Testing Decision Procedures for Security-by-Contract *

Nataliia Bielova   Ida Siahaan
Universitá di Trento, Italy
{bielova,siahaan}@disi.unitn.it

Abstract

The traditional realm of formal methods is the off-line verification of formal properties of hardware and software. In this paper we report a different approach that uses formal methods (namely the integration of automata modulo theory with decision procedures) on-the-fly, at the time an application is downloaded on a mobile application such as PDA or a smart phone.

The idea behind security-by-contract is that a mobile application comes equipped with a signed contract describing the security relevant behavior of the application and such contract should be matched against the mobile platform policy. Both are specified as special kinds of automata and the operation is just an on-the-fly emptiness test over two automata modulo theories where edges are not just finite states of labels but rather expressions that can capture infinite transitions such as "connect only to urls starting with https://".

The paper describe the prototype implementation, its integration with a state of the art decision solver (based on MathSAT and NuSMV) and the preliminary experiments that we have done for contract-policy matching.

Keywords Formal Specification, Security Policies, Mobile Code

1 Introduction and Motivations

The paradigm of pervasive services [3] envisions a nomadic user seamlessly and constantly receiving services from other devices and sensors embedded in the environment. Beside this web-service-like model, a new model is emerging based on the notion of *pervasive client downloads* [14]: users download new (and likely untrusted) applications on their mobile in order to exploit the computational characteristic of the device.

A tourist landing in a large city can download at the airport a navigation application that can guide her to shopping centers or touristic sights. The application can query internet sites or bluetooth services to find the optimal routes or discover local services. Living Search by Microsoft and Navitime by DoCoMo [2] are primitive examples of these future applications. Peer-to-peer and Web 2.0 collaborative applications share the same features: Channel4 in the UK allows people to download video on demand if they also download a P2P servent.

*Research partly supported by the Projects EU-FP6-IST-STREP-S3MS, EU-FP6-IP-SENSORIA, and EU-FP7-IP-MASTER

Unfortunately, this business model is not supported by the current security architecture of Java [17] and .NET [21]:

- mobile code runs only if its origin is trusted (i.e. digitally signed by a trusted party);
- a pervasive download will likely be from small companies which cannot afford to obtain a mobile operator's certification and thus will not run as trusted code;
- then this application should be sandboxed, its interaction with the environment and the device's data should be limited;
- yet we made this pervasive download precisely to have lots of (controlled) interaction with the pervasive environment.

As it is now this is both a business opportunity but also a big security threat: Channel 4 naive users with a pay-as-you-go subscription to internet found out at their own expenses the "surprising" effect of hosting a P2P application for video on demand.

We need a better security model where the mobile code should be run only if it satisfies a user-defined policy. This is precisely the setting where we can use *formal methods on-the-fly*: before downloading the application we just verify that it complies with the user security policies.

Unfortunately, in the general case this is equivalent to arbitrary software verification which is not practical for pervasive downloads (remember this has to be done on your smart phone while you wait). However, the idea behind model-carrying code [28] and security-by-contract [13] is that code should come accompanied with a "digest" (a security model or a security contract) that represents its essential security behavior.

The question raised is how we know that the security claims are actually true on the code. One possible solution is to use proof carrying code [23] or trust relations and digital signatures. The PCC approach enables safe execution of code from untrusted sources. PCC is based on assumption that the code producer should know all the security policies that are of interest to consumers since the producer sends the safety proof together with the mobile code. This assumption can be impractical due to various security policies among different consumers. On the other hand, if we use only trust relationship, i.e. digital signatures on mobile code, then we can only reject or accept the signature and no semantics attached to the signature. The security-by-contract proposed in [13] provides semantics to a digital signature, which was not presented beforehand. So that, when binding together the code and the contract the signer takes liability for the security claims ([31] describes mobile devices security architecture that supports integration of proof-carrying code, static verification and run-time monitoring). Then one only needs to match the contract against the platform security policies. However, whenever consumer does not trust the contract provided by the code producer then the overall architecture can take care that the code actually complies with the contract by run time monitoring(see [14] describes security by contract architecture).

The next question is *which is the best formal representation of such contract and policy*. Model carrying code papers [28] suggested finite automata. Unfortunately, finite state and even Büchi Automata are too simple to express any practical policy: already the rule "only allows connections to urls starting with `https://`" would generate an automaton with infinite transitions when instantiating urls. Languages for security-by-contract policies [1] are even more expressive.

The formal model considered for capturing contracts and policies is based on the novel concept of *Automata Modulo Theory* ($\mathcal{AMT}$). $\mathcal{AMT}$ has been introduced

in [22], which extends Büchi Automata (BA) by labeling transitions with expressions belong to decidable theories. It is suitable for formalizing systems with finitely many states but infinitely many transitions by leveraging on the power of satisfiability-modulo-theory (SMT) decision procedures. In this way we can represent the task of matching the contract with the policy as language containment problem between two automata. However, while [22] provides the theoretical framework, namely the on-the-fly matching algorithm and the complexity results of the operation, the actual implementation of the algorithm and the integration with a state-of-the-art theory solver is still left open.

1.1 The Contribution of this Paper

We discuss the overall implementation architecture and the integration issues with a state of the art decision procedure solver NuSMV [12] integrated with its MathSAT libraries [9]. 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.

To this extent we have decided to implement language inclusion as emptiness test as an on-the-fly procedure a-la-SPIN with oracle calls to the decision procedures available in NuSMV. Therefore our design decision $\mathcal{AMT}$ makes reasoning about infinite transitions systems with finite states possible without symbolic manipulation procedures of zones and regions or finite representation by equivalence classes whose memory intensive characteristic is not suitable for our application.

The second contribution is a detailed performance analysis of the integration design alternatives regarding the construction of expressions, the initialization of solver, and the caching of temporary results by considering both running time and internal metrics of various available options.

We first introduce the concept of security-by-contract (§2) and the notion of Automata Modulo theory (§3). After description of the on-the-fly algorithm for contract policy matching we introduce the architecture of our prototype (§4) and the design decisions needed for evaluation (§5). Finally we report our experimental findings (§6) and conclude with a brief discussion of related work (§7)

2 Security by Contract

In a security-by-contract paradigm [13] a *contract* accompanying an application is just a set of rules describing the security behavior of the mobile application with its host platform. We use the term *policy* to denote the set of rules that the host platform would like to be respected. As we have anticipated, such rules can then be mapped to restrictions on API usage by the application corresponding to variants of automata.

