Matching Midlet's Security Claims with a Platform Security Policy using Automata Modulo Theory[*]

Fabio Massacci   Ida Siahaan
DIT, Universitá di Trento, - Italy
name.surname@dit.unitn.it

Abstract

Model-carrying code and security-by-contract have proposed to augment mobile code with a claim on its security behavior that could be matched against a mobile platform policy before downloading the code. In this paper we show that it is possible to define very expressive policies — essentially with infinite cases — that can capture realistic scenarios (e.g. "only connections to urls starting with https") while keeping the task of matching computationally tractable. The key idea is the concept of *Automata Modulo Theory* ($\mathcal{AMT}$). $\mathcal{AMT}$ is an extension of Büchi Automata (BA), suitable for formalizing systems with finitely many states but infinitely many transitions.

The second contribution is a decision procedure for matching the mobile's policy and the midlet's security claims expressed as $\mathcal{AMT}$ by mapping the problem into a variant of on-the-fly product and emptiness test from automata theory. The tractability limit is essentially the complexity of the satisfiability procedure for the theories where most practical policies require only polynomial time decision procedures.

1 Introduction

The paradigm of pervasive services [1] envisions a nomadic user traversing a variety of environments and seamlessly and constantly receiving services from other portables, handhelds, embedded or wearable computers. When traversing environments the nomadic user does not only invoke services according a web-service-like fashion (either in push or pull mode) but also download *new* applications that are locally available. These *pervasive client downloads* will appear because service providers will try to exploit the computational power of the nomadic devices to make a better use of the services available in the environment [8].

Bootstrapping and managing security of services in this scenario is a major challenge because the current security model adopted for mobile phones (the JAVA MIDP 2.0) is the exact negation of this business idea: mobile code is run if its origin is trusted (i.e. digitally signed by a trusted party). The level of trust of determines the privileges of the code and untrusted code is forbidden to have any interaction with the environment (which is precisely what we want to do).

Even if we accept the signature, we still have another problem: there is no semantics attached to the signature. This is a problem for both code producers and consumers.

From the point of view of mobile code consumers they must essentially accept the code as-is without the possibility of making informed decisions, while from code producer they

[*] We would like to thank N. Bielova for implementing the matching prototype and the anonymous reviewers for the insightful comments that help to improve the presentation. Research partly supported by the EU with project IST-2004-27004 S3MS - www.s3ms.org.

produce code with unbounded liability. They cannot declare which security actions the code will do, because by signing the code they essentially declare that they did it. Consequently, injecting an application in the mobile market is a time consuming operation as developers must convince the operators that their code is not harmful.

Of course we could not ask ourselves any questions and just monitor everything. We apply a *security reference monitor* which observes execution of a target system and halts that system whenever it is about to violate some security policy of concern [21, 9]. While security monitors remains the bottom-line action, we could be more effective if we start asking some questions about the code.

The first traditional question is whether the code satisfies some pre-defined policy. The Bytecode verifier in Java does exactly this first preliminary check. More advanced techniques based on Proof-Carrying Code [17, 16] extend the scope of what can be actually checked. One of the limitation of the approaches based on language-based security is that the policy is tied to the programming language (as the name itself suggest) and therefore it is difficult to customize the policy on a per-user base.

We need to lift the question to a more flexible one: does the code satisfy a user-defined policy? In general case this is equivalent to arbitrary software verification which is not practical for pervasive downloads. However the idea behind model-carrying code [23] and security by contract [7] is that code should come accompanied with a "digest" (a security model or a security contract) that represents its essential security behavior. Then one only needs to check the latter against the user predefined security policies.

The next question in the pipe is how do we know that the security claims are actually true on the code. Again one possible solution is to use proof carrying code or trust relations and digital signatures. As noted in [7] the presence of the security contract provides a semantics to a digital signature, which was not present beforehand. When binding together the code and the contract the signer takes liability for the security claims. Thus we can assume that developers could use in-line monitors, static analysis or other off-line tools to guarantee the compliance of the code with the contract[1].

As the problem seems essentially solved, neither [23] nor [7] has actually solved the problem of matching the security claims of the code with the security desires of the platform. Matching can be done statically (e.g. a developer checking its claims on a variety of Vodafone's default policies) or at run time (e.g. a mobile platform deciding to actually download a midlet and run it). In [23] and in other companion papers only finite automata have been proposed and they are too simple to express even the most basic security requirement occurring in practice: a basic security policy such as only allows connections starting with "https://" already generates an infinite automaton. In [7] only a meta-level algorithm has been given showing how one can combine policies at different levels of details (such as object, session or multisession). The actual mathematical structure and algorithm to do the actual matching of the mobile's policy and the midlet's security claims is not specified.

2 The Contribution of this Paper

In this paper we provide a formal model and an algorithm for matching the claims on the security behavior of a midlet (for short *contract*) with the desired security behavior of a platform (for short *policy*) for realistic security scenarios (such as the "only https connections" mentioned afore).

[1] Some surveys of usable technologies are available at `www.s3ms-project.org`.

The formal model used for capturing contracts and policies is based on the novel concept of *Automata Modulo Theory* ($\mathcal{AMT}$). $\mathcal{AMT}$generalize the finite state automata of model-carrying code [23] and extends Büchi Automata (BA). It is suitable for formalizing systems with finitely many states but infinitely many transitions by leveraging on the power of satisfiability-modulo-theory (SMT for short) decision procedures. $\mathcal{AMT}$ enables us to define very expressive and customizable policies as a model for *security-by-contract* as in [7] and model-carrying code [24] by capturing the infinite transition into finite transitions labeled as expressions in defined theories.

The second contribution is a decision procedure (and its complexity characterization) for matching the mobile's policy and the midlet's security claims that concretize the meta-level algorithm of security-by-contract [7]. We map the problem into classical automata theoretic construction such as product and emptiness test.

