Rule Based Systems and the Intersection of Formal Methods and Testing

Rick Kuhn, Vincent Hu, David Ferraiolo, Raghu Kacker, Dylan Yaga, and Yu Lei*

National Institute of Standards and Technology
{kuhn,vhu,david.ferraiolo,raghu.kacker,dylan.yaga}@nist.gov

**University of Texas at Arlington*
ylei@uta.edu

Problem: how to test implementation correctness against access control rules
- Large number of rules
- Large number of attributes/variables
- 2-value result – grant/deny

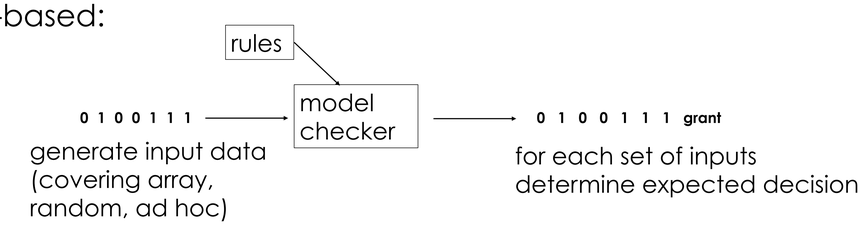
Conventional solution:
- "use cases" verifying important or common situations
- ad hoc
- often not very thorough

model-based testing solution:
- rules → formal model → model checker or other → test cases
- usually based on fault model; mutation testing
- may use some notion of policy coverage
- ACPT
- Margrave
- others

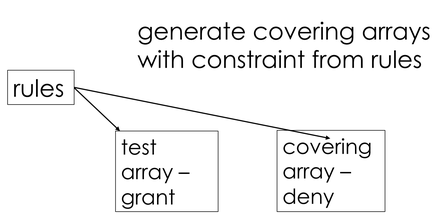
Pseudo-exhaustive testing solution using covering arrays:
- determine dependencies
- partition according to these dependencies
- exhaustively test the inputs on which an output is dependent
- for access control:
  - convert rule antecedents to k-DNF form, producing sets of k or fewer attributes that will produce a "grant" decision
  - generate separate k-way covering arrays for combinations that should produce "grant" and "deny"

COMPARISON

model-based:



our approach:



Positive Testing (The Easy Part)

- want to ensure that any set of appropriate attributes produces *grant* decision
- test set Gtest: every test should produce a response of *grant*.
- for any input where some combination of *k* input values matches a *grant* condition, a decision of *grant* is returned.
- Construct test set Gtest with one test for each term of *R* as follows:
- $Gtest_i = T_i \bigwedge_{j \neq i} \sim T_j$
- one test for each term in access control rule antecedents, with constraint removing any combination that would mask a fault
- example: testing that *ab* results in *grant*, for *ab + cd →grant*, enforce constraint ~(cd)

Negative Testing (The Hard Part)

- test set Dtest = covering array of strength *k*, for the set of attributes included in *R*
- constraints specified by ~*R*
- ensures that all deny-producing conjunctions of attributes tested
- masking is not a consideration – because of problem structure
- *deny* is issued only after all *grant* conditions have been evaluated
- masking of one combination by another can only occur for Dtest when a test produces a response of *grant*
- if so, an error has been discovered; repair and run test set again

Example: Why It Works

- rule structure:
  - R_1 → *grant*
  - R_2 → *grant*
  - …
  - R_m → *grant*
  - *else* → *deny*

	a	b	c	d	e
1	0	0	0	0	0
2	0	0	1	1	1
3	0	1	1	0	0
4	1	0	0	1	0
5	1	0	1	1	0
6	1	1	0	0	1
7	1	1	1	1	1
8	0	0	1	0	1
9	1	1	0	1	0
10	0	0	0	1	1
11	1	0	0	0	0
12	0	1	1	1	0
13	1	0	0	0	1
14	0	1	1	0	1

covering array containing all t-way tuples except for those in a *grant* condition

Number of Tests

- for positive tests, Gtest: one test for each term in the rule set, for for m rules with p terms each , mp
- for negative tests, Dtest: one covering array per rule, where each attribute in the rule is a factor

k	v	n	m	N tests	#Gtest	#Dtest
3	2	50	20	36	80	720
			50		200	1800
		100	20	45	80	900
			50		200	2250
	4	50	20	306	80	6120
			50		200	15300
		100	20	378	80	7560
			50		200	18900
	6	50	20	1041	80	20820
			50		200	52050
		100	20	1298	80	25960
			50		200	64900
4	2	50	20	98	80	1960
			50		200	4900
		100	20	125	80	2500
			50		200	6250
	4	50	20	1821	80	36420
			50		200	91050
		100	20	2337	80	46740
			50		200	116850
	6	50	20	9393	80	187860
			50		200	469650
		100	20	12085	80	241700
			50		200	604250