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 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 security claims are sealed together with a digital signature and shipped for deployment.

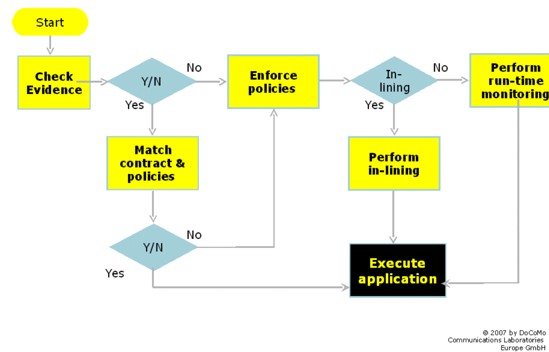


Figure 1: SxC Workflow

Table 1: End Users' Distilled Security Requirements

USE of Costly functionalities Any invocation of paid services, such as sending text messages, using GPRS or wireless connections, must be controllable by the user.

NETwork connectivity Any external connections made by the application can be controlled.

PRIvate information management It is necessary to control what data is accessed by the application such as local files, PIM items or contacts from Contact List.

INTeraction with other applets This requirement makes necessary to control means of inter-process communication, in particular sockets and memory-mapped files.

Power consumption This requirement is two-fold: it makes necessary to control the invocation of power-consuming functionality, such as WiFi connections, and to control the battery level in course of running the application. This can be mapped into the NET and USE categories.

EXTended functionality If the device is equipped with some advanced functionality, such as camera or GPS receiver, its use is likely to be controlled by policies.

At deployment time the target platform will follow the workflow that we have sketched in Fig.1 (see also [31]). At first it checks that the evidence is correct. Such evidence could be a trusted signature as in standard mobile applications. Alternative evidence could be a proof that the code satisfies the contract and then one could use PCC techniques to check it [23].

As we have evidence that the contract is trustworthy the platform will check that the claimed policy is actually compliant with the policy that our platform would like to be enforced. If this is the case, then the application can be run without further ado.

Contracts and policies may vary significantly but a number of analyses of security requirements for mobile and ubiquitous applications [20, 29, 33] have shown that we can essentially distill them in few categories (Table 1). Such requirements can then be mapped into concrete behavioral constraints on usage of APIs. Here we discuss informally the syntax and refer to [1] for details.

The contract/policy is written in ConSpec language [1]. It was suggested for security-by-contract application and its semantics in terms of automata modulo theory [22]. Contract/policy is just a *list of disjoint rules* for connections, for the Personal Identification Module, for file access and so on. The main part of rule includes a *list of event clauses*, clause is a method provided by an API. The clause specification is followed by a sequence of guard-update block pairs. The guard is a boolean expression, specifying a constraint on the method invocation. The update block is a list of actions that will be

executed in case of true guard.

The main part of a rule gives a rigorous and unambiguous definition of the behavior (semantics) of the rule. Several semantics can be used for this purpose, e.g. standard process algebras and security automata.

Example 1 *Alice is a mobile application developer. To assure her customers that the application does not alter any network configurations after Personal information management*[1] *(PIM) was opened the contract states that no connections can be made. The MIDlet cannot establish any connections, i.e. cannot open sockets, bluetooth or other connections.*

```
RULEID HIGH_LEVEL_CONNECTIONS
SCOPE Session
SECURITY STATE
 boolean opened = false;
AFTER javax.microedition.pim.PIM.openPIMList(PimListType pimListType,
       Mode mode)
PERFORM
  true -> { opened = true; }
BEFORE javax.microedition.io.Connector.open(string url)
PERFORM
  ! opened -> {skip; }
BEFORE javax.microedition.io.Connector.open(string url, int mode)
PERFORM
  ! opened -> {skip; }
BEFORE javax.microedition.io.Connector.open(string url, int mode,
       boolean timeouts)
PERFORM
  ! opened -> {skip; }
```

The BEFORE $method$ step corresponds to rules that are invoked before the executions of the $method$. Then the guards are checked after the PERFORM keyword. The first guard that evaluates to true is executed and the actions between brackets are executed.

Example 2 *Bob is a user of mobile device. He has a number of important mobile phone numbers from his business partners so he would like that if his agenda is opened all subsequent communications are secured. The policy for the host platform describes such behaviour: after PIM was opened the MIDlet can only establish secure connections, i.e. HTTPS connections.*

In the sequel, we consider a number of examples for experiments that provide a good coverage of the requirements that we mentioned afore (Table2). For instance, the ex.1 is a pimNoConn example and ex.2 is pimSecConn, both of them cover the USE, PRI and NET user requirements. We append to each problem name the _contract or _policy suffix denoting whether the rule is used to specify a contract or a policy.

3 Automata Modulo Theory ($\mathcal{AMT}$)

The formal tool used to represent policies and contracts is the concept of $\mathcal{AMT}$. The theory of $\mathcal{AMT}$ [22] is a combination of the theory of BA with the SMT problem, namely the satisfiability of first-order formulas modulo background theories. The intuition of $\mathcal{AMT}$ is that we represent a security policy as BA automaton where edges are not labeled by atomic actions but rather by expressions in a suitable theory.

[1] The PIM system on the phone has the ability to manage appointment books, contact directories, etc. in electronic form.

Table 2: Benchmark Contract and Policies

Example ID	Natural Language description	Coverage
httpHttps	The application only uses high-level network connections.	NET
https	The application only uses HTTPS network connections.	NET, PRI
maxKB512	The data received by application is bounded by 512Kb	USE, NET
maxKB1024	The data received by application is bounded by 1024Kb	USE, NET
noPushRegistry	The application does not use the push registry mechanism	USE
oneConnPushRegistry	Only one connection registered to the Push registry at a time	USE, NET
notCreateRSt	The policy allows to open record stores, but it is not allowed to create new record stores.	INT
notCreateSharedRS	The application does not create shared record stores.	INT, PRI
noSMS	No messages are sent by the application	USE
100SMS	Maximum 100 text messages can be sent by the application	USE
pimNoConn	After PIM was opened no connections are allowed	USE, PRI, NET
pimSecConn	After PIM was accessed only secure connections (HTTPS) can be opened	USE, PRI, NET

While traditional security automata are usually safety automata [15, 7] we prefer to use BA because besides safety properties, there are also some liveness properties which have to be verified. For example, "The application uses all the permissions it requests".