Since our goal is to provide this midlet-contract vs platform-policy matching on-the-fly (during the actual download of the midlet) issues like small memory footprint, and effective computations play a key role. We show that the tractability limit is the complexity of the satisfiability procedure for the underlying theories used to describe labels: we use NLOGSPACE and linear time algorithms for the automata theoretic part [14] with oracle queries to a decision procedure solver[2]. Out of a number of requirements studies it seems that most of the policies of interests can be captured by theories which only requires PTIME decision procedures.

We have further customized the decision algorithm in order to exploit the characteristics features of security policies. $\mathcal{AMT}$ is agreeable for security policies and one can use security automata á la Schneider which can be mapped to a particular form of $\mathcal{AMT}$ (with all accepting states and an error absorbing state) for which particular optimizations are possible. Security automata specified transitions as a function of the input symbols which can be the entire system state. Although infinite transition systems are really not of much practical relevance, our security automata definition does not explicitly preclude them as states can be infinite. However, $\mathcal{AMT}$ differs from security automata in transitions which are environmental parameters rather than system states.

In the next section we present an overview of security-by-contract framework providing a description of the overall life-cycle of mobile code in this setting and we also describe mobile applications security requirements and contract specification as motivations for $\mathcal{AMT}$. Then, we introduce $\mathcal{AMT}$ and the corresponding automata operation (§4) and some specific issues to be considered in $\mathcal{AMT}$. In §5 we describe an approach for lifting finite state tools to $\mathcal{AMT}$.

3 Intuitions and Motivations

To understand better the motivation behind this work we consider how a midlet-lifecycle would be in the model-carrying code or security-by-contract paradigms [23, 7] as shown in Fig. 1. In the initial stage, namely contract/policy authoring and matching phase, a developer gets hold of a policy template made available by a mobile phone operator, or his contracting companies.

[2] In a nutshell $\mathcal{AMT}$ makes reasoning about infinite state systems possible without symbolic manipulation procedures of zones and regions or finite representation by equivalence classes [13] which would not be suitable for our intended application i.e. checking security claims before a pervasive download on a mobile phone.

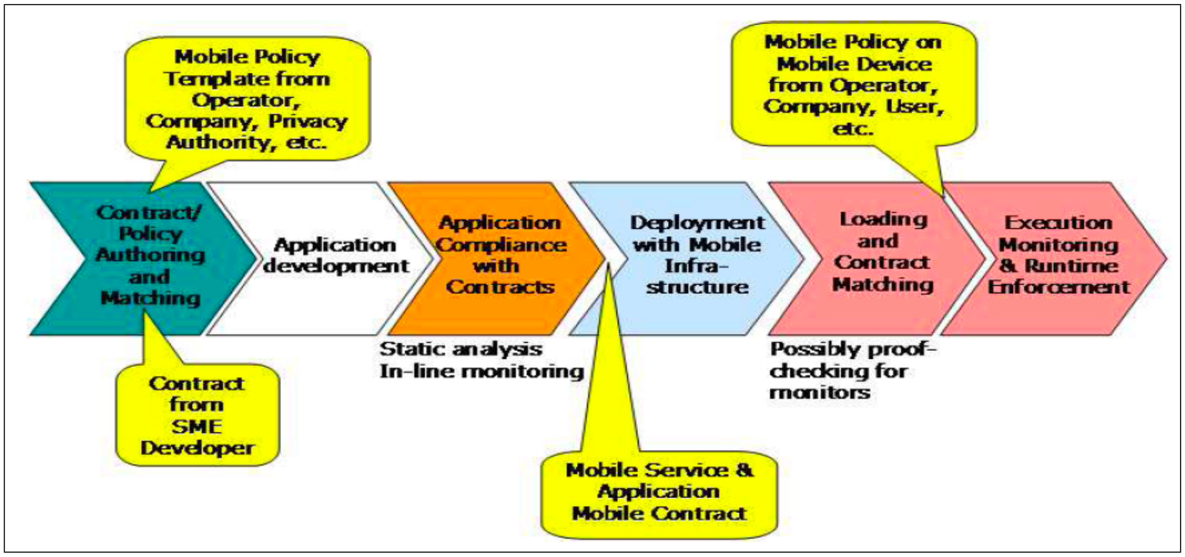

Mobile Policy Template from Operator, Company, Privacy Authority, etc.
Mobile Policy on Mobile Device from Operator, Company, User, etc.
Contract/ Policy Authoring and Matching
Application development
Application Compliance with Contracts
Deployment with Mobile Infra-structure
Loading and Contract Matching
Execution Monitoring & Runtime Enforcement
Static analysis In-line monitoring
Possibly proof-checking for monitors
Contract from SME Developer
Mobile Service & Application Mobile Contract

Figure 1: Application/Service Life-Cycle

Example 1 *The Personal information management (PIM) system on the phone has the ability to manage appointment books, contact directories, etc. in electronic form. A privacy conscious user may restrict network connectivity by stating a policy rule: "After PIM was opened no connections are allowed". This contract permits executing* `Connector.open()` *method only if* `PIM.openPIMList()` *method was never called before.*

After, or better during the application development, the mobile code developers are responsible to provide a description of the security behavior that their code finally provides. Such a code can then undergo a formal certification process which can be done by the developer's own company, the mobile operator, phone manufacturer, or any other third party for which the application has been developed. By using suitable techniques such as static analysis or monitor in-lining or proof carrying code the code is certified to comply with the developer's contract. Subsequently the code and the contract are sealed together with a digital signature and shipped for deployment.

Remark 3.1 *In the sequel, we use the word* policy *for a platform security policy. We use the word* contract *for security claims made by a code.*

At deployment time of "pervasive download" the target platform first checks that the application security claims stated in the contract comply with the platform policy, namely loading and contract matching phase. If a trusted signature is found, the application can be run without further ado.

