cardT^AP: Automated Deduction on a Smart Card

Andrew Slater[1] , Rajeev Goré[1,2*], Joachim Posegga[3], Harald Vogt[3]
{andrews,rpg}@arp.anu.edu.au {posegga,vogt}@tzd.telekom.de
tel: +61-2-62798603 fax: +61-2-62798651

[1] Automated Reasoning Project, Australian National University, 0200, Canberra
[2] Department of Computer Science, Australian National University, 0200, Canberra
[3] Deutsche Telekom AG, Technologiezentrum, IT Security, D-64307 Darmstadt

Abstract. We present the first implementation of a theorem prover which runs on a smart card. The prover is written in Java and implements a dual tableau calculus.[1] Due to the limited resources available on current smart cards, the prover is restricted to propositional classical logic. It can be easily extended to full first-order logic. The potential applications for our prover lie within the context of security related functions based on trusted devices such as smart cards.

Keywords: automated deduction, tableaux, leanTAP, security, java, proof carrying code

1 Smart Cards: the Secure PC of Tomorrow

Smart cards are currently evolving into one of the most exciting and most significant technologies of the information society. Current smart cards on the market are in fact small computers consisting of a processor, ROM and RAM, an operating system, a file system, etc. This miniature computer resides within a tamper proof chip attached to a plastic card such as a credit card. The computer only lacks the typical I/O devices, but can communicate with a smart card reading machine, and has a small EEPROM (Electrically Erasable Programmable Read Only Memory) space analogous to computer disk space. Although their resources are still quite restricted, continuous advances in chip manufacturing will soon enable smart cards with 32 bit processors and up to 128 Kb of memory. Manufacturers are also thinking about integrating small keyboards and LCD displays on these plastic cards. Thus, the next generation of smart cards will be as powerful as PCs were a few years ago.

The evolution of smart card technology resembles the development of computer technology over the last 20 years: the separation of "physics" and "logic". While early computers had to be programmed in machine language because each

* Supported by an Australian Research Council Queen Elizabeth II Fellowship

[1] For further information about the system and obtaining the code see `http://arp.anu.edu.au/~andrews/cardtap`. An interactive simulation version will be made available in the future.

bit of memory and each instruction cycle was valuable, the increase of resources and processing power made it affordable to trade resources for higher level programming concepts and languages. This separation of software and hardware was the basis for the spread of computers into everyday life during this decade.

The same phenomenon is about to take place in smart card technology: as resources and processing power increase, it will become affordable to neglect the optimal use of the card processor and memory. The most promising move in this direction is Java smart cards, where a Java virtual machine is implemented inside the card. The software determining the function of the card is no longer tied to the particular card, but multiple applications can be loaded onto, and removed from, the card as desired.

The primary purpose of smart cards will probably continue to be security-related applications since they serve primarily as a trusted device for their owner. The most important applications to date are of a cryptographic nature like authentication and encryption, e.g. for electronic cash. Future applications running on more complex cards will be able to carry out more complex operations so that the smart card of the future will be a secure, personal computer.

Current smart cards have security-related applications hard-wired onto them. Future smart cards will serve multiple purposes and will be adaptable by downloading one or more applications. Interactions between such applications, and between the card and the outside world therefore become non-trivial. Security issues arise when new and hence untrusted code is introduced to the device, and when known code on the device is requested to perform a sequence of transactions that could result in a violation of security. In both cases we can test for malicious intent by verifying that insecure situations cannot eventuate. Formal logic is not only well-suited for modelling such complex interactions but is also ideal for describing a given security model. Consequently, a trusted, secure, personal device should be able to perform logical reasoning to ensure that the card complies to its owner's security model. A concrete example of the use of formal logic for the purposes of code-safety has been given in the context of proof-carrying code [1]. The significance of implementing a theorem prover capable of operating on a smart card is that we may determine the viability of using techniques from automated deduction to absolutely ensure security in situations where the flexibility of advanced smart card technology thwarts their popular use as a trusted device. At the practical level, untrusted code verification would require an automated deduction system, using an appropriately customised logic, to prove that a sequence of Java byte-code complies with a given security model. Additionally, we suggest that these small verification systems on smart cards will have applications for testing the legality of given input command sequences for a given smart card application and a particular owner's security model.

Automated theorem proving in classical logics is now a mature field, but automation of theorem proving in non-classical logics is a thriving field in artificial intelligence research. Extensions of our system will allow verification using communication protocols expressed in modal and authentication logics [2].