Some theories of interest are difference logic $\mathcal{DL}$ equality and uninterpreted functions $\mathcal{EUF}$ $\mathcal{LA}(\mathbb{Q})$ and $\mathcal{LA}(\mathbb{Z})$. As in [8] we are particularly interested in the combination of two or more simpler theories. While this is a not complete list, our only requirement for a theory $\mathcal{T}$ is that the $\mathcal{T}$-satisfiability of conjunctions of ground literals is decidable by a $\mathcal{T}$-solver [24].

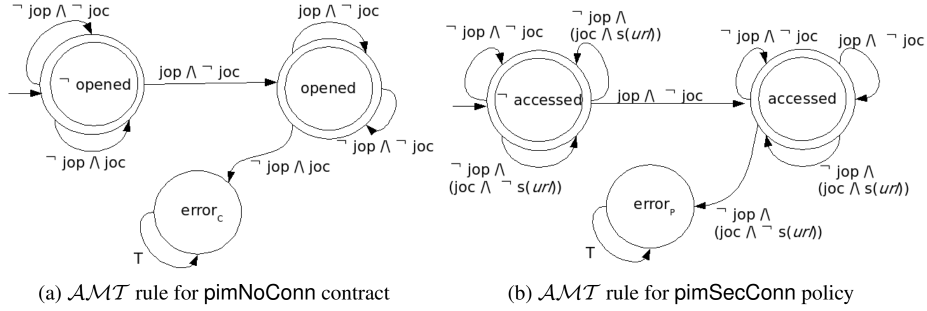
Definition 3.1 (Automaton Modulo Theory) *A tuple $A = \langle E, S, q_0, \Delta_{\mathcal{T}}, F\rangle$ where E is a set of formulae in the language of the theory $\mathcal{T}$, S is a finite set of states, $q_0 \in S$ is the initial state, $\Delta_{\mathcal{T}} : S \times E \to 2^S$ is labeled transition function, and $F \subseteq S$ is a set of accepting states.*

The runs of the system are the traces of actual values of invoked APIs, represented by assignments.

Definition 3.2 ($\mathcal{AMT}$ concrete 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: $s_0 = q_0$ and there exists expressions $e_i \in E$ where $s_{i+1} \in \Delta_{\mathcal{T}}(s_i, e_i)$ and $(\mathcal{A}, \alpha_i) \models e_i$ is satisfiable for all $i \in [0 \ldots |w|]$ (resp. $i \in N$). The trace associated with γ is sequence of assignments $w = \langle \alpha_0, \alpha_1, \alpha_2, \ldots\rangle$. 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.*

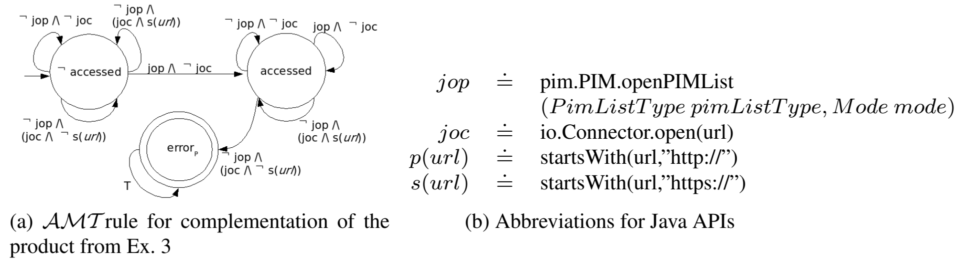
A trace is a *word* in the language of $\mathcal{AMT}$. The set α^* denotes the set of finite words over α while the set α^ω is the set of infinite words over α. The language of $\mathcal{AMT}$ is a set of words. The transition function of $A_{\mathcal{T}}$ may have many possible transitions for each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

Definition 3.3 (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.*



(a) $\mathcal{AMT}$ rule for pimNoConn contract (b) $\mathcal{AMT}$ rule for pimSecConn policy

Figure 2: $\mathcal{AMT}$ rules for the contract and policy of Ex. 3



jop	$\doteq$	pim.PIM.openPIMList $(PimListType\ pimListType, Mode\ mode)$
joc	$\doteq$	io.Connector.open(url)
$p(url)$	$\doteq$	startsWith(url,"http://")
$s(url)$	$\doteq$	startsWith(url,"https://")

(a) $\mathcal{AMT}$rule for complementation of the product from Ex. 3

(b) Abbreviations for Java APIs

We describe expressions with function names from Java VM, since we do not consider useful to invent our own names for API calls we use the *javax.microedition* APIs for notation.

Example 3 *Let us return to Bob's policy again in Fig.2b which represents an automaton for the contract. Starting from state* $\neg$`accessed`*, we stay in this state while PIM is not accessed (*$\neg jop$*). As PIM is accessed we move to state* `accessed` *and we stay in this state only if the subsequent connections are secured i.e. "https://" or we keep accessing PIM (*jop*). We enter state* $\texttt{error}_P$ *if we start an unsecured connection e.g. url starts with "http://" or "sms://" etc.*

Expressions. In order to deal with arbitrary arithmetical conditions and protocols the expressions allowed on edges follows the following syntax:

```
bool  ::= bool "&" bool | bool "|" bool | "!" bool | "(" bool ")" | str_bool | basic
basic ::= constant | var | basic = basic | basic != basic
          | basic < basic | basic > basic | basic <= basic | basic >= basic
          | basic + basic | basic - basic | basic * basic  | basic / basic
str_bool ::=  "startsWith(" str "," str_const ")"
          | "equals(" str "," str ")"
          | "indexOf(" str "," char ")==-1"
```

Where *bool* denotes expression of boolean type (e.g. simple boolean expressions or boolean function on the string), *str_bool* denotes boolean function on the string, *basic* denotes the basic expression on booleans and integers, and *var* denotes a boolean, integer or string variable.

For the automata to be correctly defined we have to establish a background theory, for example for predicate `startsWith(str,str_const)`. We use `startsWith` for extracting protocol from a URL and assume that there are no two distinct `str_const` such that the predicate holds for the same `str`, for example "http://" and "https://" are good as string constants while "http" and "https" are bad as string constants. In addition, if

`arg2` and `arg2'` are two distinct strings, then the predicate `(startsWith(arg1,arg2) & startsWith(arg1,arg2'))` must yield false for all `arg1`. These constraints and the relation between `startsWith` and `indexOf` are axiomatized as follows:

1. `startsWith(url,"http://")` $\leftrightarrow$ `! startsWith(url,"https://")`
   ``http://'' and ``https://'' are correct protocols.