Example 2 *The policy of an operator may only require that "After PIM was accessed only secure connections can be opened". This policy permits executing* `Connector.open(string url)` *method only if the started connection is a secure one i.e.* `url` *starts with "https://".*

Matching should succeed if and only if by executing an application on the platform every behavior of the application that satisfies its contract also satisfies the platform's policy. If matching fails but we still want to run the application, then we use either a security monitor in-lining into the code or run-time enforcement of the policy by running the application in parallel with a reference monitor that intercepts all security relevant actions. However with a constrained device where cpu cycles means also battery consumption, we need to minimize the run-time overheads as much as possible.

Typically the policy will cover a number of issues such as file access, network connectivity, access to critical resources or secure storage. A single contract can be seen as a list of disjoint claims (for instance rules for connections). An example of rule for sessions regarding PIM and connection is shown in Ex. 1, it could be one of the rules of a contract. Another example is rule for a the methods invocation of a Java object as shown in Ex. 2. This example can be one of the rules of a policy. Both examples describe safety properties, which are the common properties we want to verify. Although most properties are safety properties, liveness properties also exist as shown in Ex. 3.

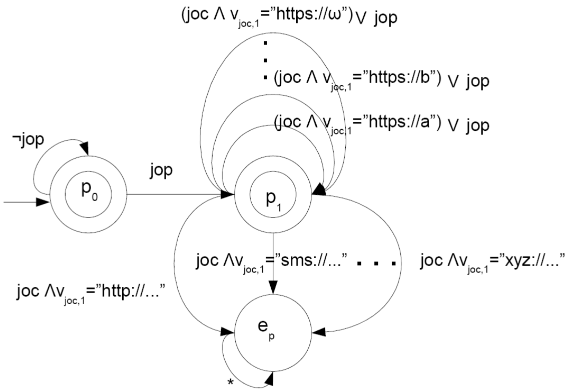
Example 3 *If the rule is that "The application uses all the permissions it requests" then for each permission p at least one reachable invocation of a method permitted by p must exist in the code. For example if p is* `io.Connector.http` *then a call to method* `Connector.open()` *must exist in the code and the url argument must start with "http". If p is* `io.Connector.https` *then a call to method* `Connector.open()` *must exist in the code and the url argument must start with "https" and so on for other constraints e.g. permission for sending SMS.*

To represent the security behavior, provided by the contract and desired by the policy, a system can be represented as an automata where transitions corresponds to platform APIs as suggested by Erlingsson [9, p.59] and Sekar et al. [23]. In this case, the operation of matching the midlet's claim with platform policy is a classical problem in automata theory: language inclusion. Namely, given two automata Aut^C and Aut^P representing respectively the formal specification of a contract and of a policy we have a match when the language accepted by Aut^C (i.e. the execution traces of the midlet) is a subset of the language accepted by Aut^P (i.e. the acceptable traces for the policy). Assuming that the automata are closed under intersection and complementation, the matching problem can be reduced to an emptiness test:$\mathcal{L}_{\mathrm{Aut}^C} \subseteq \mathcal{L}_{\mathrm{Aut}^P} \Leftrightarrow \mathcal{L}_{\mathrm{Aut}^C} \cap \overline{\mathcal{L}_{\mathrm{Aut}^P}} = \emptyset \Leftrightarrow \mathcal{L}_{\mathrm{Aut}^C} \cap \mathcal{L}_{\overline{\mathrm{Aut}^P}} = \emptyset$. In other words, there is no behavior of Aut^C which is disallowed by Aut^P. If the intersection is not empty, any behavior in it corresponds to a counterexample.

The problem with the naive encoding into automata is that even the most basic security policy such as the one we have shown in Ex. 1 and Ex. 2 will lead to automata with infinitely many transitions if we spell out all possible values of the instantiated parameters. Fig.2a represents an automaton for Ex. 2. Starting from state p_0, we stay in this state while PIM is not accessed (jop). As PIM is accessed we move to state p_1 and we stay in state p_1 only if the started connection Connector.open(string url) method is a secure one i.e. url starts with "https://" or we keep accessing PIM (jop). We enter state e_p if we start an unsecure connection Connector.open(string url) e.g. url starts with "http://" or "sms://" etc. These examples are from a Java VM. Since we do not consider useful to invent our own names for API calls we use the `javax.microedition` APIs (though a bit verbose) for the notation that is shown in Fig.2b.

4 Automata Modulo Theory

The theory of *Automata Modulo Theory* ($\mathcal{AMT}$ for short) is a combination of the theory of Büchi Automata (BA) with the Satisfiability Modulo Theories (SMT) problem. In contrast to classical security automata we prefer to use BA because besides safety properties, there


(joc ∧ v_joc,1="https://ω")∨ jop
(joc ∧ v_joc,1="https://b") ∨ jop
(joc ∧ v_joc,1="https://a") ∨ jop
¬jop
jop
p0
p1
joc ∧v_joc,1="sms://..." . . .
joc ∧v_joc,1="xyz://..."
joc ∧v_joc,1="http://..."
ep
*

(a) Infinite Transitions Security Policies

$$\begin{aligned}
joc(v_{joc,1}) &\doteq \texttt{io.Connector.open}(url)\\
jop &\doteq \texttt{pim.PIM.openPIMList}(\ldots)\\
q &\doteq \texttt{io.Connector.}type\\
&\quad\ \text{is protocol type e.g. “http”}\\
pr(q) = type &\doteq \text{permission } q\text{is for protocol } type\\
p(url) = type &\doteq url.\texttt{startsWith}(type)
\end{aligned}$$

(b) Abbreviations for Java APIs

are also some liveness properties which have to be verified, e.g. Ex 3. SMT problem, which decides the satisfiability of first-order formulas modulo background theories, pushes the envelope of formal verification based on effective SAT solvers. The theories of interest are, the theory of difference logic $\mathcal{DL}$ the theory $\mathcal{EUF}$ of equality and uninterpreted functions, the quantifier-free fragment of Linear Arithmetic over the rationals $\mathcal{LA}(\mathbb{Q})$ and that over the integers $\mathcal{LA}(\mathbb{Z})$. As in [4] we are particularly interested in the combination of two or more simpler theories.