The challenge to implement a verification system for a smart card is to identify theorem proving techniques which will function efficiently in the limited time and space resources on these small machines. Here we describe cardT^4P, the first successful implementation of a theorem prover on a Java smart card.

2 Implementation

cardT^4P is a theorem prover that uses a dual tableaux method based on leanT^4P [3]. leanT^4P is written in Prolog but cardT^4P is written in Java and must therefore function without the underlying backtracking engine. cardT^4P is designed to reside on a smart card with minimal resources, hence it is required that the program executable size be small, in this case less than 2 Kilobytes. Additionally, the stack usage is minimal and heap space, or allocated memory, is also minimal. To implement a proof procedure under these restrictions, cardT^4P simulates a recursive environment powerful enough to perform dual tableaux. While backtracking is not necessary for propositional classical logic, cardT^4P has been implemented for extensions to other logics. The trade off for using this design is efficiency: some work must be repeated as we cannot save all the information computed from intermediate states during the proof procedure. The resulting theorem prover is small enough to reside on the smart card as a Java applet that can be commanded, first to download a formula, and second to determine that formula's theoremhood, using a machine that can communicate with the card.

2.1 The Verification Method: Tableaux

The tableaux method of automated theorem proving is a syntactic refutation system that results in a natural depth first search of a *proof tree* [4]. Each "node" in the proof tree consists of a set of formulæ. A set of tableaux rules specifies how the tree may be constructed, and these rules guide the transformation of sets of formula from a parent node to children. A tableaux may also be described as an "upside-down left-handed sequent system" [5]. The tableaux method begins with a single node (the root) containing the negation of the formula to be tested for theoremhood. If every branch of the ensuing tableaux leads to a contradiction then the root node is deemed unsatisfiable. Thus the original formula is a theorem. The dual-tableaux method avoids the initial negation operation and begins with the formula itself. In order to do this the method provides a "dual" set of tableaux rules to construct the proof tree. Figure 1 shows these rules for the binary operators AND and OR. Rules for their negations may be created by using the DeMorgan laws.

A *path* in the proof tree terminates when no more rules can be applied. The final node of such a path contains a set of literals; that is, a set of atomic formula and negations of atomic formula. The initial formula is falsifiable, and hence a non-theorem, if we can consistently assign "false" to each literal. If a set contains p and $-p$ then such an assignment is impossible, hence we "close" the path when the set contains a tautology, i.e., there is some atomic formula p such that both

$$\text{(AND)}\ \frac{[A\&B]}{[A]\ \text{left child} \quad | \quad [B]\ \text{right child}} \qquad \text{(OR)}\ \frac{[A \vee B]}{[A, B]\ \text{only child}}$$

Fig. 1. Some Dual Tableaux Proof Tree Generation Rules

p and $-p$ are in the set. If every path in the proof tree for a formula closes then that formula has been verified as a theorem.

2.2 Input Formula Specifications

Like leanT^AP, cardT^APonly accepts formulæ in Negated Normal Form (NNF). This means that negation symbols must be "pushed in" toward the atomic propositions, and this can be done with a simple polynomial-time algorithm. cardT^APalso uses Reverse Polish Notation (RPN). The RPN format reduces the amount of space required for the description of the formula and makes the parsing process simpler. Additionally, simple optimisations can be performed efficiently when translating any given propositional formula into NNF RPN. Some of the excess EEPROM space not used by executable code is used as virtual memory to store the given formula. Once the formula is downloaded onto the Smart Card it is seen as an EEPROM file. Since EEPROM access is slow, the efficiency of accessing the formula can be enhanced by using a small buffer, or cache, in local memory as a "window" into the formula.

2.3 Prover Execution

The theorem proving process takes advantage of the RPN format of the formula by viewing it as the parse tree for that formula. The dual tableaux algorithm allows us to scan the parse tree and use a simple strategy to generate and traverse the tableaux *proof tree*. By requiring the formula to be in NNF the only branching tableaux rule needed is one for conjunctions. Therefore, in the generated proof tree, branches only occur at conjunctions and the arguments to disjunctions are interpreted as a list of additional subformulæ to be processed. The prover traverses each *path* from the root of the proof tree to the leaves. During each traversal *state information* about the propositional variables for that path is accumulated and checked for path closure i.e. a tautology in the path. The state information is reset before traversing any path.