2. `startsWith(url,"http://")` $\rightarrow$ `! indexOf(url,':' ) == -1`
   the `url` starts with `"http://"` that contains the `':'` character.

The security behaviors provided by the contract and desired by the policy can be represented as automata where transitions correspond to invocation of APIs as suggested by Erlingsson [15, p.59] and Sekar et al. [28]. Then the operation of matching the midlet's claim with platform policy can be mapped into classical problems in automata theory.

One possible alternative is *language inclusion*: 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 execution traces of the midlet described by Aut^C is a subset of the acceptable traces for Aut^P. To check this property we can complement the automaton of the policy, thus obtaining the set of traces disallowed by the policy and check its intersection with the traces of the contract. If the intersection is not empty, any behavior in it corresponds to a security violation, pursued in [22].

The other alternative is the notion of *simulation*: we have a match when every APIs invoked by Aut^C can also be invoked by Aut^P. In other words, every behavior of Aut^C is also behavior of Aut^P. Simulation is usually a stronger notion than language inclusion as it requires that the policy allows the actions of the midlet's contract in a "step-by-step" fashion, whereas language inclusion looks at an execution trace as a whole.

In this paper we use the approach of language inclusion as in [22], 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 application) is a subset of the language accepted by Aut^P (i.e. the acceptable traces for the policy). Matching problem can be reduced to an emptiness test.

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. It means that we will have to complement the policy automaton.

Example 4 *Bob's policy is negated so that now a violating path is accepted by the automaton. In Fig.3a if the PIM was accessed then we could open the HTTP connection and this will lead us to the error state ($\mathtt{error}_P$). We could also open any other connections except for HTTPS connection.*

These operations require that the language of the theory under consideration is closed under intersection and complementation. In a nutshell, one simply uses the classical operation on automata but instead of checking edges for equality of labeling transitions we check them for satisfiability of the conjunction of the labeling expressions [22]. We consider only the *complementation of deterministic* $\mathcal{AMT}$, because in our application domain all security policies are naturally deterministic, as the platform owner should have a clear idea on what to allow or disallow. This constraint arises also due to BA complementation, the nondeterminic complementation is complicated and

exponentially blow-up in the state space [11]. Safra in [25] gives a better lower bound ($2^{O(n \log n)}$), however it is still exponential (see [32]).

In the particular example (Ex. 4) complementation is just done by switching accepting states and non-accepting states because we have a classical security automata (i.e. only concerned about safety properties [18]). This would be more complex if we had liveness properties in the original automaton i.e. the original policy had more non-accepting states beside the error state. While in classical security automaton the only non-accepting state is the error state.

At this stage we only need to decide which algorithm we use for emptiness testing. One of the key observations is that we will seldom need to crack a big nut (the whole of the policy against the whole of the contract) but rather repeatedly crack many small nuts (the "Access to PIM" rule in the policy against the corresponding one in the contract, the rules for network access and so on). Further, we have a good but limited memory footprint (a smart phone is the target platform) which rules out symbolic manipulation procedures by zones and regions of the whole state space. Another important observation is that whereas the policy is somehow pre-loaded in the device, and is unlikely to change frequently, the contract will only come on the fly together with the application (See [14, 31] for some descriptions of the whole run-time architecture)

Our decision was to integrate a truly on-the-fly Nested DFS [27] with decision procedure (DP) for SMT. The algorithm takes as input the application's contract and the mobile platform's policy as $\mathcal{AMT}$ and then starts a depth first search procedure over the initial pair of states. When a suspect pair of states is reached we have two cases. If one state is an error state of the complemented policy then we report a security policy violation without further ado. Otherwise we start a new depth first search from the suspect states to determine whether they are in a cycle, in other words they are reachable from themselves. In this latter case, we report an availability violation.

Note that our language inclusion approach differs from simulation approach. We can consider the corresponding concrete automaton which is constructed by replacing each transition labeled with an expression from the theory with the infinitely many transitions labeled by the corresponding satisfying assignments. Automata that are different at the theory level might have the same concrete representation. For example, take two automata modulo theory C and P, one with splitting edge ($joc\&protocol(url) =$ "$http''$ and $joc\&protocol(url) =$ "$https''$) and the other with OR edge ($joc\&(protocol(url) =$ "$http'' \vee protocol(url) =$ "$https''$)). Both have the same concrete model. Such equivalence is obvious because at the concrete level if the assignment α_{1i} is such that $(\mathcal{A}, \alpha_{1i}) \models joc\&protocol(url) =$ "$http''$ or $(\mathcal{A}, \alpha_{2i}) \models joc\&protocol(url) =$ "$https''$ then clearly $(\mathcal{A}, \alpha_i) \models joc\&(protocol(url) =$ "$http'' \vee protocol(url) =$ "$https''$). In other words, $\vee$ has the maximal model and thus in the transitions corresponding to the disjunction in the theory it is the union of all assignments in the concrete automaton.

This leads to $\mathcal{AMT}$ fair simulation is stronger than $\mathcal{AMT}$ language inclusion. For example if we have policy represented as P and contract represented as C, where both automata accept the same language but according to simulation $\models joc\&(protocol(url) =$ "$http'' \vee protocol(url) =$ "$https''$) $\rightarrow joc\&protocol(url) =$ "$http''$ does not hold, thus we do not have simulation. Technically this is a consequence of the maximal model for $\vee$.

The on-the-fly matching algorithm presented here is not a fairly standard automata inclusion, as we need to make call to decision procedure which is some theorem solver. Thus, when theory $\mathcal{T}$ is decidable with an oracle for the SMT problem in the complexity class $\mathcal{C}$ then: the non-emptiness problem for $\mathcal{AMTT}$ is decidable in

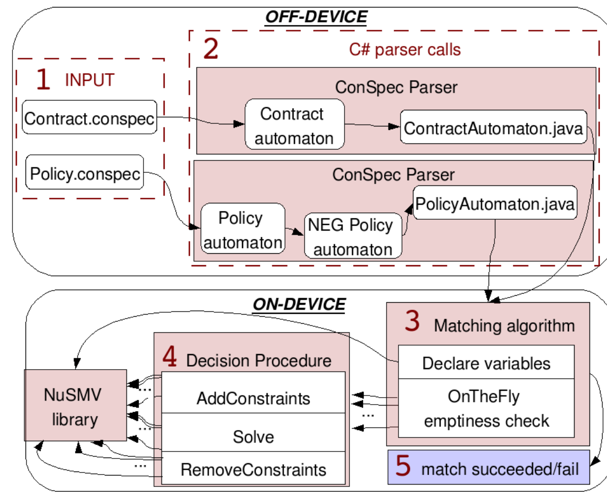


Figure 3: Contract-Policy Architecture

$LIN-TIME^{\mathcal{C}}$ and $NLOG-SPACE^{\mathcal{C}}$ [22].

