Matching Policies with Security Claims of Mobile Applications[*]

Nataliia Bielova Marco Dalla Torre Nicola Dragoni Ida Siahaan
University of Trento
surname@dit.unitn.it

Abstract

*The Security-by-Contract (*S×C*) framework has been recently proposed to address the* trust relationship *problem of the current security model adopted for mobile devices. The key idea of* S×C *(similar to the one of Model-Carrying Code) is to augment mobile code with a claim on its security behavior (a* contract*) that could be matched against a mobile platform policy before downloading the code. The rational is that, thanks to* S×C*, a digital signature does not just certify the origin of the code but also bind together the code with a contract. In this paper we address one of the key issue of the* S×C *paradigm, namely the contract-policy matching problem, proposing a prototype for matching policies with security claims of mobile applications. This result can be considered a key step towards the achievement of the* S×C *main goal: provide a semantics for digital signatures on mobile code, thus being a step in the transition from trusted code to trustworthy code.*

1. Introduction

Mobile devices are increasingly popular and powerful. Yet, the growth in computing power of such devices has not been supported by a comparable growth in available software. One of the reasons for this lack of applications is also the current security model adopted for mobile phones, which is essentially based on *trust relationships*: mobile code is accepted if it is digitally signed by a trusted party. The problem with trust relationship is twofold. At first we can only reject or accept the signature. This means that interoperability in a domain is either total or not existing: an application from a not-so-trusted source can be denied network access, but it cannot be denied access to a specific protocol, or to a specific domain. The second (and major) problem, is that *there is no semantics attached to the signature.* This is a problem for both code producers and consumers.

The *Security-by-Contract* (S×C) framework [3] has been recently proposed to address this unsatisfactory situation. The key idea (similar to the one of Model-Carrying Code [7]) is that the digital signature should not just certify the origin of the code but rather bind together the code with a contract. Loosely speaking, a *contract* contains a description of the relevant features of the application and the relevant interactions with its host platform. A mobile platform could specify platform contractual requirements, a *policy*[1], which should be matched by the application's contract.

1.1. Contribution of this Paper

In this paper we address one of the key issue of the S×C paradigm, i.e. the contract-policy matching problem: *given a contract that an application carries with itself and a policy that a platform specifies, is the contract compliant with the policy?*

The main contribution of this paper is a prototype that matches security claims of mobile code with the security desires of the platform. Some preliminary works, such as [3, 5], have studied this issue from different points of view, but no one has actually solved the contract-policy matching problem. Therefore, this contribution can be considered a key step towards the achievement of the S×C main goal: provide a semantics for digital signatures on mobile code, thus being a step in the transition from trusted to trustworthy code.

The rest of the paper is organized as follows. We start in Section 2 presenting the basic notions of the S×C framework, focusing on the contract-policy matching issue. Then in Section 3 we briefly discuss the policy language exploited to specify contracts and policies in the S×C framework. In Section 4 we introduce the theory underlying the implemented matching algo-

[*] This work is partly funded by the project EU-IST-STREP-S3MS (www.s3ms.org).

[1] In the sequel we will refer to policy as the security requirements on the platform side and by contract the security claims made by the mobile code.

rithm, showing how ConSpec specifications are mapped into the theory. Finally, in Section 5 we describe the overall matching prototype. We conclude the paper highlighting some ongoing future works.

2. S×C Framework

The S×C framework for mobile code is essentially shaped by three groups of stake-holders: mobile operator, service provider and/or developer, mobile user. Mobile code developers are responsible to provide a description of the security behavior of their code.

Definition 2.1 (Contract) *A contract is a formal complete and correct specification of the behavior of an application for what concerns relevant security actions (Virtual Machine API Calls, Operating System Calls).*

Loosely speaking, a contract contains a description of the relevant features of the application and the relevant interactions with its host platform. Security contract may include fine-grained resource control (e.g., silently initiate a phone call or send an SMS), memory usage, secure and insecure web connections, user privacy protection, confidentiality of application data, constraints on access from other applications already on the platform. By signing the code the developer binds it with the claims on its security-relevant behavior, i.e. its contract, and thus provides a *semantics to digital signatures.* This represents one of the key ideas behind the security-by-contract approach: a digital signature should not just certify the origin of the code but rather bind together the code with a contract describing its security relevant features.