Since the method is tree based it could be solved nicely with recursion, however this prover cannot be recursive as it would quickly exhaust the available stack space. There are also space restrictions on how much information we can retain at any step in the verification. Typically a dual tableaux theorem prover is capable of remembering or copying the accumulated state information at the conjunction nodes in the proof tree before it takes the first branch. By doing so it can return to that branching point and traverse the alternate path using the previously saved state information. cardT^AP does not have enough memory space to arbitrarily store a 'state' for every branch point. cardT^AP simulates the depth first traversal by requiring that every possible path in the proof tree, from root to leaf, is traversed separately, and that during each traversal the state information for that path is accumulated. To achieve this cardT^AP maintains a simple binary map of the conjunctions encountered in the proof tree. This *conjunction map* allows the prover to be directed through each branching possibility, in a left to right order, and thus explore every path iteratively by using the map.

Disjunctions do not cause branching in the proof tree, however every argument must be made available while searching for termination in a path. The strategy here is to process the first disjunct and save the position of the second in case it is needed later. cardT^AP maintains a *disjunction list* so that, if the prover reaches a node which does not close or branch, it can search the disjunction list for further subformulæ to process. Each such subformula is the second argument to some disjunction previously encountered in the current path. If a disjunct is available then that subformula may be immediately processed as if it were attached to the current node. The list actually holds the location of subformulæ so that they may be easily found within the main formula. Each path, from root to leaves, must generate a new disjunction map as the path is traversed.

The proof tree traversal is performed within a loop which terminates when:

(i) A path ends without closure and no remaining disjunctive arguments remain to be processed. Since the path has not closed during the traversal the formula is a non-theorem.
(ii) All paths in the proof tree have been traversed (the conjunction map is exhausted), and have closed. In this case the formula is a theorem.

2.4 Example Execution

The following example illustrates how the algorithm tests the formula $(a\&b\&c)+(-a+-b+-c)$, which is derived from a DeMorgan equivalence. Here we use the same notation as is accepted by cardT^AP where Boolean AND is &, Boolean OR is $+$, and negation is $-$. Note also that our NNF RPN notation also assumes that both AND and OR are binary operators. The formula we give to cardT^AP is $+\&\&abc++-a-b-c$. While we could not afford the resources to implement higher level data structures, such as lists, we will use these concepts in the example to simplify the description and note that they were easily simulated with equivalent non-recursive data structures in the actual code.

The first path taken by cardT^AP is illustrated in Figure 2 by the unbroken arrows. We will call the conjunction map the *AND map*, and use it as a list containing directions of either *Left* or *Right*. The list storing secondary arguments to disjunctions is called the *OR list* and this stores the position of the disjunctive subformulæ that we may want to process later. Before the process starts both the *AND map* and the *OR list* are empty, and the state information indicates that no propositional variables have been encountered. The first token is $+$ so the

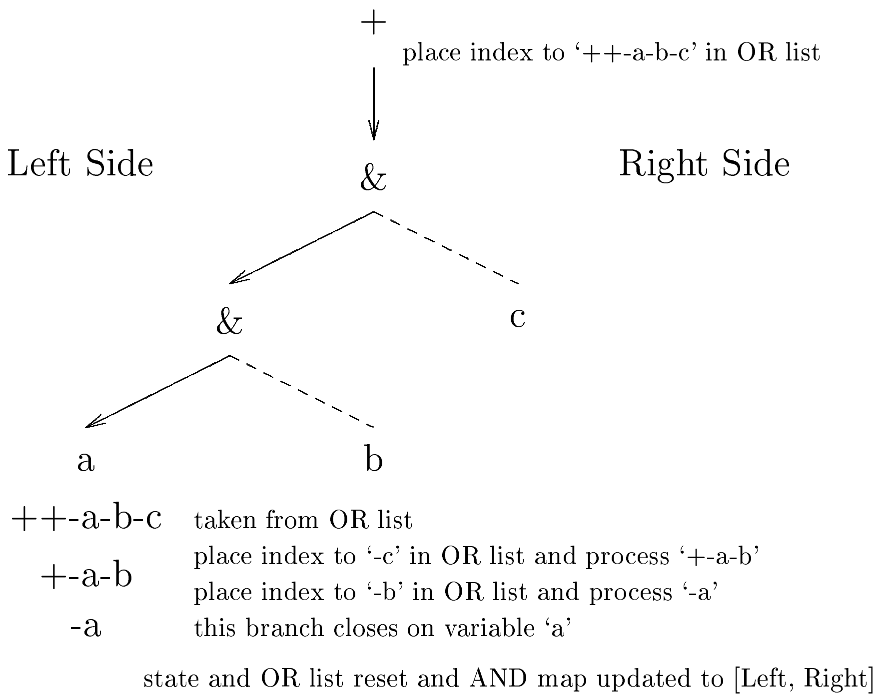

Fig. 2. The traversal of the first path in the proof tree for the formula $+\&\&abc++-a-b-c$. The first path follows the default *AND map*, [*Left*, *Left*].