Returning to Ex.3, the prototype translates the specifications into $\mathcal{AMT}$ (Fig.2) and runs the on-the-fly algorithm creating and visiting the product automaton. In the Bob and Alice case this means that visiting the two automata in parallel we should reach a state where `opened` (see Fig. 2a) and $\mathtt{error}_P$ is reached. When running the matching algorithm we find out that there is no cycle through any accepting state (the accepting state for contract and negated policy, i.e. accepting contract and violating policy at the same time), meaning contract matches policy.

4 The Architecture

In this section we describe the conceptual architecture of the prototype that implements the overall matching algorithm and supports integration with state of the art decision procedure solver NuSMV [12] integrated with its MathSAT libraries [9]. The main aim is to provide a concrete overview of how the prototype is implemented so that one can easily understand the possible options for integration with the solver. The contract-matching prototype takes as input a contract and a policy both specified in ConSpec and checks whether or not the contract matches the policy. A sketch of the prototype architecture is shown in Fig. 3.

Our first observation is that the policy has to be deployed on the device and it is unlikely to change frequently. The second observation is that, even if applications (and related contracts) will change frequently and dynamically, the binding between an application and its contract will be pretty static. If a digital signature or a proof carrying code is used, the contract has to be shipped with the application. This contract must be essentially included in the JAR file that represents the application and must be directly accessible to the virtual machine that is responsible for the matching and the enforcement of the security policy (see [31] for details).

In this paper we thoroughly describe a work made on a Java platform for a Desktop PC and give some experimental results on .NET implementation for a Mobile platform. The prototype is basically separated in 2 parts: on-device and off-device implementations. During off-device part execution, the contract and policy are transformed into a

suitable internal representation for the on-the-fly algorithm. The policy automaton is also complemented at this step of the execution. In on-device part of the prototype the main on-the-fly algorithm runs over already created contract and policy as AMT and makes a significant amount of calls to the decision procedure while it's execution.

Let us now describe initial architecture for Java platform that subsequently remained the same for the .NET architecture. The initial parsing algorithm just transforms a contract (resp. a policy) into a Java class, ContractAutomaton.java (resp. PolicyAutomaton.java) that can be directly manipulated by the actual algorithm responsible for the on-the-fly policy matching (i.e. emptiness test). If the policy option is specified then the parser also performs the complementation of the policy. Management of the variables declaration is discussed later in the §5.

So far we have not used any of the tricks of the trade (e.g. bit state hashing) that characterize the symbolic automata representations because we wanted the contract and policy to be manageable under a generic Java MIDP platform without need of extra libraries. Also, the current experiments that we have run on larger applications do not seem to require managing significantly big automata but rather many little automata, one per each security rule. Another reason was that we wanted the policy to be also potentially executable in parallel with the code.

Since a contract-policy matching algorithm should frequently call the decision procedure during its' running, we have found a design decision for an internal representation of $\mathcal{AMT}$. This particular form of $\mathcal{AMT}$ supports all the options of integration with solver that we address in this paper.

Let us describe this form in more details. We associate a number of variables to every edge, where $method$ is an API call that the policy is supposed to rule, $cond$ - a guarded command which must be true in order for the method to be executed, for instance a $cond$ specifies that the url must start with the string "https".

For further representation simplification, we follow the semantics for security automata proposed in [1] so that we have a prioritized execution among guards: we go to the next guard only if the guards before it have all failed. Such information is represented in $otherConds$ - the other guarded commands that failed before reaching the current guard $otherMethods$ - an expression consists of all other methods that are not supposed to rule at the current moment e.g. $\neg m_1 \wedge \neg m_2$ where m_1 and m_2 are methods that are not supposed to rule.

Once contract and policy automata are made available to the main system, the latter can run the on-the-fly procedure which has been also implemented in Java using only MIDP libraries to guarantee portability (and we have similarly developed a .NET mobile implementation in C#).

Next stage is non-trivial point because we need to interact with a state-of-the-art decision procedure for mathematical theories. We took the design decision to use the solver as a black box for the general algorithm so it gives the answer whether the problem is satisfiable or not. During the process of implementation it appeared to be not entirely possible[2] we have tried to be close as possible to this decision. In this way it could be easy to also try a different decision procedure such as MathSAT by Bozzano et al. [9], DPLL(T) by Tinelli [24] or CVC-lite [4]. For the same reason we have further decided to interface with the solver without using its internal data structure but rather to interact with the decision procedure by using strings. While this creates a bit of overhead for parsing, it makes it significantly easier to replace the solver. An

[2]The obvious reason is there are issues related to presence of O.S. libraries that might be required by a solver and not by another one. More subtle reasons are related to garbage collection and are further discussed in Sec. 5.

industry level application committing to a particular solver would likely bypass this step.

Among the different possibilities we have used the decision procedure libraries behind the tools MathSAT and NuSMV [12]. In this way we could support expressions in the edges of the automaton modulo theory that are arbitrarily complex boolean expression, mathematical expression and uninterpreted function symbols as we have shown in §3.

5 Design Decisions

Different design decisions are made in order to *decide the best configuration of integrating automata-based inclusion algorithm with decision procedure* as the problem is not trivial. Every option of the configuration we propose below has different memory impact and this information and results of such analysis is very important because of the resource constraints of mobile device. This is not studied in classical decision procedure integration papers because the problem of resources is irrelevant. The time in classical research is considered different if it is linear or exponential, i.e. constant factor is not taken into account. For achieving our goal even small changes in time makes sense.

In integrating matching algorithm with the theory solver we faced a number of design options:

One_vs_Many Solver in object oriented languages is by itself an object. We could either create only one instance of solver, relying on the solver to assert and retract expressions on demand, or create a new instance of the solver every time we call the decision procedure.

