neg\Diamond$ obtained via the equivalences $\neg\Box\varphi = \Diamond\neg\varphi$ and $\neg\Diamond\varphi = \Box\neg\varphi$:

$$(\Box\mathbf{S4})\ \frac{X, \Box\varphi}{X, \Box\varphi, \varphi} \qquad\qquad (\Diamond\mathbf{S4})\ \frac{X, \Box Y, \Diamond\varphi}{\Box Y, \varphi}$$

An **S4**-tableau for a finite set of formulae Z is a binary tree of nodes where: the root node contains Z and the children are obtained by an application of some tableau rules for **S4** to the parent node. The rules are applied systematically so that the $\Box$**S4** rule is applied only to a "saturated" node: a node to which all other rules have already been applied. A branch of such an **S4**-tableau is closed if its leaf node contains both φ and $\neg\varphi$ for some formula φ; otherwise the branch is open. The whole **S4**-tableau is closed if every branch in it is closed, otherwise it is open.

Theorem 1 (Soundness and Completeness). *The finite set* $\{\neg\varphi\}$ *has a closed* **S4***-tableau iff the formula* φ *is* **S4***-valid [Fit83,Gor99].*

Corollary 1. *If every* **S4***-tableau for the finite set* $\{\neg\varphi\}$ *is open then the formula* φ *is* **S4***-satisfiable.*

The completeness proof gives a systematic method for proof-search which consists of repeatedly applying all the rules of classical propositional logic, and the ($\Box$**S4**)-rule, until no more applications of these rules is possible. The resulting node then corresponds to a "saturated" world in the underlying Kripke model under construction; see [Gor99] for details. Some $\Diamond$-formula is then singled out for attention from this "saturated" node and a "successor" is created for it using the ($\Diamond$**S4**)-rule.

Naive proof search for a closed **S4**-tableau for some finite set of formulae Z using this systematic method can lead to infinite loops viz $Z = \{\Box\Diamond p, p\}$:

$$\dfrac{\dfrac{\Box\Diamond p, p}{\Box\Diamond p, \Diamond p, p}\,(\Box\mathbf{S4})}{\Box\Diamond p, p}\,(\Diamond\mathbf{S4})$$

One solution for detecting loops involves storing certain nodes as they are encountered, checking for repetitions, and backtracking if necessary.

3 Algorithms

We now describe our algorithm and data structures in more detail.

3.1 Terms

Negated Normal Form: Input formulae are assumed to be in negated normal form (NNF): all implications $\varphi \to \phi$ are rewritten as $\neg\varphi\vee\phi$, and negations are pushed through all connectives so the negation symbol appears only before atomic formulae. This requirement is not restrictive since every formulae can be converted into a logically equivalent NNF formula in linear time [BG98]. The advantage of using NNF is that the formulae in the parse tree of the NNF formula constitute all of the formulae that can appear in any node of the search tree.

Parse tree: A formula is parsed as a tree, where each node has at most two children. The nodes are characterized as `CONJ`, `DISJ`, `ALL`, `SOME` if they are of type $\varphi\wedge\psi, \varphi\vee\psi, \Box\varphi, \Diamond\varphi$ respectively. At the leaves there are literals: atomic formulae or their negations. Each node represents a subformula of the original NNF formula, with the root representing the whole NNF formula. With the tableaux rules above, it can be shown that the subformulae that appear in the parse tree are all the formulae that can appear in the nodes of the search tree. Clearly, the number of nodes in the parse tree is less than or equal to the length of the formula (it is exactly the length of the formula less the number of negative symbols). The number of formulae in any node of the search tree is therefore less than the length of the original NNF formula.

The parse tree is indexed so that each parent node has a smaller index than its children. This simplifies the visit sequence of the parse tree, as can be seen below.

Search Tree: In the sequel we refer to the **S4** tableau as the search tree.

Model Tree: We refer to the underlying Kripke (tree-)model as the model tree. By "path" we mean an R-path in this model tree.

Good Nodes: A node of the search tree is "good" if no application of the $\Diamond$**S4** rule to this node gives a closed branch.

$n_\Diamond$, $n_\Box$, $n_\vee$, $n_\wedge$**:** The number of subformulae of the appropriate type in the original NNF formula.