Example 4 *When comparing a policy and a contract where* ***protocol(url)=''https''*** *and* ***port(url)=8080****, we do not need to extract a protocol from the url. It is enough that we deal with protocol and port as uninterpreted functions and apply $\mathcal{EUF}$.*

Example 5 *We can use $\mathcal{LA}(\mathbb{Q})$ when the actions of the policy or the contract sets limits on resources such as no communication allowed if battery level falls below 30%.*

This is not a complete list of theories, in sequel we consider only theories $\mathcal{T}$ such that the $\mathcal{T}$-satisfiability of conjunctions of ground literals is decidable by a $\mathcal{T}$-solver [18].

We assume the usual notion of signature Σ with variables $V = \{x, y, z, v, ...\}$, function symbols $\mathcal{F} = \{c, d, f, g, ...\}$ and predicate symbols $\mathcal{P} = \{p, q, ...\}$. Terms and formulae are defined in the usual way over the boolean connectives $\neg, \vee, \wedge$. A first-order Σ-structure $\mathcal{A}$ consists of a set A of elements as *domain*, a mapping of each n-ary function symbol $f \in \Sigma$ to a total function $f^{\mathcal{A}} : A^n \to A$, a mapping of each n-ary predicate symbol $p \in \Sigma$ to a relation $p^{\mathcal{A}} \subseteq A^n$.

Let $\mathcal{A}$ denote a structure, ϕ a formula, and $\mathcal{T}$a theory, all of signature Σ. $(\mathcal{A}, \alpha) \models \phi$: ϕ is true in $\mathcal{A}$ under the variable assignment $\alpha : V \to A$. ϕ is satisfiable in (satisfied by) $\mathcal{A}$: $(\mathcal{A}, \alpha) \models \phi$ for some α. We denote by E as a set of formulae.

Definition 4.1 (Automaton Modulo Theory ($\mathcal{AMT}$)) *A tuple $A_{\mathcal{T}} = \langle E, S, q_0, \Delta_{\mathcal{T}}, F\rangle$ where E is a set of formulae in the language of the theory $\mathcal{T}$, S is a finite set of states, $q_0 \in S$ is the initial state, $\Delta_{\mathcal{T}} : S \times E \to 2^S$ is labeled transition function, and $F \subseteq S$ is a set of accepting states.*

We say that $(s, e, t) \in \Delta_{\mathcal{T}}$ when $t \in \Delta_{\mathcal{T}}(s, e)$. The intuition in $\mathcal{AMT}$ is that a system can be represented with variables representing parameters over invoked methods. Hence, variables are only environment variables and we can represent them as edge variables without memory. This observation leads to a subtle difference between traditional state variables in infinite systems and edge variables. For example a guard $x < 3$ in classical hybrid automata for state variable x means that after taking the transition x must be smaller than 3. In our case, since x is some external parameter of a Java method, this means that this edge will be taken each time the Java method is invoked with a value of x smaller than 3.

Definition 4.2 ($\mathcal{AMT}$ run) *Let $A_{\mathcal{T}} = \langle E, S, q_0, \Delta_{\mathcal{T}}, F\rangle$ be an automaton modulo theory $\mathcal{T}$. A* run modulo $\mathcal{T}$ *of $A_{\mathcal{T}}$ on a finite (respectively infinite) word (trace)$w = \langle \alpha_0, \alpha_1, \alpha_2, \ldots\rangle$ of assignments is a sequence of states $\sigma = \langle s_0, s_1, s_2 \ldots\rangle$, such that:*

1. *$s_0 = q_0$*
2. *there exists expressions $e_i \in E$ such that $s_{i+1} \in \Delta_{\mathcal{T}}(s_i, e_i)$ and $(\mathcal{A}, \alpha_i) \models e_i$ is satisfiable for all $i \in [0 \ldots |w|]$ (resp. $i \in N$).*

A finite run is accepting if $s_{|w|}$ goes through some accepting states. An infinite run is accepting if the automaton goes through some accepting states infinitely often as in BA.

A trace is a *word* in the language of $\mathcal{AMT}$. The set α^* denotes the set of finite words over α while the set α^ω is the set of infinite words over α. The language of $\mathcal{AMT}$ is a set of words. The transition function of $A_{\mathcal{T}}$ may have many possible transitions for each state and expression, hence $A_{\mathcal{T}}$ may be non-deterministic.

Definition 4.3 (Deterministic $\mathcal{AMT}$) *$A_{\mathcal{T}} = \langle E, S, q_0, \Delta_{\mathcal{T}}, F\rangle$ is a* deterministic automaton *modulo theory $\mathcal{T}$ iff for every $q \in S$ and every $q_1, q_2 \in S$ and every $e_1, e_2 \in E$, if $q_1 \in \Delta_{\mathcal{T}}(q, e_1)$ and $q_2 \in \Delta_{\mathcal{T}}(q, e_2)$, where $q_1 \neq q_2$ then in the theory $\mathcal{T}$ the expression $e_1 \wedge e_2$ is unsatisfiable.*

Return to our examples Ex. 1, Ex. 2, and Ex. 3 we illustrate in Fig. 2a,Fig. 2b, and Fig. 2c respectively how $\mathcal{AMT}$ can be used to formally specify safety and liveness properties. The notation is explained in Fig.2b.