On the other side we can see that users and mobile phone operators are interested that all codes that are deployed on their platform are secure. In other words they must declare their security policy:

Definition 2.2 (Policy) *A policy is a formal complete specification of the acceptable behavior of applications to be executed on the platform for what concerns relevant security actions (Virtual Machine API Calls, Operating System Calls).*

A contract should be negotiated and enforced during development, at time of delivery and loading, and during execution of the mobile application.

2.1. Contract-Policy Matching

A key problem to be addressed to bring the S×C framework to its full potential is the contract-policy matching issue: *given a contract that an application carries with itself and a policy that a platform specifies, is the contract compliant with the policy?*

Contract-policy matching represents a common problem in the life-cycle because it must be done at all levels: both for development and run-time operation. Intuitively, matching should succeed if and only if by executing the application on the platform every behavior of the application that satisfies its contract also satisfies the platform's policy. More formally[2]:

Definition 2.3 (Exact Matching) *Matching should succeed if and only if by executing the application on the platform every trace that satisfies the application's contract also satisfies the platform's policy.*

Running Examples. The proposed concepts, which are further detailed in the rest of this article, can be illustrated by using the following two running examples.

Example 1 *Let us consider an application's contract that consists of two significant rules: (1) the application only uses HTTPS network connections; (2) no messages can be sent by the application.*

The platform's policy has two rules: (1) the application uses only high-level (HTTP, HTTPS) network connections; (2) maximum five text messages can be sent by the application.

It should be intuitive that in this case the application's contract matches the platform's policy. In fact, the security behaviour claimed in the application's contract correspond to the allowed security behaviour stated in the platform's policy.

Example 2 *Let us consider now an application's contract with just one rule ensuring that the amount of data once received by application is bounded by 1024 Kb.*

The platform's policy has one rule allowing only to receive the amount of data bounded by 500 Kb.

In this case, the contract-policy matching must fail, since the application can receive more data than the one allowed by the mobile platform.

3. Contract and Policy Specification

In this Section we provide an overview of the BNF syntax of ConSpec, the language exploited to specify contracts and policies within the context of the S×C framework. A full description of the language is outside the scope of the paper (interested readers can consult [1]). For the sake of simplicity and space limits, herein we focus only on the main features of the language.

[2] See [3] for a more formal treatment of these concepts

3.1. ConSpec Syntax

A specification in ConSpec is a non-empty list of rules. Each rule is defined for the specific area of contract (e.g. rule for the SMS messages, for Bluetooth connections etc.) and describes security properties for the given area. Fig. 1 shows a fragment of the ConSpec syntax for specifying one single rule.

```
MAXINT MaxIntValue
MAXLEN MaxLenValue
RuleID Identifier

SCOPE <Object ClassName | Session | MultiSession
    | Global>

SECURITY STATE
    [CONST] | <bool | int | string>
                      VarName1 = <DefaultValue1>
    | <int> VarName2 = <DefaultValue2>
                      RANGE <FromValue> ..  <ToValue>
                      ...

<BEFORE | AFTER | EXCEPTIONAL> EVENT MethodSignature1
    PERFORM
    condition1 -> action1
               ...
    conditionM1 | ELSE> -> actionM1
                           ...
<BEFORE | AFTER | EXCEPTIONAL> EVENT MethodSignatureK
    PERFORM
    condition1 -> action1
               ...
    conditionMK | ELSE> -> actionMK
```

Figure 1: A Fragment of the ConSpec Syntax

The `RuleID` tag identifies the area of the contract, e.g. for restriction of sending text messages the identifier could be `"TEXT_MESSAGES"` or for accessing the file system the identifier could be `"FILE_ACCESS"`.

Each rule consists of three parts: scope definition, state declaration and list of event clauses.

There are different scopes in ConSpec: scope `Object` is used when the rule can be applied for the object of specific class; scope `Session` if the security properties are applicable for the single run of the application; scope `Multisession` when the rule describes behavior of the application during it's multiple runs and scope `Global` for executions of all applications of a system.

The state declaration defines the state variables to be used in the current rule of ConSpec specification. The variables can be constant and non-constant. All the non-constant variables characterize the state of the automaton defined by the rule. Constant variables are simply used in the specification and don't play significant role in automaton construction.