3.2 Storing Nodes in the Search Tree

The parse tree provides access to the finite list of all formulae that can appear in the nodes of the search tree. Thus each node in the search tree can be represented as a bit string, whose bits indicate whether or not the corresponding formulae is present in the node. This bit string has length equal to the length of the original formula. Thus storing one node in the search tree requires n bits, where n is the length of the original formula.

3.3 Loop Checking

As shown in Section 2.2, an **S4**-tableau can contain an infinite branch. This problem can be solved by noticing that only a finite number of different nodes can appear in a search tree, and by avoiding examining any node twice. Thus if a node has ever been encountered before, it can be safely ignored.

The naive solution is to store each "saturated" node of the branch in a check list since these nodes correspond to worlds in the counter-model under construction, and to explicitly look for repeated worlds. When a world re-appears, we must terminate proof search along this (open) branch and backtrack to the previous application of the ($\Diamond$**S4**)-rule to choose a principal formula $\Diamond\psi$ different from the $\Diamond\varphi$ that lead to this loop; see the ($\Diamond$**S4**)-rule. If all such choices lead to open (or looping) branches, then we backtrack higher up to the previous application of the ($\Diamond$**S4**)-rule, and so on until all avenues have been explored, or a closed **S4**-tableau is found. This, however, is not a practical method, since it would require exponential space to store all the possible nodes of the search tree.

A better method is to check for repetitions of the nodes obtained by the application of the $\Diamond$**S4** rule since these contain the "core" of the new world. That is, the $\Diamond$**S4** rule is a transitional rule which goes from one world in the tree model to one of its successors. Thus the latter method looks for repetitions of the initial configuration of each newly generated world. This also guarantees the solution for the infinite branch problem, yet requires polynomial space. The price is that identical nodes on different branches now have to be treated separately.

3.4 A Straightforward Nondeterministic Algorithm

The following algorithm returns true if a given formula φ_0 of length n is **S4**-satisfiable, and false otherwise. It requires n^4 space.

stack ::= empty
checkList ::= empty
currentWorld ::= $\{\varphi_0\}$
do
  while there are $\wedge$, $\vee$, $\Box$ rules that can be applied **do**
    apply these rules to *currentWorld*
    if the rule applied is an $\vee$ rule **then**
      push (*checkListSize*, left child of the rule) onto *stack*
      currentWorld ::= the right child of rule
    end if
  end while
  if *currentWorld* is inconsistent **then**
    if the *stack* is empty **then**
      return false
    else
      (*checkListSize*, *currentWorld*) ::= pop from *stack*
      resize *checkList* to *checkListSize*
      continue
    end if
  end if
  if *currentWorld* appears in *checkList* **then**
    return true
  end if
  if no ($\Diamond$**S4**)-rules can be applied **then**
    return true
  end if
  non-deterministically pick a formula $\Diamond\varphi$ from *currentWorld*
  apply ($\Diamond$**S4**)-rule using $\Diamond\varphi$ to *currentWorld* to get *newWorld*
  put *currentWorld* into *checkList*
  currentWorld ::= *newWorld*
while *stack* is not empty

Space Complexity It can be seen that *stack* grows by 1 only when the ($\vee$) rule is applied. Thus for one world, we may need to store the maximum of $n_\vee$ possible configurations. Also the transitional rule ($\Diamond$**S4**) which moves from one world to another preserves all the $\Box$-formulae, thus the set of all $\Box$-formulae (the core) in consecutive worlds in a branch of the search tree do not decrease. Therefore there are at most $n_\Box$ different cores in the same branch of the search tree. Also the worlds that have the same core will form a chain in the branch. We need to find the maximum number of worlds in a chain that have the same core.

Since each new world is formed by taking all the $\Box$ formulae from the previous world together with one of the $\Diamond$ formulae, the maximum number of worlds in the chain that have the same core is $n_\Diamond$. Altogether, the maximum number of configurations that we need to store in the stack is $n_\vee \times n_\Box \times n_\Diamond \leq n^3$ (a better approximation is $n^3/3$).

