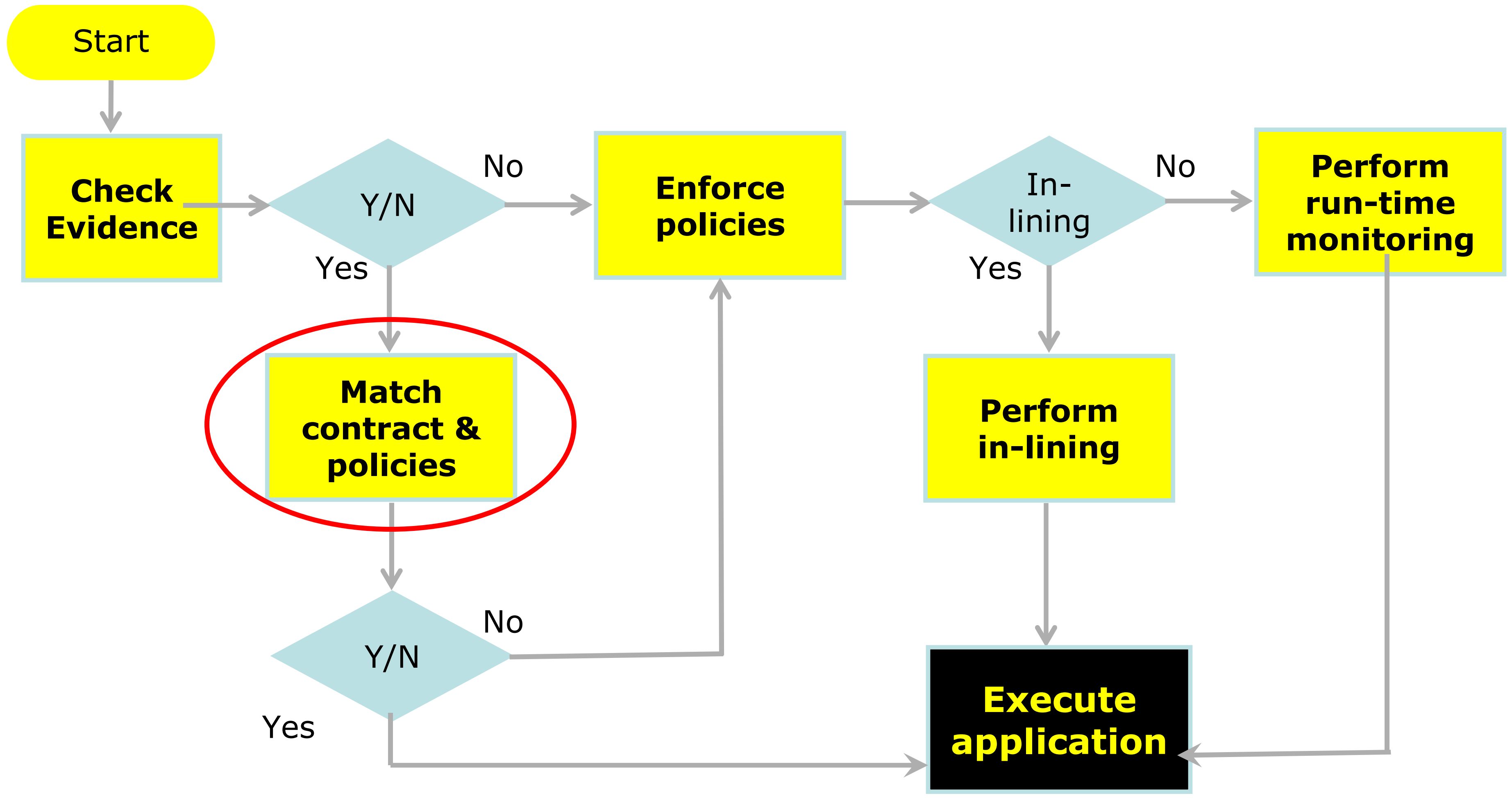
Testing Decision Procedures for Security-by-Contract
(Extended Abstract)