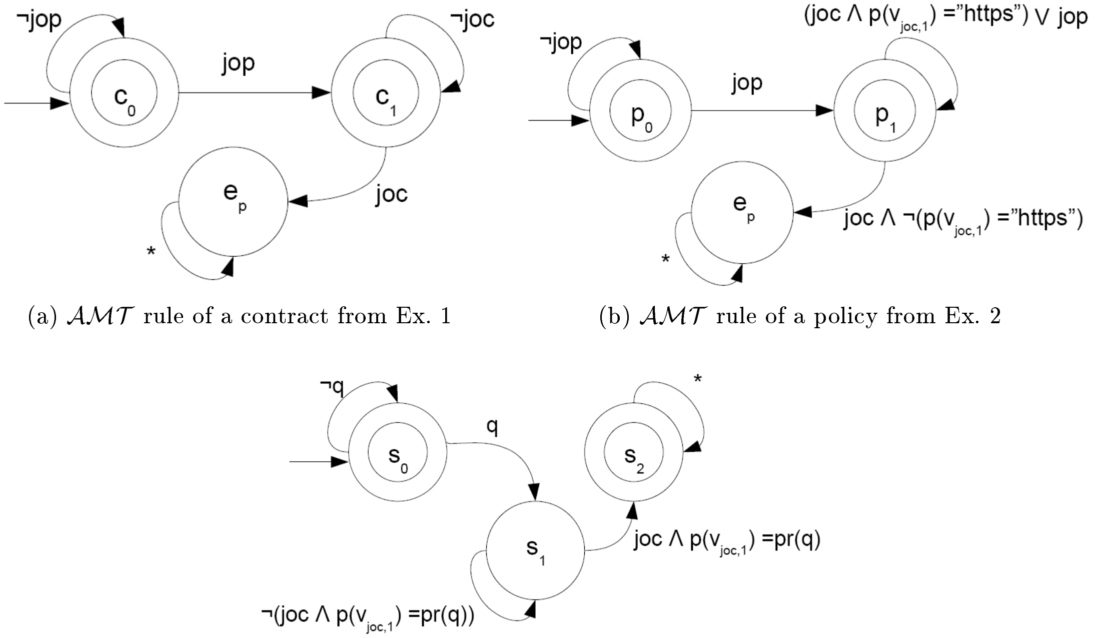
We define $\mathcal{AMT}$ operations for intersection and complementation requiring that the theory under consideration is closed under intersection, and complementation (union is similar to the standard one). However, in this paper we will only define complementation and use on-the-fly approach for we are interested in on-device model checking.

$\mathcal{AMT}$ Complementation. There are several possibilities for BA complementation [5, 19]. We consider here only the *complementation of deterministic $\mathcal{AMT}$*, for all security policies in our application domain are naturally deterministic (as the platform owner should have a clear idea on what to allow or disallow).

Definition 4.4 ($\mathcal{AMT}$ Complementation) *Given an $\mathcal{AMT}$ $A_{\mathcal{T}} = \langle E, S, q_0, \Delta_{\mathcal{T}}, F\rangle$ the complement automaton $A^c_{\mathcal{T}} = \langle E, S^c, q^c_0, \Delta^c_{\mathcal{T}}, F^c\rangle$ is:*

1. *$S^c = S \times \{0\} \cup (S - F) \times \{1\}, \qquad q^c_0 = (q_0, 0), \qquad F^c = (S - F) \times \{1\}$,*
2. *and for every $q \in S$ and $e \in E$*

$$\begin{array}{rcl} \Delta^c_{\mathcal{T}}((q,0),e) & = & \begin{cases} \{(\Delta_{\mathcal{T}}(q,e),0)\} & \Delta_{\mathcal{T}}(q,e) \in F \\ \{(\Delta_{\mathcal{T}}(q,e),0),(\Delta_{\mathcal{T}}(q,e),1)\}, & \Delta_{\mathcal{T}}(q,e) \notin F \end{cases} \\ \Delta^c_{\mathcal{T}}((q,1),e) & = & \{(\Delta_{\mathcal{T}}(q,e),1)\},\ \textit{if } \Delta_{\mathcal{T}}(q,e) \notin F \end{array}$$


¬jop
jop
¬joc
c0
c1
ep
joc
*
¬jop
(joc ∧ p(vjoc,1) ="https") ∨ jop
jop
p0
p1
ep
joc ∧ ¬(p(vjoc,1) ="https")
*
¬q
q
*
s0
s2
s1
joc ∧ p(vjoc,1) =pr(q)
¬(joc ∧ p(vjoc,1) =pr(q))

(a) $\mathcal{AMT}$ rule of a contract from Ex. 1

(b) $\mathcal{AMT}$ rule of a policy from Ex. 2

(c) $\mathcal{AMT}$ rule of liveness from Ex. 3

Figure 2: $\mathcal{AMT}$ Examples

Proposition 4.1 *Let $A_{\mathcal{T}}$ be an $\mathcal{AMT}$ over a set of α. Then there is a (possibly nondeterministic) $\mathcal{AMT}$ $A^c_{\mathcal{T}}$ such that $L_\omega(A^c_{\mathcal{T}}) = \alpha^\omega - L_\omega(A_{\mathcal{T}})$.*

This is the general construction, but if we only consider the fact that we complement a policy automaton for safety with a special property, namely it has all (but one) accepting states. As we complement the states, we would have only one accepting state which is $(err, 1)$. However, we can collapse it with a non accepting state $(err, 0)$. Hence, we do not need to mark states with 0 and 1 and the only accepting state is (err). Furthermore the complementation transitions remain the same as the original transitions.