It can be seen that *checkList* contains only the node which is obtained by applying a ($\Diamond$**S4**) rule. It is also resized so that all the nodes it contains are the initial configurations of the worlds in the current search branch. Thus *checkList* is always smaller than or equal to *stack*. Overall, the maximum number of configurations we might need to store in *stack* and *checkList* is of order n^3. Given that a configuration needs n bits, the space complexity of this algorithm is n^4 [Lad77a,Lad77b].

3.5 An Improved Algorithm

First, the non-determinism must be eliminated. Note that a node in the search tree is good if every node that is obtained from it by applying the ($\Diamond$**S4**) rule is good. Thus non-determinism can be eliminated by examining every node obtainable from a node by the ($\Diamond$**S4**) rule. This can be done by storing the world configuration before applying the ($\Diamond$**S4**) rule and keeping the index of the last ($\Diamond$**S4**) rule applied to that configuration.

Second, it is not necessary to store all possible different configurations of a world since we can obtain some configurations from others. Notice that when applying an $\vee$ rule, choosing a different child of the $\vee$ formula gives a different possible configuration. Thus if each subformula of the original formula is named differently then it is possible to move from one node in the search tree to its sibling without storing them. This naming of the nodes of the parse tree forbids common sub-expressions and therefore introduces some redundancy. There may be duplications of the same subformula, but they are named with different indices, so they must be examined independently.

In the following algorithm, the stack stores only the configurations that have been fully expanded with non-($\Diamond$**S4**) rules.

checkList ::= empty
stack ::= empty
currentWorld ::= world consisting of the original formula
do
  while there are $\wedge$, $\Box$**S4** rules that can be applied **do**
    apply these rules to *currentWorld*
  end while
  if there are $\vee$ rules which can be applied **then**
    apply these rules to *currentWorld* such that
    if one child of the $\vee$ formula is already in *currentWorld* **then**
      take that child into *currentWorld*
    else
      take the first child of the $\vee$-formula in to *currentWorld*

```
        end if
        continue
    end if
    if currentWorld is inconsistent then
        mark currentWorld as closed
    end if
    if currentWorld is marked closed then
        if there is another sibling of currentWorld then
            take currentWorld to be that sibling
            reset lastK_index to indicate no ◇S4 rules have been applied yet
            continue
        else if the stack is empty then
            return false
        else
            (lastK_index, currentWorld) ::= pop from stack
            pop from checkList
            mark currentWorld as closed
            continue
        end if
    end if
    if no more (◇S4) rules can be applied then
        if stack is empty then
            return true
        else
            (lastK_index, currentWorld) ::= pop from stack
            pop from checkList
        end if
    end if
    lastK_index ::= the next ◇-formula from currentWorld
    apply the ◇S4 rule to currentWorld to get newWorld
    if newWorld appears in checkList then
        continue
    end if
    put (lastK_index, currentWorld) into stack
    put newWorld into checkList
    currentWorld ::= newWorld
    continue
while stack is not empty
```

Note that now *checkList* and *stack* are of the same size. The *checkList* is actually the core of the world configuration stored at the corresponding location in *stack*. Thus it can be obtained from *stack* by generating the core of each world in *stack*. This gives a more efficient use of space. It can be seen that for one node in the search tree, we need only one location in *stack*. Thus the space requirement is reduced by a factor of $n_\vee$: the maximum number of configurations

stored in *stack* at one time is $n_\Box \times n_\Diamond \leq n^2$, and the space complexity of this algorithm is n^3 (a better approximation is $n_\Box \times n_\Diamond \leq n^2/2$, and the space requirement is $n^3/2$).

4 Data Structures

4.1 The Parse Tree

The parse tree is stored in two byte arrays, one (*childs*) of length $2n$ and the other (*nature*) of length n. The $2i$ and $2i+1$ entries in the first array indicate the children of the i-th node in the parse tree (i.e. the indices of some other nodes in the parse tree, or the atom at that node), while the i-th entry in the second array indicates if the i-th node in the parse tree is a $\Box$, $\Diamond$, $\vee$, $\wedge$, $\neg$ or `PRP`. The $\vee$ and $\wedge$ nodes have two children. The $\Box$, $\Diamond$, $\neg$ and `PRP` nodes have one child, thus the second child for these node is redundant. This is not a severe problem, since the parse tree is fixed through the proving procedure. Note that in case of the $\neg$ and `PRP` nodes, the $2i$ entry of the *child* array contains the atom itself, not the index to another node in the parse tree.