Variables can be boolean, integer or string. As the states have to be finite all the types have to be bounded. For this reason ConSpec specification has two tags: `MAXINT` to define maximum value of integer and `MAXLEN` to define maximum length of string. In some cases the variable should have less interval then the keyword `RANGE` is used for more precise bounding.

Event clauses define the transitions of the automaton constructed from the ConSpec rule. Each event clause has the list of guarded commands and update blocks which will be performed when the guarded command holds.

Every event is defined by a modifier and a signature API method, including name of the class, method name and optionally list of parameters. The modifiers (`BEFORE`, `AFTER` and `EXCEPTIONAL`) indicate in which moment the update block must be executed.

`Condition` is a boolean expression on the state variables and possible parameters of the method. `Condition` can be replaced by the `ELSE` keyword; in this case the corresponding `UpdateBlock` will perform only if all the other blocks evaluated to false. If `Condition` is equal to *false*, then the current event can never run according to this specification.

Example 3 *Fig. 2-3 show the ConSpec specifications of the contract and policy of Ex. 1, respectively.*

```
MAXINT 10000 MAXLEN 10
RULEID HIGH_LEVEL_CONNECTIONS

SCOPE Session

SECURITY STATE
boolean opened = false;

BEFORE javax.microedition.io.Connector.open
(string url) PERFORM
url.startsWith("https://") && !opened ->
  {opened = true;}
url.startsWith("https://") && opened -> {skip;}


RULEID SMS_MESSAGES
SCOPE Session

SECURITY STATE

BEFORE javax.wireless.messaging.MessageConnection.send
(javax.wireless.messaging.TextMessage msg) PERFORM
  false -> {skip;}

AFTER javax.wireless.messaging.MessageConnection.send
(javax.wireless.messaging.TestMessage msg) PERFORM
  false -> {skip;}
```

Figure 2: ConSpec Spec. of the Contract from Ex.1

Example 4 *Fig. 4-5 show the ConSpec specifications of the contract and the policy of Ex. 2, respectively.*

```
MAXINT 10000 MAXLEN 10
RULEID HIGH_LEVEL_CONNECTIONS

SCOPE Session

SECURITY STATE
boolean opened = false;

BEFORE javax.microedition.io.Connector.open
(string url)
PERFORM
(url.startsWith("http://") || url.startsWith("https://"))
    && !opened -> {opened = true;}
(url.startsWith("http://") || url.startsWith("https://"))
    && opened -> {skip;}


RULEID SMS_MESSAGES
SCOPE Session

SECURITY STATE
CONST int maxMessage = 5;
int messageSent = 0 RANGE 0..5;

BEFORE javax.wireless.messaging.MessageConnection.send
(javax.wireless.messaging.TextMessage msg) PERFORM
  messageSent<maxMessage -> {skip;}

AFTER javax.wireless.messaging.MessageConnection.send
(javax.wireless.messaging.TextMessage msg) PERFORM
  true -> {messageSent = messageSent + 1;}
```

Figure 3: ConSpec Spec. of the Policy from Ex.1

```
MAXINT 10000 MAXLEN 10
RULEID LIMITED_DATA

SCOPE Session

SECURITY STATE
CONST int maxKbRecieve = 1024;

BEFORE System.Net.Sockets.BeginReceive
(Byte[] buffer, int offset, int size,
System.Net.Sockets.SocketFlags socketFlags,
System.AsyncCallback callback, Object state)
PERFORM
    size < maxKbRecieve -> {skip;}
```

