National Institute of Standards and Technology

Combinatorial Testing and its Applications

Rick Kuhn

National Institute of
Standards and Technology
Gaithersburg, MD

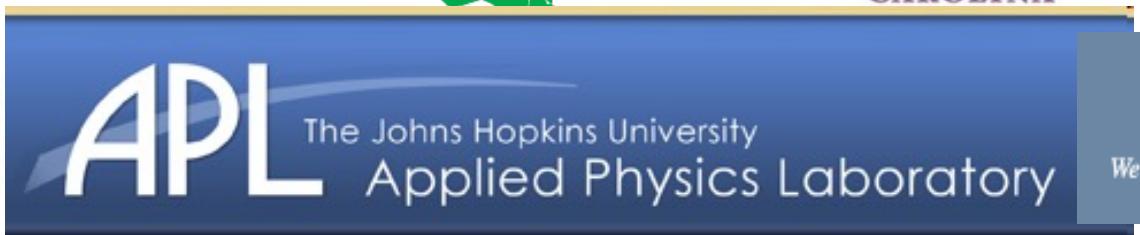
CS/SE 6367

Univ of Texas Dallas. June 15, 2023

What is NIST and why are we doing this?

- US Government agency, whose mission is to support US industry through developing better measurement and test methods
- 3,000 scientists, engineers, and staff including 4 Nobel laureates
- **Project goal – <u>improve cost-benefit ratio for testing</u>**

UNIVERSITY OF TEXAS ARLINGTON



Background

Why combinatorial testing? - examples

- Cooperative R&D Agreement w/ Lockheed Martin
  - 2.5 year study, 8 Lockheed Martin pilot projects in aerospace software
  - Results: save 20% of test costs; increase test coverage by 20% to 50%
- Rockwell Collins applied NIST method and tools on testing to FAA life-critical standards
  - Found practical for industrial use
  - Enormous cost reduction

Average software: testing typically 50% of total dev cost
Civil aviation: testing >85% of total dev cost (NASA rpt)

Applications

Software testing – primary application of these methods

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

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)

So this is a **2-way interaction**

=> testing all pairs of values can find this fault

How are interaction faults distributed?