MUTEX_SOLVER if an edge in the automaton correspond to a call to a method it is obviously incompatible with another edge calling a different method. Such constraints could be directly incorporated into the algorithm without the need to represent them as boolean mutual exclusion constraints on the boolean variables representing method invocations. In this case all the method names are declared as mutex constants at the moment of declaring all variables, then the expression sent to the solver has the following structure: $method = name \wedge cond \wedge otherConds$. Hence, if the method names of two edges are not the same then the DecisionProcedure returns false.

MUTEX_MC allows the on-the-fly algorithm to check whether method names are the same. The DecisionProcedure is called with parameters: $cond \wedge otherConds$ only if this check is passed.

PRIORITY_MC the semantics for security policy is that guards are evaluated using *priority or* hence we can optimize the expressions sent to the decision procedure as lemmas. Using the lemma, the Expression sent to the DecisionProcedure is minimized and it has only $cond$.

CACHING_MC Since many edges will be traversed again and again we could save time by caching the results of the matching. The solver itself has a caching mechanism that could be equally used (CACHING_SOLVER).

While we assumed that all decision could be just taken after considering preliminary experimental results it turned out that at least for the One_vs_Many decision this was not possible. The cause is the management of garbage collection both by the Java virtual machine and by the libraries of MathSAT/NuSMV which requires only one instance of solver exists at time in order to interact correctly with the NuSMV library. This leads to use a static invocation for the solver and set significant constraints on the interaction.

For example, before starting to visit all constraints to the library, all variables used in expressions must be declared. The NuSMV library has to invoke *DeclareNewBooleanVar, DeclareNewWordVar, DeclareNewStringVar* methods for declaration of boolean, integer and string variables respectively. Only after declaring all the variables from contract and policy expressions, the on-the-fly algorithm can actually start invoking the decision procedure in its visit. A consequence of this rule is that with this implementation we cannot insert edges that introduce new variables because the solver can be called only after declaring all the variables and adding all the needed constraints.

Therefore, during the visit of the algorithm we must at first upload constraints to the solver with the *AddConstraint* method of the NuSMV class and then remove them with the *RemoveConstraint*. As per today, we still occasionally get bugs in the NuSMV/MathSAT solver or the main on-the-fly algorithm due to this complication of a single instance object.

The rest design alternatives can be implemented and tested thus giving way to the six alternative configurations (see Fig. 4d) of the interactions between the solver and the on-the-fly emptiness check algorithm.

6 Experiments on Desktop and on Device

To select the best option we collected data on resources used, namely number of visited states, number of visited transitions, running time for each problem in each design alternative, and the number of solved problems against time. For sake of example we list in Table 3 some sample possible combinations of policy-contract (mis)matching pairs. For instance, the contract pimNoConn_contract.pol represents an ex.1 and corresponds to the $\mathcal{AMT}$ shown in Fig.2a. Similarly, the policy pimSecConn_policy.pol corresponds to ex.2 and related $\mathcal{AMT}$ in Fig.2b.

With the exception of the pathological problem P100, which has been designed that way, most problems have few states and transitions and, as we shall see in the next table (Table 4 showing performance of ten times run for each problem set and each design alternative), they also require little time for being assessed.

Let us notice that the number of states and transitions in the $\mathcal{AMT}$ for each contract and policy in Table 3 is a number of reachable states and transitions. During the running of matching algorithm there may be the case when the algorithm stops working (producing "do not match" answer) without reaching all the states of contract and/or policy. And this case is explicitly shown in P6, P7 and P8 examples in Table 4. That is why we only present here the number of reachable states in Table 3 and number of visited states during on-the-fly running in Table 4.

We run our experiments on a Desktop PC (Intel(R) Pentium(R) D CPU 3.40GHz, 3389.442MHz, 1.99GB of RAM, 2048 KB cache size) with operating system Linux version 2.6.20-16-generic, Kubuntu 7.04 (Feisty Fawn). Currently, we are also porting the application to the mobile for actual detailed profiling, namely HTC P3600 (3G PDA phone) with ROM 128MB, RAM 64MB, Samsung®SC32442A processor 400MHz

Table 3: Problems Suit

Problem	Contract	Policy	SC	TC	SP	TP
P1	size_100_512_contract.pol	size_10_1024_policy.pol	2	4	2	4
P2	maxKB512_contract.pol	maxKB1024_policy.pol	2	4	2	4
P3	noPushRegistry_contract.pol	oneConnRegistry_policy.pol	2	3	3	9
P4	notCreateRS_contract.pol	notCreateSharedRS_policy.pol	2	4	2	4
P5	pimNoConn_contract.pol	pimSecConn_policy.pol	3	7	3	9
P6	2hard_contract.pol	2hard_policy.pol	3	7	3	7
P7	httpI_contract.pol	httpsI_policy.pol	3	7	3	7
P8	3hard_contract.pol	3hard_policy.pol	3	7	3	7
P100	noSMS_contract.pol	100SMS_policy.pol	2	4	102	304

(a) Abbreviations

SC: Number of States Contract TC: Number of Transitions Contract
SP: Number of States Policy TP: Number of Transitions Policy

and operating system Microsoft®Windows Mobile®5.0 with Direct Push technology.

For the sake of example we present the result obtained for alternative with MUTEX_MC ONE_INSTANCE CACHING_SOLVER in Table 4. These results are mapped into diagram shown in Fig.4a for matching problems and Fig.4c for not matching problems. Notice that we only provide the cumulative running time that is necessary to solve all problems as in the CASC theorem proving competition. This is important because our goal is to match (or not match) all rules in a contract with all corresponding rules in a policy. Thus, the value of the single problem is not important except for some cases where the average output might be significantly off due to some off scale rule.

We singled out P100 as a challenging artificial problem because it has a large number of states compared to the others: essentially this happened because we draw an automaton modulo theory with 100 states and which traverse from one state to another by adding 1 to the number of SMS sent. In this case there is a difference between M1 and M2, namely around 8%. In order to study this anomaly in more details, we generated more unreal problem sets: as P100 with combination of sent SMS none, 1, 10, and 100 for both contract and policy. The generated cases cumulative running time of implementation is propositional to the number of problems solved (see Fig.4b).