second argument in the disjunction, which is the subformula $++-a-b-c$, has it's position in the formula placed in the *OR list*. This leaves the first argument of the main disjunction, $\&\&abc$, to be processed immediately. This subformula is a conjunction of $\&ab$ and c. Since the tree traversal defaults to taking the left branch first we record the action by adding this direction to the end of the *AND map* (initially empty). We then move along the formula to process the first conjunct $\&ab$. Again a conjunction is encountered and we process the left branch and record the action by placing the direction taken (*Left*) at the end of the *AND map*. We are now left with the literal token a which is recorded as being encountered in our state environment. It is obvious that we cannot continue the traversal, and that the proof tree has not closed yet, so we must refer to the *OR list*. The *OR list* holds one element which we take out and proceed processing again in a similar nature. In this case, however, every time we encounter a disjunction we place an index to the second argument in the *OR list* and then process the first argument. Eventually we find the literal token $-a$ which closes this path.

Since the first path closed and the *AND map* is non-empty we must evaluate further possible paths to complete the proof tree. To ensure that all paths are taken the *AND map* is updated in the following way:

(i) Remove all *Right* entries at the end of the list, since we now have investigated these paths.
(ii) If entries remain then change the last entry, which must be *Left*, to *Right*.
(iii) If no entries remain in the *AND map* then we are done and the formula is a theorem.

Note that in case (iii) we are left with a theorem since the algorithm terminates with a result of failure as soon as we reach the end of an open path. Each time we begin traversal of a new path we reset the *OR list*, reset the propositional variable state information, but retain the *AND map*. When a path is traversed the *AND map* is used as a guide through the tree, always defaulting to left when a new conjunction is encountered.

3 Experimental Results

The theorem prover was developed and initially tested in a simulated environment. The simulated execution evaluated the theoremhood of all test data correctly. The corresponding execution times indicated that the inclusion of optimisation techniques, such as windowing the EEPROM stored formula in a local memory buffer, enhanced the performance significantly. This version of the theorem prover, while operating in hardware limitations close to that of the Java smart card later used, had to be modified to execute on that smart card. The limitations of the Java smart card used did not permit any extra enhancements to be included due to program size, neither did it permit much separation of code due to the stack usage of procedure calls. It is envisaged that the next generation of Smart Cards entering the market in mid-98 will not require these modifications.

We successfully ran cardT^4P on a smart card provided by Schlumberger [6] implementing JavaCard API V1.0 [7]. This card handles applications of up to 2.8K and offers approximately 200 bytes of main memory during run time. Our test formulæ consisted of 17 theorems of propositional logic [8] converted into NNF RPN. We also tested some non-theorems, obtained by mutating some of these 17 theorems.

The prover applet is limited by statically defined restrictions on the length and complexity of the formula. This is required as a dynamic interpretation of formulæ may exhaust the resources of the smart card. In testing mode the formula could use up to 26 distinct variables, could have at most 20 disjunctions, and could have at most 20 nested conjunctions. The theorem length was limited to a maximum of 126 symbols. With larger hardware resources available on the card, these limitations may be safely extended.

Each formula was loaded onto the card individually and tested using the proof procedure described above. The interaction was performed through *LoadSolo*, a simple tool for communicating with the card, which came with the Cyberflex Development Kit [6]. cardT^4P returned an answer code indicating whether or