Figure 4: ConSpec Spec. of the Contract from Ex.2

4. Automata Modulo Theory ($\mathcal{AMT}$)

Having contract and policy specified in ConSpec, we would like to concretely solve the problem of matching the security claims of the code (contract) with the security desired by the platform (policy). The problem seems essentially solved, in [7] and [3], but none has actually solved it. For instance, in [3] 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 matching is discussed in [5], but only from a theoretical point of view. The key idea is based on the introduction of the concept of *Automata Modulo Theory* ($\mathcal{AMT}$). $\mathcal{AMT}$ is an extension of Büchi Automata (BA), suitable for formalizing systems with finite states but infinite transitions. $\mathcal{AMT}$ enables us to define very expressive and customizable policies as a model for *security-by-contract* as in [3] and model-carrying code [7] by capturing the infinite transition into finite transitions labeled as expressions in defined theories. To represent the security behavior, a system can be represented as an automaton where transitions corresponds to the invoked methods as in the works on model-carrying code [7]. In this case, the operation of matching the application's claim with platform policy is a classical problem in automata theory, known as *language inclusion* [2]. 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). Assuming that the automata are closed under intersection and complementation, the matching problem can be reduced to an emptiness test:

```
MAXINT 10000 MAXLEN 10
RULEID LIMITED_DATA

SCOPE Session

SECURITY STATE
CONST int maxKbRecieve = 500;

BEFORE System.Net.Sockets.BeginReceive
(Byte[] buffer, int offset, int size,
System.Net.Sockets.SocketFlags socketFlags,
System.AsyncCallback callback, Object state)
PERFORM
    size < maxKbRecieve ->{skip;}
```

Figure 5: ConSpec Spec. of the Policy from Ex.2

$$\mathcal{L}_{\mathrm{Aut}^C} \subseteq \mathcal{L}_{\mathrm{Aut}^P} \Leftrightarrow \mathcal{L}_{\mathrm{Aut}^C} \cap \mathcal{L}_{\overline{\mathrm{Aut}^P}} = \emptyset$$

4.1. $\mathcal{AMT}$ Theory

The theory of $\mathcal{AMT}$ [5] is a combination of the theory of Büchi Automata (BA) with satisfiability-modulo-theory (SMT) problem. SMT problem, which decides the satisfiability of first-order formulas modulo background theories, pushes the envelope of formal verification based on effective SAT solvers. In contrast to classical security automata we prefer to use BA because, besides safety properties, there are also some liveness properties which have to be verified. An example of liveness is "The application uses all the permissions it requests".

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, $\Delta_{\mathcal{T}} : S \times E \rightarrow 2^S$ is labeled transition function, and $F \subseteq S$ is a set of accepting states.*

Returning to our running examples, we illustrate in Fig. 6 and Fig. 9 how $\mathcal{AMT}$ can be used to formally specify the security properties introduced in Ex. 1 and Ex. 2, respectively.

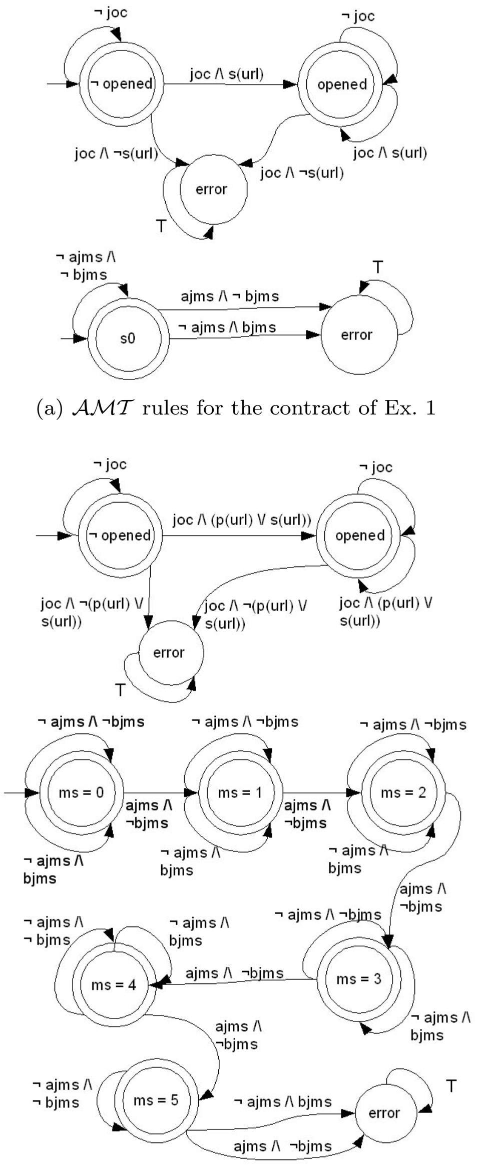
$\mathcal{AMT}$ operations for intersection and complementation require that the theory under consideration is closed under intersection and complementation (union is similar to the standard one). We consider only the *complementation of deterministic $\mathcal{AMT}$*, because in our application domain all security policies are naturally deterministic (as the platform owner should have a clear idea on what to allow or disallow) (further details can be found in [5]).