All methods seem to perform equally well because the problems are not stressful enough for the different configurations. This is actually a promising result for the deployment to the resource constrained in mobile device domain. Therefore, we have implemented the same algorithm for the mobile platform HTC P3600 (3G PDA phone). We run the problem suit of P1-P8 and P100 with MUTEX_MC ONE_INSTANCE CACHING_SOLVER configuration.

Table 4 shows the results on device, where the runtime of every single problem running is longer than on Desktop PC. This result is obvious due to higher performance of desktop platform. However, the cumulative time of solved problems is still manageable for the mobile user to obtain. The algorithm's runtime will be longer for the problems that match (the algorithm has to run over all states until the cycle is found) than for the problems that do not match (the algorithm stops working as soon as counterexample is found). Note also that the number of visited states and transitions for the matched problems are the same exactly because of the search all over the states; otherwise the counterexample can be found in a different time and it does not depend on the run.

Table 4: Running Problem Suit 10 Times

(a) Running Problem Suit

MUTEX_MC ONE_INSTANCE CACHING_SOLVER

Problem	Desktop				Mobile				Result
	ART (s)	CRT (s)	SV	TV	ART (s)	CRT (s)	SV	TV	
P1	2.4	2.4	2	6	4.3	4.3	2	6	Match
P2	2.4	4.8	2	6	4.1	8.4	2	6	Match
P3	2.4	7.2	3	11	3.9	12.3	3	11	Match
P4	2.4	9.6	2	6	4.0	16.3	2	6	Match
P5	4.7	14.3	3	11	4.1	20.4	3	11	Match
P6	2.9	2.9	4	4	3.8	3.8	3	6	Not Match
P7	2.8	5.7	5	7	3.8	7.6	2	4	Not Match
P8	2.9	8.6	5	7	3.8	11.4	3	6	Not Match
P100	9.3	9.3	102	307	11.3	11.3	102	307	Match

(b) Abbreviations

ART: Average Runtime for 10 runs SV: Number of Visited States
CRT: Cumulative Average Runtime TV: Number of Visited Transitions

Cumulative time of problems is presented in Fig. 5a for matching and Fig.5b for not matching.

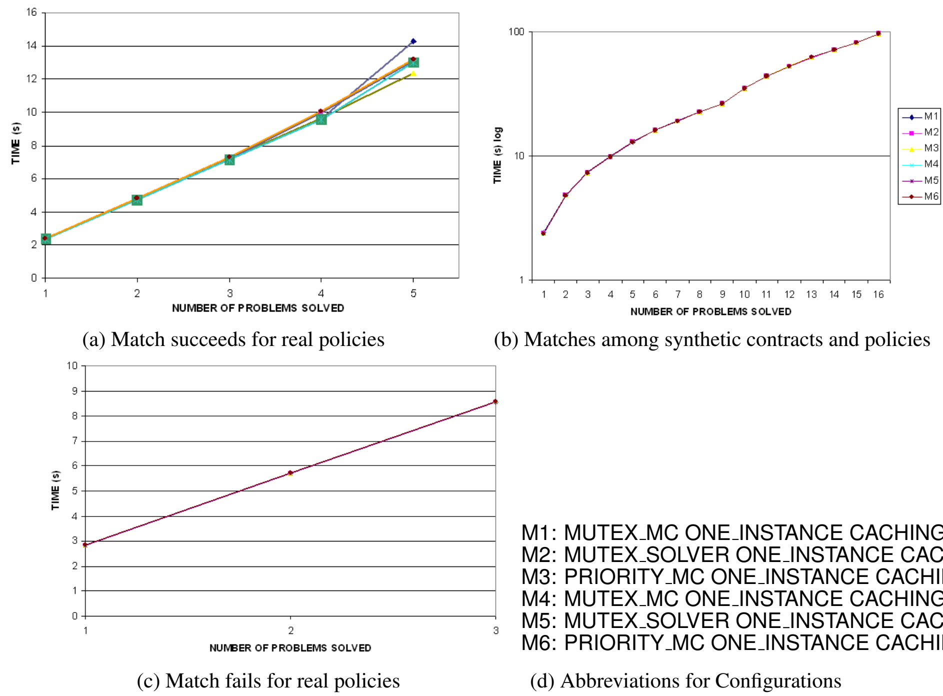
In this paper we state the time of the running on the mobile platform for one design decision just in order to give the reader a feeling how the matching algorithm with integrated decision procedure can run in real life and that it will take a reasonable time. Even for a policy that is transformed into Automata Modulo Theory with hundred of states and in case algorithm reaches all of them, it is still takes only 11.3 seconds to complete the procedure.

Our current implementation uses PRIORITY_MC ONE_INSTANCE CACHING_MC configuration. PRIORITY_MC is preferred because of the nature of rules in policies which is *priority or*, also because MUTEX_SOLVER does not allow empty methods such as $\neg m_i \wedge \neg m_j$ which is possible in the matching algorithm. ONE_INSTANCE is chosen because of garbage collection problem. CACHING_MC is desired in order to save calls to solver for the already solved rules.

7 Related Work and Conclusions

Mobile code security can be achieved by several approaches, for example *code signing* to ensure the origin of the code by trust relationship, *proof-carrying code (PCC)* to ensure safety by explicit proof, *model-carrying code (MCC)* that carries security-relevant behavior of the producer mobile code [28], and *security-by-contract (SxC)* where a digital signature should not just certify the origin of the code but rather bind together the code with a contract [13].

Security-by-contract (SxC). Security-by-contract [13] proposed to augment mobile code with a claim on its security behavior that can be matched against a mobile platform policy on-the-fly, which provides semantics for digital signatures on mobile code. Security-by-contract attempts to overcome the major limitation of MCC, namely not



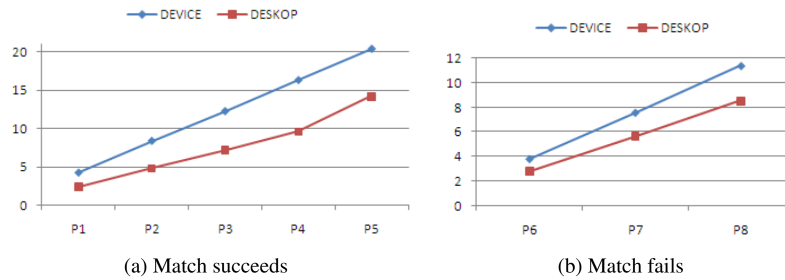
(a) Match succeeds for real policies

(b) Matches among synthetic contracts and policies

(c) Match fails for real policies

(d) Abbreviations for Configurations

Figure 4: Cumulative response time of matching algorithm on Desktop PC



(a) Match succeeds

(b) Match fails

Figure 5: Cumulative response time of matching algorithm on the Mobile Device

fully developed issue of contract matching and limited to finite state automata which are too simple to describe realistic policies. In coping with this challenge, we propose an application of formal methods that goes beyond the traditional realm of off-line verification of formal properties of hardware and software. The formal model considered for capturing contracts and policies is based on the novel concept of *Automata Modulo Theory* ($\mathcal{AMT}$).