4.2 The Nodes in the Search Tree

As discussed above, each node in the search tree can be represented as a bit string of length n. To examine all the ($\Diamond$**S4**)- rules that can be applied to a node (i.e all the $\Diamond$ formulae in that node), we need to store the index to the last $\Diamond$ formula that has been examined, requiring one more byte.

4.3 The Stack

Nodes are stored in a stack so that other branches in the search tree can be generated from the current branch, and so that a new node can be checked for duplication. This requires searching through all nodes in the stack, and also pushing and popping from the stack.

There are several possible implementations for the stack. The first two options store the stack as an array of bytes, and do not need extra memory for pointers. Since multidimensional arrays are not supported, access to elements of the stack will require some extra computations.

The upper bound of the size of the stack is known as seen above. Thus the stack can be allocated at the beginning of the procedure. We then need not worry about the growth of the stack. However, this is not a practical method, since the stack rarely grows to its theoretical upper bound size. Also, the limited amount of memory on the card will restrict the size of the input formulae. For example, with 512 bytes, the length of the formula will be less than 16 ($16^3/2$ bits = 512 byte). Allocating more than the card's RAM is possible, however it involves swapping to and from the EEPROM and will slow down the proof procedure. Consequently, the card reader usually cannot wait for the card and will throw an Exception.

Another way to implement the stack is to pre-allocate a small stack, and gradually increase its size by a large step when it becomes full. This ensures that the memory is used more effectively. However it still contains redundancy since it allocates more space than required each time it becomes full. Thus the longer formulae will result in larger redundancy. It also involves a lot of copying each time the stack grows.

The stack can also be implemented as a one way link list. Each node in the search tree is an element of the list. This requires extra memory for a pointer to the next element. However this extra memory becomes insignificant for long formulae. There is no redundancy. With this approach, the program has been tested for formulae of length up to 120.

5 Implementation

The program consists of two packages: `card` and `client`. The `client` package contains the classes for parsing the formula and converting it into NNF. Parsing is done by using Javacup (version 1.0j). We need a scanner (`scanner.java`) and a specification (`parser.cup`) for the formula, and Javacup automatically creates the parser. There is also a card proxy which manages the interactions with the card on behalf of the users. The class for testing is also in this package. Note that all of these operations are done off-board on a terminal (PC).

The `card` package contains an interface and two classes that are downloaded onto the card. These are classes `prover` and `State`, and the interface `proverInterface` The interface `proverInterface` provides access to the services offered by the prover. These include loading the formula onto the card and proving. The interface also defines constants that are used by the prover, and are also used in the parsing and converting procedures. The class `prover` contains the codes for the proving procedure. The `prover` object that is loaded onto the card reserves enough space to hold the longest formula. When the formula is put onto the card, it is stored in the object. The user then must explicitly call the `prove` procedures. The `prove` method reads the formula from the object, performs simplifications and then starts looking for a model for the formula. (Note that there is a separation between loading the formula and proving. This is due to the fact that loading is rather complicated, and it is discussed in the next paragraph).

Despite the limitations of the card, the prover is able to work for a number of long formulae. Tests have been conducted for formulae of length up to 120. Passing the input to the card and storing input in the card then requires greater care because communication with the card is not simple. There is an upper-bound for the amount of data that can be transfered in one transmission (approaching 64K is not recommended). Long formulae therefore need to be broken into small pieces. Here, the input arrays are split into pieces of length 32 bytes and each pieces is passed separately to the card, together with its length and position in the original array. Thus the maximum amount of data in one transmission is 34 bytes (one byte for the length and one for the position).

Since the inputs to the `prove` method are not ready in one pass, they need to be stored in the object `prover`, requiring the reservation of space for the longest input. Note that this also implies more time is required for copying the input from EEPROM to RAM in the `prove` procedure.

6 Results

This section shows the average time spent on the card in proving randomly generated formulae of various lengths (from 20 to 120). As can be seen, the time increases with the length of the formulae since longer formulae generally require more stack space, and require more arithmetic operations in the calculations.