4.2. On-the-Fly State Model Checking with Decision Procedure

In $\mathcal{AMT}$ we are interested in finding counterexamples faster and we combine algorithm based on Nested DFS [6] 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 state. 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 then we report a security policy violation without further ado. Second, when a suspect state does not contain an error state of complemented policy we start a new depth first search from the suspect state to determine whether it is in a cycle, i.e. it is reachable from itself. If it is we report availability violation.

5. Matching Prototype

In this Section we describe the overall implemented prototype for contract-policy matching. We first describe the architecture of the prototype, focusing on how the prototype works, then we discuss what happens if the prototype is executed with the running examples as inputs. Finally, we describe some implementation details.



(a) $\mathcal{AMT}$ rules for the contract of Ex. 1

(b) $\mathcal{AMT}$ rules for the policy of Ex. 1

Abbreviations for JAVA APIs:

joc	$\doteq$	io.Connector.open(url)
$p(url)$	$\doteq$	url.startsWith("http://")
$s(url)$	$\doteq$	url.startsWith("https://")
$ajms$	$\doteq$	after MessageConnection.send(message)
$bjms$	$\doteq$	before MessageConnection.send(message)
s_0	$\doteq$	initial state, the system is staying in this state until it sends the message
ms	$\doteq$	amount of already sent messages

Figure 6: $\mathcal{AMT}$ Rules of Ex. 1

5.1. Architecture

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 (according to the On-the-Fly algorithm discussed in the previous Section). The prototype is basically composed of three tools: two ConSpec parsers and the main matching algorithm. One parser, written in C#, takes as input a ConSpec file and returns a Java source code file containing instructions on how to generate the list of automata objects retrieved from the ConSpec rules. A second parser, written in Java, is used to extract all the needed meta-information from the ConSpec file to allow correct identification and handling of the rules. Fig. 7 shows a sketch of the project workflow.

(1) The C# parser runs on contract and policy. The result is two files: ContractRules.java and PolicyRules.java.

(2) The main part of the algorithm starts: now two ConSpec files are read by Java parser. This parser creates two Policy class instances, each of them contains the list of rules grouped by scope. We also add to each rule the corresponding automaton (AutomatonMTT class instances) created by the C# parser, in order to perform inclusion match.

(3) With contract and policy represented as Policy class instances made by Java parser we start the matching procedure. For every rule in the policy we must have a corresponding rule in the contract with same RULEID and SCOPE tags: if this is not the case the whole match fails, otherwise we perform the inclusion match on this couple.

(4) If the inclusion match fails, the whole procedure halts with a “failure”. If it succeeds the procedure continues with the next pair of rules. The described step is repeated until all rules in the policy have been successfully checked against the rules in the contract.

Example. Back to our running examples, let us consider Ex.1. Given a contract and a policy in ConSpec (Fig.2 and 3) as inputs, the prototype translates that specifications into $\mathcal{AMT}$ rules (Fig.6). Such automata are then used as inputs of the On-the-Fly algorithm. Specifically, for each rule in the policy we search for corresponding rule in the contract and run On-the-Fly emptiness checking algorithm on the corresponding two $\mathcal{AMT}$ rules. The first pair of rules with the same `RuleID` are the one showed in Fig.2 and 3. Since the contract allows to use HTTPS connections only while the policy allows to use both HTTP and HTTPS connections the obtained result states that the contract matches the policy. The result of running the On-the-Fly algorithm with these two automata is shown in Fig. 8 (note that there are no cycles). For the other rule in the contract an appropriate rule in the policy is found. Here the contract forbids the application to send messages while the policy prescribes that the application can send bounded amount of messages. As a result, the matching algorithm ends successfully: the contract matches the policy.