Nataliia Bielova, Fabio Massacci, Ida Siahan
University of Trento

Joint CSF/LICS 2008 Short-Talk Session on Logic and Security
Carnegie Mellon University, Pittsburgh – USA
June 24, 2008

SxC Workflow – User's View



On-the-fly Policy-Contract Algorithm

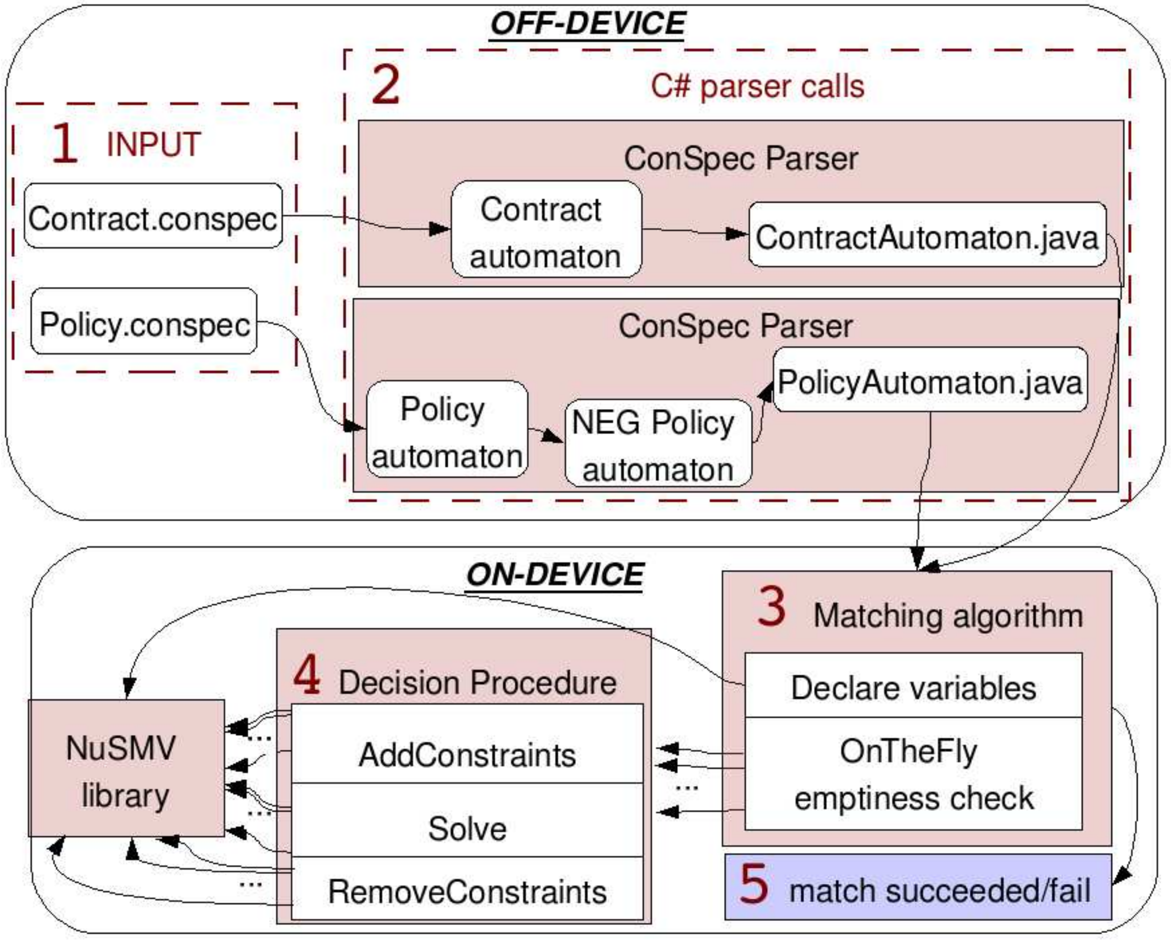
- Finding counterexamples faster:
  - combine algorithm b on Nested DFS [Schwoon & Esparza]
  - decision procedure for SMT [Cimatti et al.]
  - Explicit on-the-fly parallel visit of NegPolicy and Contract
- Algorithm
  - Start a DFS procedure over the initial pair of states (nP&C)
  - IF DP sat on pair of outgoing edges => visit successor pair
  - IF an accepting pair of states in $\mathcal{AMT}$ is reached:

    IF one is error state of neg policy => security violation

    ELSE start a new DFS

    - *IF cycle back => availability violation*.

Contract-Policy Architecture



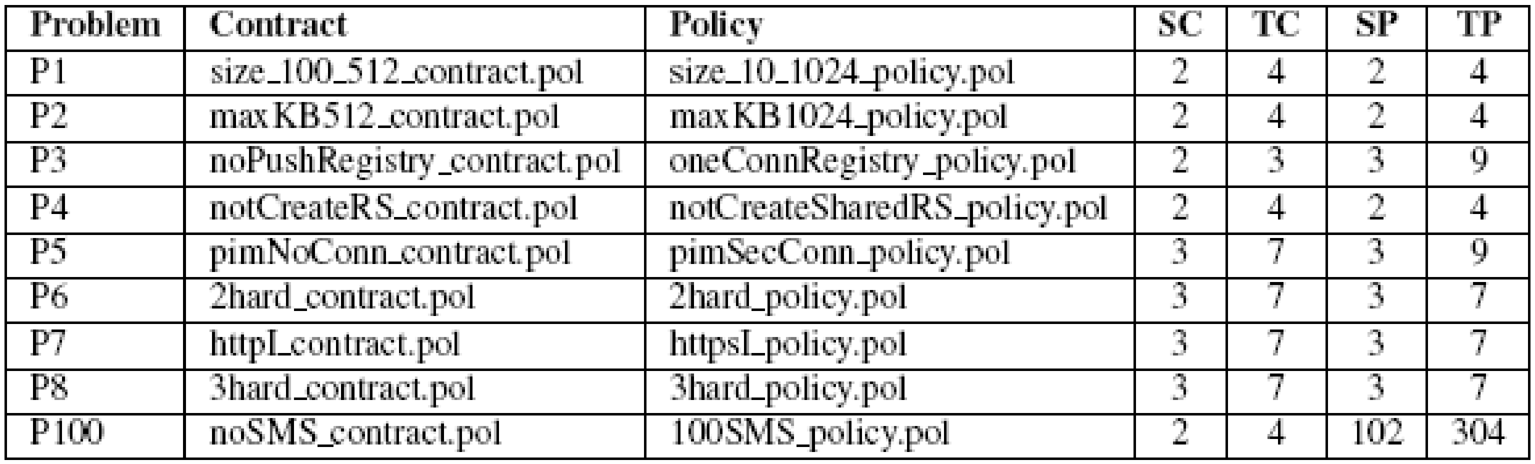
Design Decisions

- **One vs Many**
  - 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^cond^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 ^ 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).
- Current implementation uses PRIORITY MC ONE INSTANCE CACHING MC configuration.
  - PRIORITY MC: the nature of rules in policies which is priority OR
  - MUTEX SOLVER does not allow empty methods such as ¬mi ^ ¬mj which is possible in the matching algorithm.
  - ONE INSTANCE: garbage collection problem.
  - CACHING MC: save calls to solver for the already solved rules.

Università degli Studi di Trento

Experiments on Desktop and on Device

- Implemented on a Java platform for 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)
- Some experimental results on .NET implementation for a Mobile platform i.e. ported to HTC P3600
  - 3G PDA phone with ROM 128MB, RAM 64MB, SamsungR SC32442A processor 400MHz



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	httpL_contract.pol	httpsL_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

Problems Suit