Definition 4.5 ($\mathcal{AMT}$ Intersection) *Given a nondeterministic $\mathcal{AMT}$ $\left\langle E, S^{\mathbf{C}}, q_0^{\mathbf{C}}, \Delta_{\mathcal{T}}^{\mathbf{C}}, F^{\mathbf{C}} \right\rangle$ and a nondeterministic $\mathcal{AMT}$ $\left\langle E, S^{\overline{\mathbf{P}}}, q_0^{\overline{\mathbf{P}}}, \Delta_{\mathcal{T}}^{\overline{\mathbf{P}}}, F^{\overline{\mathbf{P}}} \right\rangle$, the intersection $\mathcal{AMT}$ automaton $\langle E, S, q_0, \Delta_{\mathcal{T}}, F\rangle$ is:*

1. $S = S^{\mathbf{C}} \times S^{\overline{\mathbf{P}}} \times \{1, 2\}, \qquad q_0 = (q_0^{\mathbf{C}}, q_0^{\overline{\mathbf{P}}}, 1), \qquad F = F^{\mathbf{C}} \times S^{\overline{\mathbf{P}}} \times \{1\},$
2. $\Delta_{\mathcal{T}} = \{((q^{\mathbf{C}}, q^{\overline{\mathbf{P}}}, x), a, (q'^{\mathbf{C}}, q'^{\overline{\mathbf{P}}}, y)) | (q^{\mathbf{C}}, a, q'^{\mathbf{C}}) \in \Delta_{\mathcal{T}}^{\mathbf{C}} \wedge (q^{\overline{\mathbf{P}}}, a, q'^{\overline{\mathbf{P}}}) \in \Delta_{\mathcal{T}}^{\overline{\mathbf{P}}} \wedge \mathit{condition}\}$

We use marker 1 when an accepting state of first component has been visited and 2 when accepting states from both components have been visited. The condition is defined by the following rules: if $q^{\mathbf{C}} \in F^{\mathbf{C}} \wedge x = 1$ then $y = 2$, if $q^{\overline{\mathbf{P}}} \in F^{\overline{\mathbf{P}}} \wedge x = 2$ then $y = 1$, otherwise $x = y$

Proposition 4.2 *If theory $\mathcal{T}$ is decidable and let $A^{\mathbf{C}}_{\mathcal{T}}$, $A^{\overline{\mathbf{P}}}_{\mathcal{T}}$ be $\mathcal{AMT}$. Then there is an $\mathcal{AMT}$ $A_{\mathcal{T}}$ such that $L_\omega(A_{\mathcal{T}}) = L_\omega(A^{\mathbf{C}}_{\mathcal{T}}) \cap L_\omega(A^{\overline{\mathbf{P}}}_{\mathcal{T}})$.*

Algorithm 1 MatchSpec Function

```
Require: $L^C = \langle (\mathrm{ID}^C_1,\ \mathrm{Spec}^C_1), \ldots, (\mathrm{ID}^C_n,\ \mathrm{Spec}^C_n) \rangle$, $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$
Ensure: 1 if $L^C$ matches $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$, 0 otherwise
if $\exists (\mathrm{ID}^C,\ \mathrm{Spec}^C) \in L^C \wedge \mathrm{ID}^C = \mathrm{ID}^P$ then
    if $\mathrm{HASH}(\mathrm{Spec}^C) = \mathrm{HASH}(\mathrm{Spec}^P)$ then
        return(1)
    else if $\mathrm{Spec}^C \approx \mathrm{Spec}^P$ then
        return(1)
    else if $\mathrm{Spec}^C \sqsubseteq \mathrm{Spec}^P$ then
        return(1)
    else {Restriction: if same ID then same specification must match}
        return(0)
else
    MatchSpec$\left(\left(*,\ \oplus_{(\mathrm{ID}^C,\ \mathrm{Spec}^C) \in L^C}\right), (*,\ \mathrm{Spec}^P)\right)$
```

5 On-the-fly State Model Checking with Decision Procedure

The outer algorithm for matching the midlet's claim with the security policy is taken from [7] and in particular the MatchSpec function (Alg. 1). It checks the match between a set of pairs $\mathcal{L}^C = \langle (\mathrm{ID}^C_1,\ \mathrm{Spec}^C_1), \ldots, (\mathrm{ID}^C_n,\ \mathrm{Spec}^C_n) \rangle$ and $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$ representing respectively the rules of the contract and a rule of the policy to be matched and returns 1 if there exists a pair $(\mathrm{ID}^C,\ \mathrm{Spec}^C)$ in L^C that matches $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$ or the combination of all the specifications in L^C matches $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$, otherwise 0.

Constructing the product automaton explicitly is not practical for mobile devices. At first this can lead into an automaton too large for the mobile limited memory footprint. Second, in order to construct a product automata we need software libraries for the explicit manipulation and optimizations of symbolic states, which are computationally heavy and not available on mobile phones. Furthermore we can exploit the explicit structure of the contract/policy as a number of separate requirements. Hence, we use on-the-fly emptiness test (constructing product automaton while searching the automata). On-the-fly emptiness test can be lifted from the traditional algorithm by Coucubertis et al. [6] while modification of this algorithm from Holzmann et al's [15] considered as state-of-the-art (used in Spin [14]). Gastin et al [11] proposed two modifications to [6]: one to find counterexamples faster, and another to find the minimal counterexample.

In $\mathcal{AMT}$ we are interested in finding counterexamples faster and we combine algorithm based on Nested DFS [22] with decision procedure for SMT. The algorithm takes as input the midlet's claim and the mobile platform's policy as $\mathcal{AMT}$ and then starts a depth first search procedure **check_safety** $(q_0^{\mathbf{C}}, q_0^{\overline{\mathbf{P}}}, 1)$ over the initial state $(q_0^{\mathbf{C}}, q_0^{\overline{\mathbf{P}}}, 1)$. When a suspect state which is an accepting state in $\mathcal{AMT}$ is reached we have two cases. First, when a suspect state contains an error state of complemented policy $(err^{\overline{\mathbf{P}}})$ then we report a security policy violation without further ado.[3] Second, when a suspect state which is an accepting state in $\mathcal{AMT}$ does not contain an error state of complemented policy $(S^{\overline{\mathbf{P}}} \backslash \{err^{\overline{\mathbf{P}}}\})$ we start a new depth first searches (Alg3)from the suspect state to determine whether it is in a cycle, in other words it is reachable from itself. If it is, then we report availability violation.