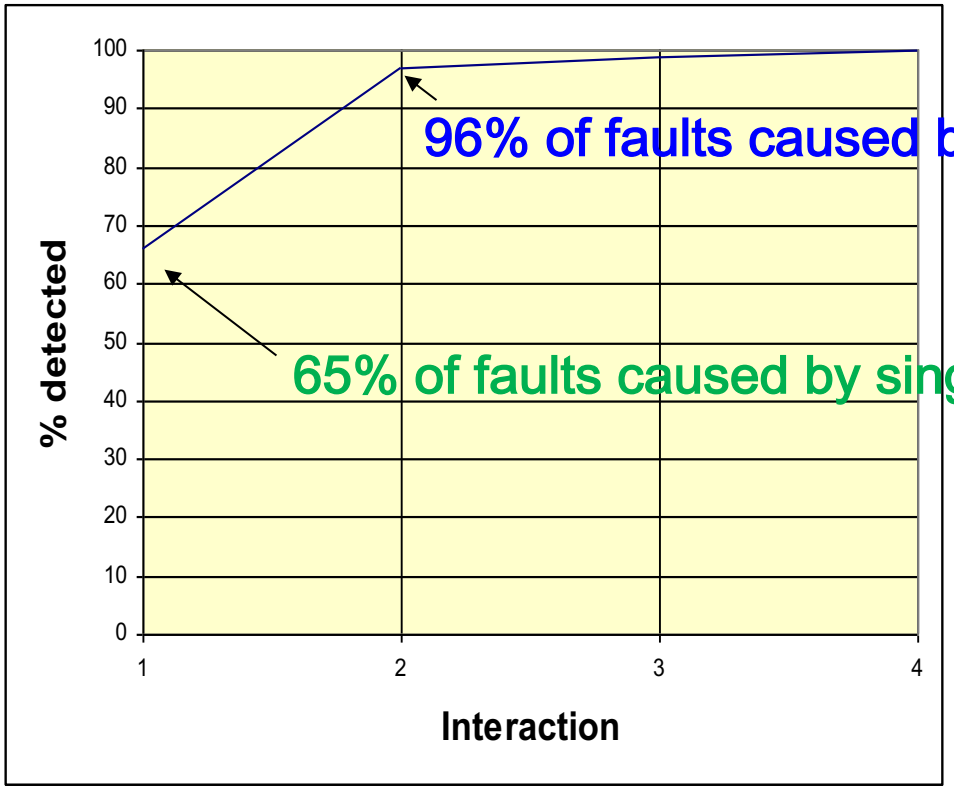
- Interactions e.g., failure occurs if

  pressure < 10 (1-way interaction)

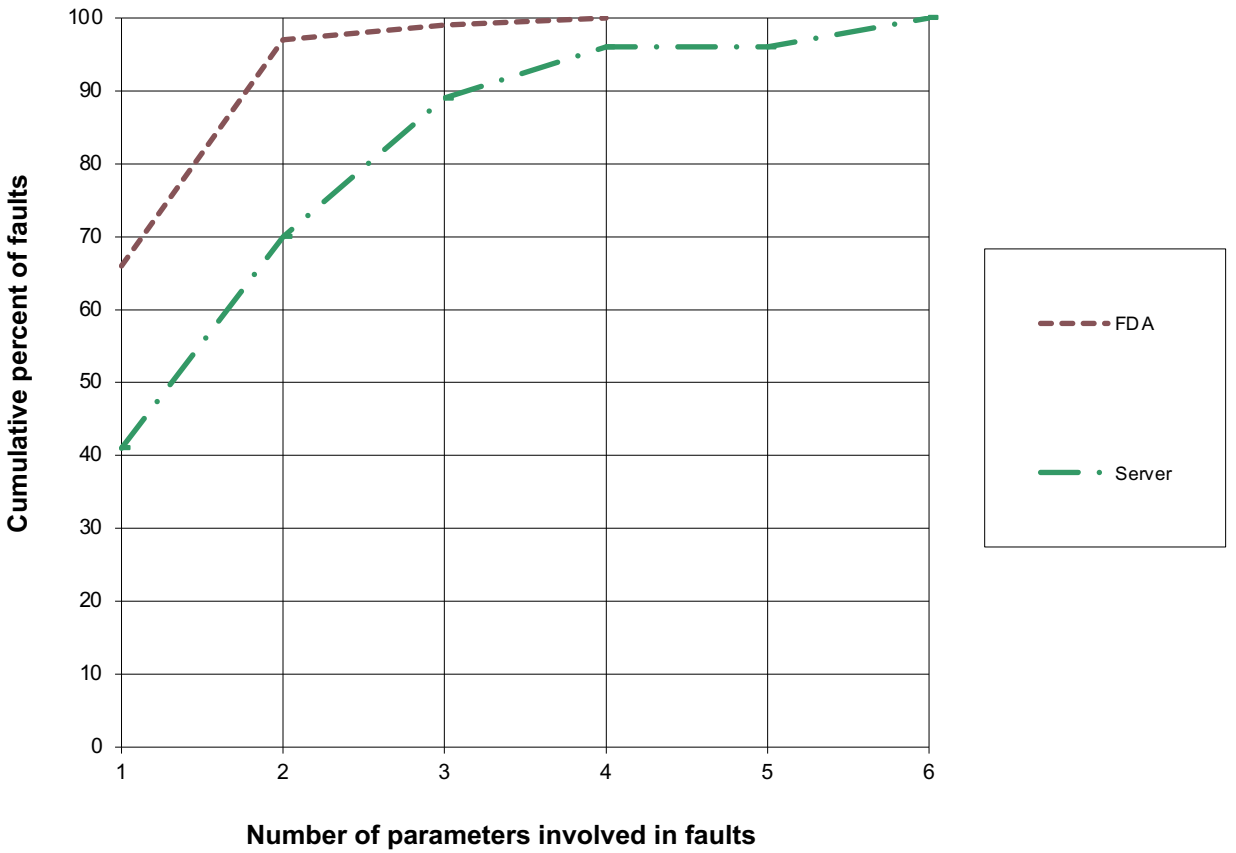
  pressure < 10 & volume > 300 (2-way interaction)

  pressure < 10 & volume > 300 & velocity = 5 (3-way interaction)