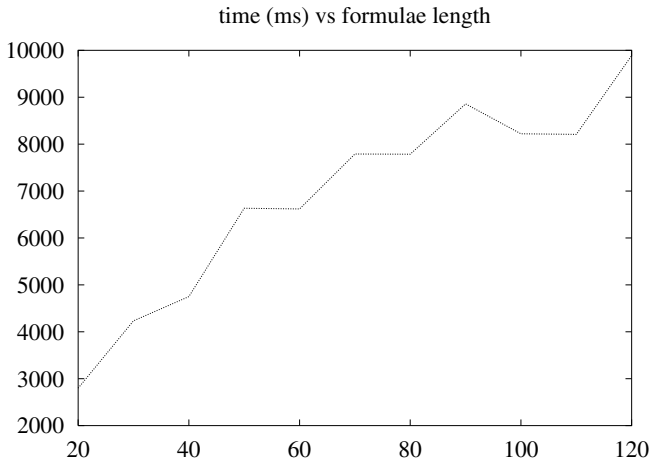

Fig. 2. Time (ms) vs formulae length

7 Conclusions and Further Work

We have shown that even modal logic **S4** can be handled on a Java card. Thus transitive modal logics are not necessarily beyond the scope of Java cards. We now need to invent appropriate logics of permissions and obligations to allow us to capture basic security notions like "trust". This is the subject of further work.

Another method for loop checking is to keep track of certain formula using a history mechanism [Heu99]. We intend to investigate whether such a history mechanism can be easily used in `CardS4`.

References

BAN90. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8:18–36, 1990. 113

BG98. N. Bonnette and R. Goré. A labelled sequent system for tense logic Kt. In *AI98: Proceedings of the Australian Joint Conference on Artificial Intelligence*, LNAI 1502:71-82. Springer, 1998. 115

Fit83. M. Fitting. *Proof Methods for Modal and Intuitionistic Logics*, volume 169 of *Synthese Library*. D. Reidel, Dordrecht, Holland, 1983. 114

Gir99. Pierre Girard. Which security policy for multiapplication smart cards. In *Proceedings USENIX Workshop on Smartcard Technology*, pages 21–28, Chicago, USA, 1999. 112

Gol87. R. I. Goldblatt. *Logics of Time and Computation*. CSLI Lecture Notes Number 7, Center for the Study of Language and Information, Stanford, 1987. 113

Gor99. R. Goré. Chapter 6: Tableau methods for modal and temporal logics. In M. D'Agostino, D. Gabbay, R. Hänle, J. Posegga, editor, *Handbook of Tableau Methods*, pages 297–396. Kluwer Academic Publishers, 1999. 114, 115

GNg00. R Goré and L. D. Nguyen. CardKt: Automated Multi-modal Deduction on Javacards for Multi-application Security. In Proc. Java Card Workshop. Springer LNCS, to appear 2001. 113

Heu99. A. Heuerding. *Automated Deduction in Some Propositional Modal Logics*. Institut für Angewandte Mathematik und Informatik. Universitäte Bern, Switzerland, 1999. 122

HC96. G. E. Hughes and M. J. Cresswell. *A New Introduction To Modal Logic*. Routledge, 1996. 113, 114

Hud. J. Hudelmaier. Improved decision procedures for the modal logics K, T and S4. 114

Lad77a. R. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal of Computing*, 6(3):467–480, September 1977. 114, 118

Lad77b. Richard Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal of Computing*, 6(3):467–480, 1977. 114, 118

Mat97. Anish Mathuria. *Contributions to Authentication Logics and Analysis of Authentication Protocols*. PhD thesis, School of Information Technology and Computer Science, University of Woolongong, Woolongong, Australia, 1997. 113

PB00. P. Girard, J.-L. Lanet, V. Wiels, G. Zanon, P. Bieber, J Cazin. Checking secure interactions of smart card applets. Technical report, Gemplus R&D Centre, 2000. `http://www.gemplus.com/smart/r_d/projects/pacap.htm`. 112

RG93. A. Rao and M. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 318–324. Morgan-Kauffman, 1993. 113

Shv90. Grigori F. Shvarts. Autoepistemic modal logics. In Rohit Parikh, editor, *Theoretical Aspects About Reasoning About Knowledge*, pages 97–109, 1990. 113