[3] The Error state is a convenient mathematical tool but the trust assumption of the matching algorithm is that the code obeys the contract and therefore it should never reach the error state where everything can happen.

Algorithm 2 **check_safety**$(q^{\mathbf{C}}, q^{\overline{\mathbf{P}}}, x)$ Procedure

```
Require: state $q^{\mathbf{C}}$, state $q^{\overline{\mathbf{P}}}$, marker $x$;
$map(q^{\mathbf{C}}, q^{\overline{\mathbf{P}}}, x) := \texttt{in\_current\_path}$;
for all $((q^{\mathbf{C}}, a^{\mathbf{C}}, q'^{\mathbf{C}}) \in \Delta^{\mathbf{C}}_{\mathcal{T}})$ do
  for all $((q^{\overline{\mathbf{P}}}, a^{\overline{\mathbf{P}}}, q'^{\overline{\mathbf{P}}}) \in \Delta^{\overline{\mathbf{P}}}_{\mathcal{T}})$ do
    if $(DecisionProcedure(a^{\mathbf{C}} \wedge a^{\overline{\mathbf{P}}}) = SAT)$ then
      $y := condition(q^{\mathbf{C}}, q^{\overline{\mathbf{P}}}, x, F^{\mathbf{C}}, F^{\overline{\mathbf{P}}})$
      if $(map(q'^{\mathbf{C}}, q'^{\overline{\mathbf{P}}}, y) = \texttt{in\_current\_path} \wedge ((\mathtt{q}^{\mathbf{C}} \in \mathtt{F}^{\mathbf{C}} \wedge \mathtt{q}^{\overline{\mathbf{P}}} = \mathtt{err}^{\overline{\mathbf{P}}} \wedge \mathtt{x} = \mathtt{1}) \vee (\mathtt{q'}^{\mathbf{C}} \in \mathtt{F}^{\mathbf{C}} \wedge \mathtt{q'}^{\overline{\mathbf{P}}} = \mathtt{err}^{\overline{\mathbf{P}}} \wedge \mathtt{y} = \mathtt{1})))$ then
        report policy violation;
      else if $(map(q'^{\mathbf{C}}, q'^{\overline{\mathbf{P}}}, y) = \texttt{in\_current\_path} \wedge ((\mathtt{q}^{\mathbf{C}} \in \mathtt{F}^{\mathbf{C}} \wedge \mathtt{q}^{\overline{\mathbf{P}}} \in (\mathtt{S}^{\overline{\mathbf{P}}} \backslash \{\mathtt{err}^{\overline{\mathbf{P}}}\}) \wedge \mathtt{x} = \mathtt{1}) \vee (\mathtt{q'}^{\mathbf{C}} \in \mathtt{F}^{\mathbf{C}} \wedge \mathtt{q'}^{\overline{\mathbf{P}}} \in (\mathtt{S}^{\overline{\mathbf{P}}} \backslash \{\mathtt{err}^{\overline{\mathbf{P}}}\}) \wedge \mathtt{y} = \mathtt{1})))$ then
        report availability violation;
      else if $(map(q'^{\mathbf{C}}, q'^{\overline{\mathbf{P}}}, y) = \texttt{safe})$ then
        check_safety$(q'^{\mathbf{C}}, q'^{\overline{\mathbf{P}}}, y)$;
if $(q^{\mathbf{C}} \in F^{\mathbf{C}} \wedge q^{\overline{\mathbf{P}}} \in S^{\overline{\mathbf{P}}} \wedge x = 1)$ then
  check_availability$(q^{\mathbf{C}}, q^{\overline{\mathbf{P}}}, x)$;
  $map(q^{\mathbf{C}}, q^{\overline{\mathbf{P}}}, x) := \texttt{availability\_checked}$;
else
  $map(q^{\mathbf{C}}, q^{\overline{\mathbf{P}}}, x) := \texttt{safety\_checked}$;
```

Algorithm 3 **check_availability**$(q^{\mathbf{C}}, q^{\overline{\mathbf{P}}}, x)$ Procedure

```
Require: state $q^{\mathbf{C}}$, state $q^{\overline{\mathbf{P}}}$, marker $x$;
for all $((q^{\mathbf{C}}, a^{\mathbf{C}}, q'^{\mathbf{C}}) \in \Delta^{\mathbf{C}}_{\mathcal{T}})$ do
  for all $((q^{\overline{\mathbf{P}}}, a^{\overline{\mathbf{P}}}, q'^{\overline{\mathbf{P}}}) \in \Delta^{\overline{\mathbf{P}}}_{\mathcal{T}})$ do
    if $(DecisionProcedure(a^{\mathbf{C}} \wedge a^{\overline{\mathbf{P}}}) = SAT)$ then
      $y := condition(q^{\mathbf{C}}, q^{\overline{\mathbf{P}}}, x, F^{\mathbf{C}}, F^{\overline{\mathbf{P}}})$
      if $(map(q'^{\mathbf{C}}, q'^{\overline{\mathbf{P}}}, y) = \texttt{in\_current\_path})$ then
        if $(q'^{\overline{\mathbf{P}}} = err^{\overline{\mathbf{P}}})$ then
          report policy violation;
        else
          report availability violation;
      else if $(map(q'^{\mathbf{C}}, q'^{\overline{\mathbf{P}}}, y) = \texttt{safety\_checked})$ then
        $map(q'^{\mathbf{C}}, q'^{\overline{\mathbf{P}}}, y) := \texttt{availability\_checked}$
        check_availability$(q'^{\mathbf{C}}, q'^{\overline{\mathbf{P}}}, y)$;
```

Function **condition**(s, t, x, F_1, F_2) returns 2 if$(s \in F_1 \wedge x = 1)$, returns 1 if $(t \in F_2 \wedge x = 2$ $)$, otherwise it returns x.