Name	RPN NNF Formula	Execution Time
	Pelletier's 17 theorems:	
P1	+&&p-q&-qp&+-pq+q-p	21.9 s
P2	+&pp&-p-p	6.9 s
P3	++-pq+-qp	2.0 s
P4	+&&-p-q&-q-p&+pq+qp	22.1 s
P5	+&+pq&-p-r+p+-qr	8.7 s
P6	+p-p	1.7 s
P7	+p-p	1.7 s
P8	+p&+-pq-p	3.2 s
P9	+&pq+&-pq+&-p-q&p-q	27.6 s
P10	++&pq&-p-q+&q-r+&p&-q-r&r+-p-q	93.0 s
P11	+&pp&-p-p	7.0 s
P12	+&+&+&pq&-p-qr&+&p-q&-pq-r+&p+&qr&-q-r&-p +&q-r&-qr&+&+&pq&-p-q-r&+&p-q&-pqr +&p+&q-r&-qr&-p+&qr&-q-r	-
P13	+&&-p+-q-r+&-p-q&-p-r&+p&qr&+pq+pr	110.0 s
P14	+&+&p-q&-pq+&-qp&q-p&+&pq&-p-q&+q-p+-qp	160.0 s
P15	+&&p-q&p-q&+-pq+-pq	22.1 s
P16	++-pq+-qp	2.0 s
P17	+&&&p+-qr-s+&&p-q-s&&pr-s&+s+-p &q-r&+s+-pq+s+-p-r	-
	Non-theorems:	
N1	&-pp	2.0 s
N2	+pp	2.4 s
N3	p	1.8 s
N4	-p	1.8 s
N5	+&&p-q&-qp&+-pq+qp	7.2 s
N6	+&+p-q&-qp&+-pq+qp	25.3 s
N7	+&pp&p-p	2.7 s
N8	++-pq+q-p	5.0 s
N9	+p&+-pqp	6.7 s
N10	+&pq+&-p-q+&-p-q&p-q	7.3 s

Table 1. Median smart card execution times using the test data set.

not the formula was a theorem. All measurements were made by hand: each theorem was proved 3 times, the fastest and the slowest times were discarded. The run-times presented in Table 1 include communication overhead.

Timing constraints enforced by the card, due to requirements of ISO smart card standards, either raised an exception or garbled communication during some of the longer computations. These problems could be partially solved by interspersing commands which send data from the card to the reader. These modifications are sufficient for proving the shorter theorems, theorems P10 and P13, but for the larger ones, like theorem P14, additional modifications had to be made. All modifications concern only additional commands for communication. Theorems P12 and P17 could not be proved with any version of card$T^A\!P$.

4 Conclusions and Outlook

The current version of card$T^A\!P$ is a propositional logic theorem prover written in Java. The methodology is essentially the same as that of the Prolog first order logic theorem prover lean$T^A\!P$. With greater available resources on smart cards, an extension of card$T^A\!P$ to first-order logic is straightforward. Additionally the tableaux method is easily extended to incorporate modalities [9], allowing modal and in particular temporal notions in the logic [10]. These extensions are of interest with respect to authentication logics and the use of verification to enhance the security of smart cards since authentication logics are based on modal logics [2].

Our experiments yielded very slow execution times which reflect the computational power of current smart card technology, but we succeeded in demonstrating that it is possible for automated deduction software to reside and execute on a smart card. While smart card technology is not yet powerful enough to verify potential applets, it is approaching the capability of verification of communication sequences based on logical formalisations of the communication protocol.

The simplicity of the Java code is a direct result of the tableau methodology which nicely partitions the problem into multiple branches, each of which can be explored using the limited resources individually. In contrast, "global" procedures such as resolution would not have been as well suited since they accumulate information rather than partitioning it.

References

1. George Necula and Peter Lee. Proof carrying code. Technical Report CMU-CS-96-165, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, September 1996.
2. Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.
3. Bernhard Beckert and Joachim Posegga. lean$T^A\!P$: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.
4. Melvin Fitting. *First Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.
5. Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. John Wiley and Sons, 1987.
6. Schlumberger Inc. Cyberflex. http://www.cyberflex.austin.et.slb.com, 1997.
7. JavaSoft Inc. Javacard API. http://www.javasoft.com/products/javacard/, 1997.
8. Francis J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.
9. Bernhard Beckert and Rajeev Goré. Free variable tableaux for propositional modal logics. In D Galmich, editor, *Proceedings of the International Conference on Theorem Proving with Analytic Tableaux and Related Methods*, volume Lecture Notes in Artificial Intelligence of *LNCS*, pages 91–106. Springer, May 1997.

10. Nicolette Bonnette and Rajeev Goré. A labelled sequent system for tense logic $\mathbf{K_t}$. In *These proceedings*, 1998.

This article was typeset using the LaTeX macro package with the LLNCS2E class.