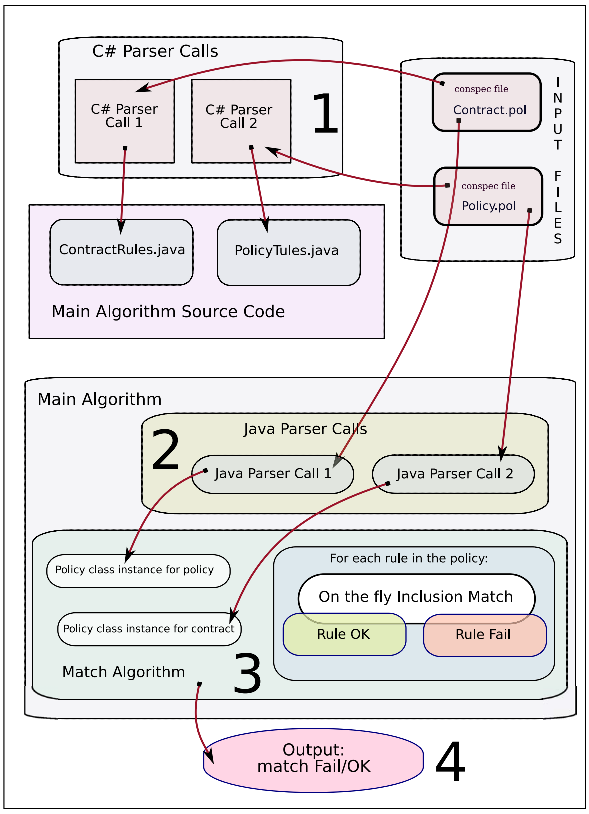


Figure 7: Prototype Architecture and Execution Flow

Let us focus on Ex. 2. The ConSpec specifications of Fig. 4 and 5 are translated into the two automata represented in Fig. 9. Here the matching fails because the algorithm finds a cycle. This is because the contract allows to receive more data then the policy.

5.2. Implementation Details

At the current state the prototype runs on a PC equipped with linux based operating systems and on 32 bits processors. We run our experiments on a laptop with an Intel Centrino processor and a Fedora Core 7 linux distribution. The following software must also be installed to run the prototype: (1) Java SDK version 6. (2) Apache Ant (http://ant.apache.org/). We need it so that we can execute the C# parser, compile the

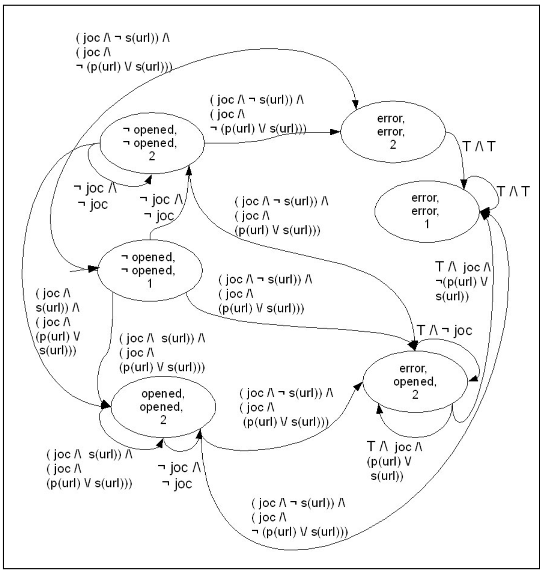
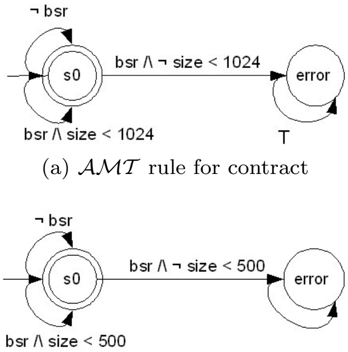

(joc /\ ¬ s(url)) /\ (joc /\ ¬ (p(url) \/ s(url)))
¬ opened, ¬ opened, 2
(joc /\ ¬ s(url)) /\ (joc /\ ¬ (p(url) \/ s(url)))
error, error, 2
T /\ T
error, error, 1
T /\ T
¬ joc /\ ¬ joc
¬ joc /\ ¬ joc
(joc /\ ¬ s(url)) /\ (joc /\ (p(url) \/ s(url)))
¬ opened, ¬ opened, 1
(joc /\ s(url)) /\ (joc /\ (p(url) \/ s(url)))
(joc /\ ¬ s(url)) /\ (joc /\ (p(url) \/ s(url)))
T /\ joc /\ ¬(p(url) \/ s(url))
T /\ ¬ joc
(joc /\ s(url)) /\ (joc /\ (p(url) \/ s(url)))
error, opened, 2
opened, opened, 2
(joc /\ ¬ s(url)) /\ (joc /\ (p(url) \/ s(url)))
T /\ joc /\ (p(url) \/ s(url))
(joc /\ s(url)) /\ (joc /\ (p(url) \/ s(url)))
¬ joc /\ ¬ joc
(joc /\ ¬ s(url)) /\ (joc /\ ¬ (p(url) \/ s(url)))