Off-line Verification. Our approach is different from off-line verification while we use integration of emptiness test for automata modulo theory with satisfiability using decision procedures. Such reasoning capabilities should then be used at the time an application is downloaded on a mobile application such as PDA or a smart phone. The usage of decision procedures allowed us to cope with automata modulo theories where edges are not just finite states of labels but rather expressions that can capture infinite transitions such as “connect only to urls starting with https://”. In the off-line verifi-

cation realm, the idea of embedding decision procedures into a higher level reasoner is well accepted and was one of the strongholds of the PVS system. At theoretical level Tinelli in [30] combines order-sorted first-order theories and their decision procedures for theories satisfying certain conditions into a decision procedure for their union, where SMT problems themselves can be addressed by tools such as CVC [4], UCLID [10], MathSAT [9].

Infinite States System. Infinite numbers of transitions in security policies by labeling each transition with a computable predicate instead of an atomic symbol has been studied in [26] and implemented in systems like PoET/PSLang toolkit [16]. Edit automata [5] extend security automata to model the transforming effects of in-lined reference monitors and is implemented in the Polymer system [6]. These approaches focus on the relations between code and security claims on the code. The Mobile system [19] implements a linear decision algorithm that verifies that annotated .NET bytecode binaries satisfy a class of policies that includes security automata and edit automata.

Conclusions. We have described the prototype implementation, its integration with a state of the art decision solver (based on MathSAT and NuSMV) and the preliminary experiments that we have done for contract-policy matching in order to select the more suitable design combination. We are currently undertaking development into two major directions: porting the implementations on the mobile and developing a version working for .NET.

Acknowledgments

We would like to thank Prof. F. Massacci for his insightful comments and suggestions. We thank M. Roveri and A. Cimatti for the support in the usage of the NuSMV and MathSAT libraries and for hammering down a decision procedure for URLs. We also acknowledge Marco Dalla Torre for support in the integration of the tools.

References

[1] I. Aktug and K. Naliuka. Conspec – a formal language for policy specification. In *Proc. of the 1st Int. Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM2007)*, ENTCS. Elsevier Sci., 2007. To appear.

[2] M. Arikawa, S. Konomi, and K. Ohnishi. Navitime: Supporting pedestrian navigation in the real world. *IEEE Pervasive Comp. Magazine*, 6(3):21–29, 2007.

[3] J. Bacon. Toward pervasive computing. *IEEE Pervasive Comp. Magazine*, 1(2):84, 2002.

[4] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. of CAV'04*, 2004.

[5] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Found. of Comp. Security*, Copenhagen, Denmark, July 2002.

[6] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *Proc. of the ACM SIGPLAN 2005 Conf. on Prog. Lang. Design and Implementation*, pages 305–314. ACM Press, 2005.

[7] L. Bauer, J. Ligatti, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *Int. J. of Inform. Sec.*, 4(1-2):2–16, 2005.

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

[9] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *J. of Autom. Reas.*, 35(1–3):265–293, 2005.

[10] R. E. Bryant, S.K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. of CAV'02*, pages 78–92, London, UK, 2002. Springer-Verlag.

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

[12] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc. of CAV'02*, 2002.

[13] 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*. Springer-Verlag, 2007.

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

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

[16] U. Erlingsson and F.B. Schneider. Irm enforcement of java stack inspection. In *Proc. of Symp. on Sec. and Privacy*, page 246. IEEE Press, 2000.

[17] L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.

[18] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *TOPLAS*, 28(1):175–205, 2006.

[19] K.W. Hamlen, G. Morrisett, and F.B. Schneider. Certified in-lined reference monitoring on .net. In *Proc. of the 2006 workshop on Prog. Lang. and analysis for security*, pages 7–16, New York, NY, USA, 2006. ACM Press.

[20] M. Hilty, A. Pretschner, C. Schaefer, and T. Walter. Usage control requirements in mobile and ubiquitous computing applications. In *Proc. of the Int. Conf. on Sys. and Net. Comm. (ICSNC 2006)*, page 27. IEEE Press, 2006.

[21] B. LaMacchia and S. Lange. *.NET Framework security*. Addison Wesley, 2002.

[22] F. Massacci and I. Siahaan. Matching midlet's security claims with a platform security policy using automata modulo theory. In *Proc. of The 12th Nordic Workshop on Secure IT Systems (NordSec'07)*, 2007.

[23] G.C. Necula. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang.*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[24] 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, 2006.

[25] 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.

[26] F.B. Schneider. Enforceable security policies. *TISSEC*, 3(1):30–50, 2000.

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

[28] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D.C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proc. of the 19th ACM Symp. on Operating Sys. Princ.*, pages 15–28. ACM Press, 2003.

[29] MOBIUS Project Team. Framework- and application-specific security requirements. Public Deliverable of EU Research Project D1.2, MOBIUS - Mobility, Ubiquity and Security, Report available at http://mobius.inria.fr, 2006.

[30] C. Tinelli and C. Zarba. Combining decision procedures for sorted theories. In J. Alferes and J. Leite, editors, *Proc. of JELIA'05*, volume 3229 of *LNAI*, pages 641–653. Springer-Verlag, 2004.

[31] D. Vanoverberghe, P. Philippaerts, L. Desmet, W. Joosen, F. Piessens, K. Naliuka, and F. Massacci. A flexible security architecture to support third-party applications on mobile devices. In *Proc. of the 1st ACM Comp. Sec. Arch. Workshop*, 2007.

[32] M.Y. Vardi. Büchi complementation a 40-year saga. March 2006.

[33] A. Zobel, C. Simoni, D. Piazza, X. Nuez, and Daniel Rodriguez. Business case and security requirements. Public Deliverable of EU Research Project D5.1.1, S3MS- Security of Software and Services for Mobile Systems, Report available at www.s3ms.org, 2006.