Remark 5.1 *Our algorithm is tailored for particular $\mathcal{AMT}$ for security contract-policy matching and exploits the particular form of the policy. A generic on-the-fly algorithm for the BA emptiness test of an $\mathcal{AMT}$ is obtained by removing all specialized tests and reporting only availability violation (corresponds to a non-empty automaton).*

We are now in the position to state our main results:

Theorem 5.1 *Let the theory $\mathcal{T}$ be decidable with an oracle for the SMT problem in the complexity class $\mathcal{C}$ then:*

1. *The non-emptiness problem for $\mathcal{AMTT}$ is decidable in $LIN-TIME^{\mathcal{C}}$.*
2. *The non-emptiness problem for $\mathcal{AMTT}$ is $NLOG-SPACE^{\mathcal{C}}$.*

6 Conclusion and Related Work

Model-carrying code [23] and security-by-contract [7] proposed to augment mobile code with a claim on its security behavior that can be matched against a mobile platform policy before code downloading. We showed that it is possible to define very expressive (essentially infinite) policies that can capture realistic scenarios while keeping the task of matching computationally tractable. The key idea is the concept of *Automata Modulo Theory* ($\mathcal{AMT}$). $\mathcal{AMT}$is an extension of Büchi Automata (BA), suitable for formalizing systems with finitely many states but infinitely many transitions.

Infinite numbers of transitions in security policies by labeling each transition with a computable predicate instead of an atomic symbol has been studied in [20]. Security automata were later implemented in several systems for example PoET/PSLang toolkit [10], which can enforce security policies whose transitions pattern-match event symbols using regular expressions. Edit automata [2] are another model for achieving this. Edit automata extend security automata to model the transforming effects of in-lined reference monitors. This was implemented in the Polymer system [3] to dynamically compose security automata. However, all mentioned approaches focus on the relations between code and security claims on the code (which we call contract), while $\mathcal{AMT}$ focuses between the security claims of the code and the platform desired security behavior. Most recently, the Mobile system [12] implements a linear decision algorithm that verifies that annotated .NET binaries satisfy a class of policies that includes security automata and edit automata. This work fits into step three in our lifecycle while $\mathcal{AMT}$ falls into steps one and five.

$\mathcal{AMT}$ makes it possible to match the mobile's policy and the midlet's contract by mapping the problem into a variant of on-the-fly product and emptiness test from automata theory, without symbolic manipulation procedures of zones and regions nor finite representation of equivalence classes. The tractability limit is essentially the complexity of the satisfiability procedure for the theories, called as subroutines, where most practical policies require only polynomial time decision procedures.

A known problem with security automata and infinity yet to be addressed is the encoding of policies such as "we must allow certain strings that we have seen in the past". If the set of strings is unbounded, then it is difficult (if not impossible) to encode it with finite states. Another interesting problem for future work is a scenario when the claimed security contract is missing (as it is the case for current MIDP applications). In that case, based on the platform security policy, the "claimed" security contract could be inferred by static analysis as an approximation automaton. If such an approximation is matched, then monitoring the code becomes unnecessary. The feasibility of this approach depends on the cost of inferring approximation automata on-the-fly.

References

[1] J. Bacon. Toward pervasive computing. *IEEE Perv.*, 1(2):84, 2002.

[2] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security*, Copenhagen, Denmark, July 2002.

[3] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM Press.

[4] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In K. Etessami and S. Rajamani, editors, *Proc. of CAV'05*, volume 3576 of *LNCS*, pages 335–349. Springer-Verlag, 2005.

[5] J. Büchi. On a decision method in restricted second-order arithmetic. In E. N. et al., editor, *Int. Congress on Logic, Methodology and Philosophy of Science*, pages 1–11, California, 1962. Stanford University Press.

[6] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Form. Methods Syst. Des.*, 1(2-3):275–288, 1992.

[7] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *Proc. of EuroPKI'07*, 2007.

[8] N. Dragoni, F. Massacci, C. Schaefer, T. Walter, and E. Vetillard. A security-by-contracts architecture for pervasive services. In *3rd International Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing*. IEEE Computer Society Press, 2007.

[9] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. Technical report 2003-1916, Department of Computer Science, Cornell University, 2003.

[10] U. Erlingsson and F. Schneider. Irm enforcement of java stack inspection. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 246, Washington, DC, USA, 2000. IEEE Computer Society.

[11] P. Gastin, B. Moro, and M. Zeitoun. Minimization of counterexamples in SPIN. In *Model Checking Software: 11th International SPIN Workshop (SPIN'04)*, volume 2989 of *LNCS*, pages 92–108. Springer, 2004.

[12] K. Hamlen, G. Morrisett, and F. Schneider. Certified in-lined reference monitoring on .net. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 7–16, New York, NY, USA, 2006. ACM Press.

[13] T. Henzinger, R. Majumdar, and J. Raskin. A classification of symbolic transition systems. *ACM Trans. Comput. Logic*, 6(1):1–32, 2005.

[14] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004.

[15] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.

[16] G. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[17] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI'96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 229–243, New York, NY, USA, 1996. ACM Press.

[18] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *J. of the ACM*, 53(6):937–977, Nov 2006.

[19] S. Safra. On the Complexity of omega-Automata. In *IEEE Symp. on Found. Comp. Science (FOCS'88)*, pages 319–327, White Plains, New York, USA, 1988. IEEE Press.

[20] F. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[21] F. Schneider, J. Morrisett, and R. Harper. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 86–101, London, UK, 2001. Springer-Verlag.

[22] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. Technical Report 2004/06, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, November 2004.

[23] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP-03)*, pages 15–28. ACM Press, 2003.

[24] V. N. Venkatakrishnan, R. Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *Proc. of NSPW '02*, 2002.