Figure 8: On-the-Fly algorithm: since there are no cycles, contract matches policy



(a) $\mathcal{AMT}$ rule for contract

(b) $\mathcal{AMT}$ rule for policy

Abbreviations for JAVA APIs:

bsr	$\doteq$	System.Net.Sockets.BeginReceive (Byte[] buffer, int offset, int size, System.Net.Sockets.SocketFlags socketFlags, System.AsyncCallback callback, Object state)
size	$\doteq$	the argument of `BeginRecieve` method
s_0	$\doteq$	initial state, while the size less then maximum the system is staying in this state

Figure 9: $\mathcal{AMT}$ Rules of Ex. 2

java sources and run them in an automated fashion. (3) Mono (http://www.mono-project.com) provides the necessary software to run .NET applications in non Microsoft environments.

Implementation of the Parser. The parser implements the mapping from a ConSpec policy to a java source file (list of AutomatonMTT class instances). The parser works as follows (Fig. 10). At first step (I), a syntax tree containing all the significant items of the policy is made from the ConSpec input file.

In the second transformation (II) the parser finds all the events and builds a specific AST structure. Each event now has a list of guarded commands. Each guarded command consists of condition on state variables, condition on parameters of the method and actions for the guard.

During next step (III) the automaton is built from the AST. This transformation is the most interesting. First, we generate the list of expressions that will be used for creating the transitions taking into account that only one security event at a time may happen. Second, we create all the states and then all the transitions for every state and every generated expression. The detailed procedure of mapping ConSpec to $\mathcal{AMT}$ is outside the scope of the paper. Interested readers can find it in [4].

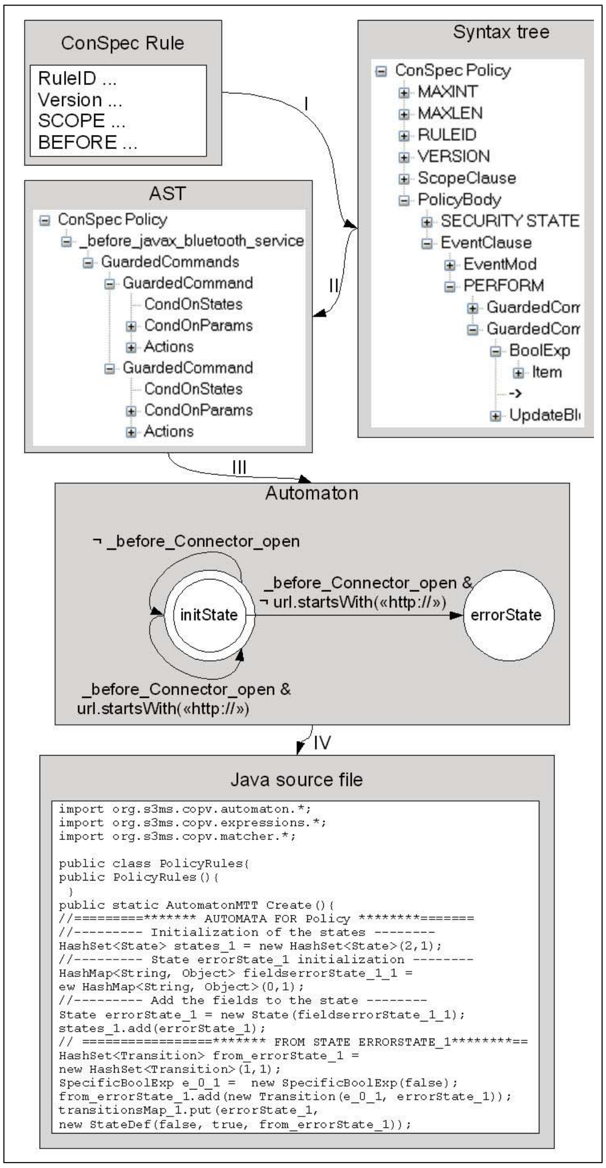
Finally, the last step (IV) creates the java source code containing the instance of the automaton.

Implementation of the On-the-Fly Algorithm. The On-the-Fly matching has been implemented according to the algorithm described in Section 4. Fig. 11 sketches a high-level view of the algorithm.

The On-the-Fly procedure interacts with the SMT solver NuSMV (http://nusmv.irst.itc.it/) for satisfiability checks. The instance of the NuSMV class is created only once at the beginning of the On-the-Fly procedure; then we declare variables, add constraints and remove constraints from the library every time we call the solver. Constraints for solver are often repeated during the algorithm running (at least we call the solver on the same constraints first time during searching for accepting state and second time searching for cycles). To avoid calling the solver frequently for the same problem we added two lists in the DFSAlgorithm class: Table_SAT and Table_UNSAT. The Table_SAT contains the constraints that are checked by the solver and result is SAT. Similarly Table_UNSAT contains the constraints that are checked by the solver and result is UNSAT.

6. Conclusions and Future Work

The main goal of this work has been to provide a concrete answer to the following question: *given a contract that an application carries with itself and a policy that a platform specifies, how can we check whether or*



```
import org.s3ms.copv.automaton.*;
import org.s3ms.copv.expressions.*;
import org.s3ms.copv.matcher.*;

public class PolicyRules{
public PolicyRules(){
 }
public static AutomatonMTT Create(){
//========******* AUTOMATA FOR Policy *******=======
//--------- Initialization of the states --------
HashSet<State> states_1 = new HashSet<State>(2,1);
//--------- State errorState_1 initialization --------
HashMap<String, Object> fieldserrorState_1_1 =
ew HashMap<String, Object>(0,1);
//--------- Add the fields to the state --------
State errorState_1 = new State(fieldserrorState_1_1);
states_1.add(errorState_1);
// ================******* FROM STATE ERRORSTATE_1********==
HashSet<Transition> from_errorState_1 =
new HashSet<Transition>(1,1);
SpecificBoolExp e_0_1 =  new SpecificBoolExp(false);
from_errorState_1.add(new Transition(e_0_1, errorState_1));
transitionsMap_1.put(errorState_1,
new StateDef(false, true, from_errorState_1));
```

Figure 10: ConSpec Parser Structure

not the contract is compliant with the policy?

To address this issue we have proposed a prototype implementing a matching algorithm based on a well-defined automata theory. In the paper we have presented both the theory and the prototype as well as several illustrative examples.

Future work will include the integration of several algorithms suitable for matching at "run-time" and the complete porting on mobile devices.

Acknowledgements. The authors thank Prof. Massacci for his insightful comments and suggestions.

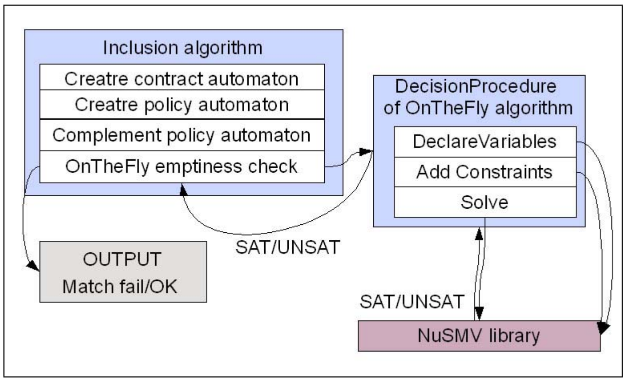


Figure 11: High-Level View of the On-the-Fly Alg.

References

[1] I. Aktug and K. Naliuka. Conspec - a formal language for policy specification. In Proc. of the 1st Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM2007), 2007.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[3] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Proc. of the 4th European PKI Worshop*, pages 297–312. Springer-Verlag, 2007.

[4] N. Dragoni, F. Massacci, K. Naliuka, I. Siahaan, T. Quillinan, I. Matteucci, and C. Schaefer. Methodologies and tools for contract matching. Public Deliverable D2.1.4, EU Project S3MS, Report available at www.s3ms.org, 2007.

[5] F. Massacci and I. Siahaan. Matching midlet's security claims with a platform security policy using automata modulo theory. NordSec, 2007.

[6] 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.

[7] 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 *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP-03)*, pages 15–28. ACM Press, 2003.