- Surprisingly, no one had looked at interactions > 2-way before



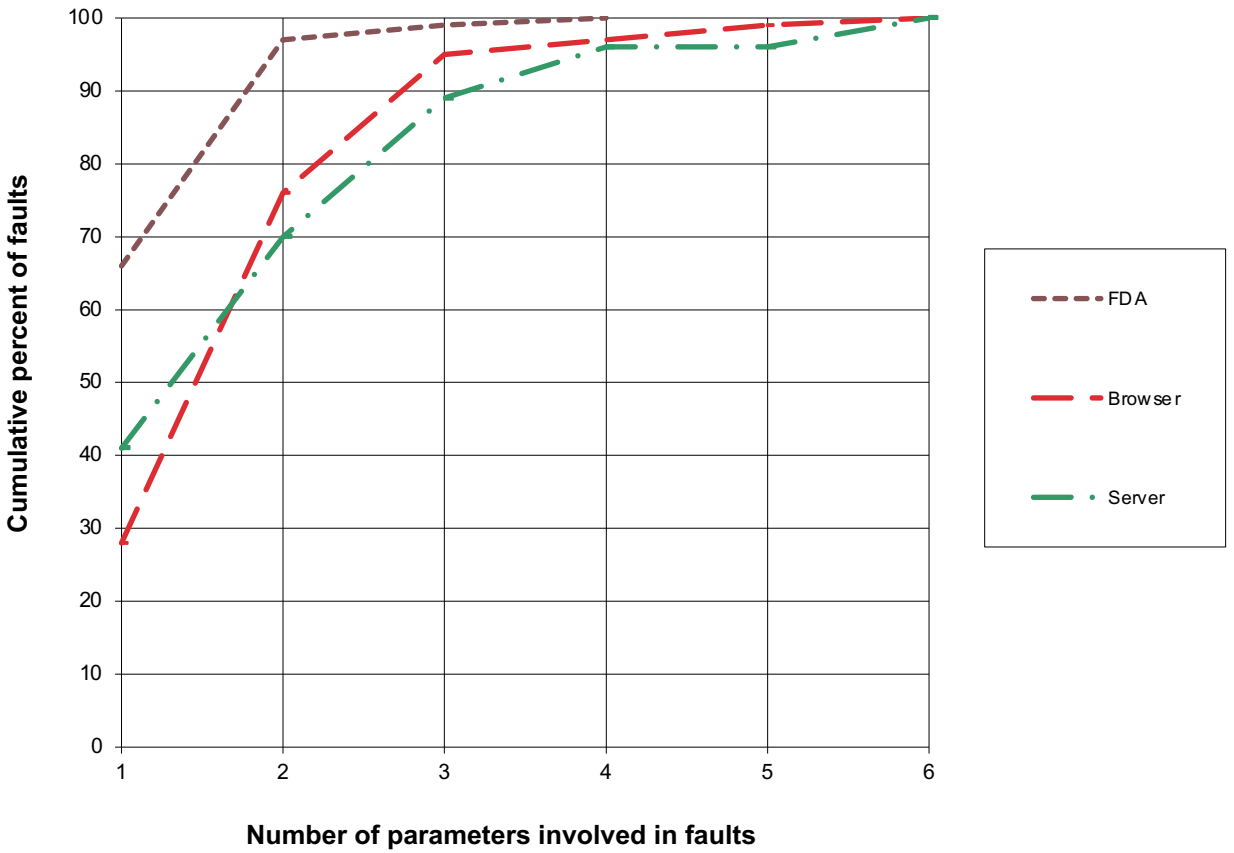
Server



These faults more complex than medical device software!!

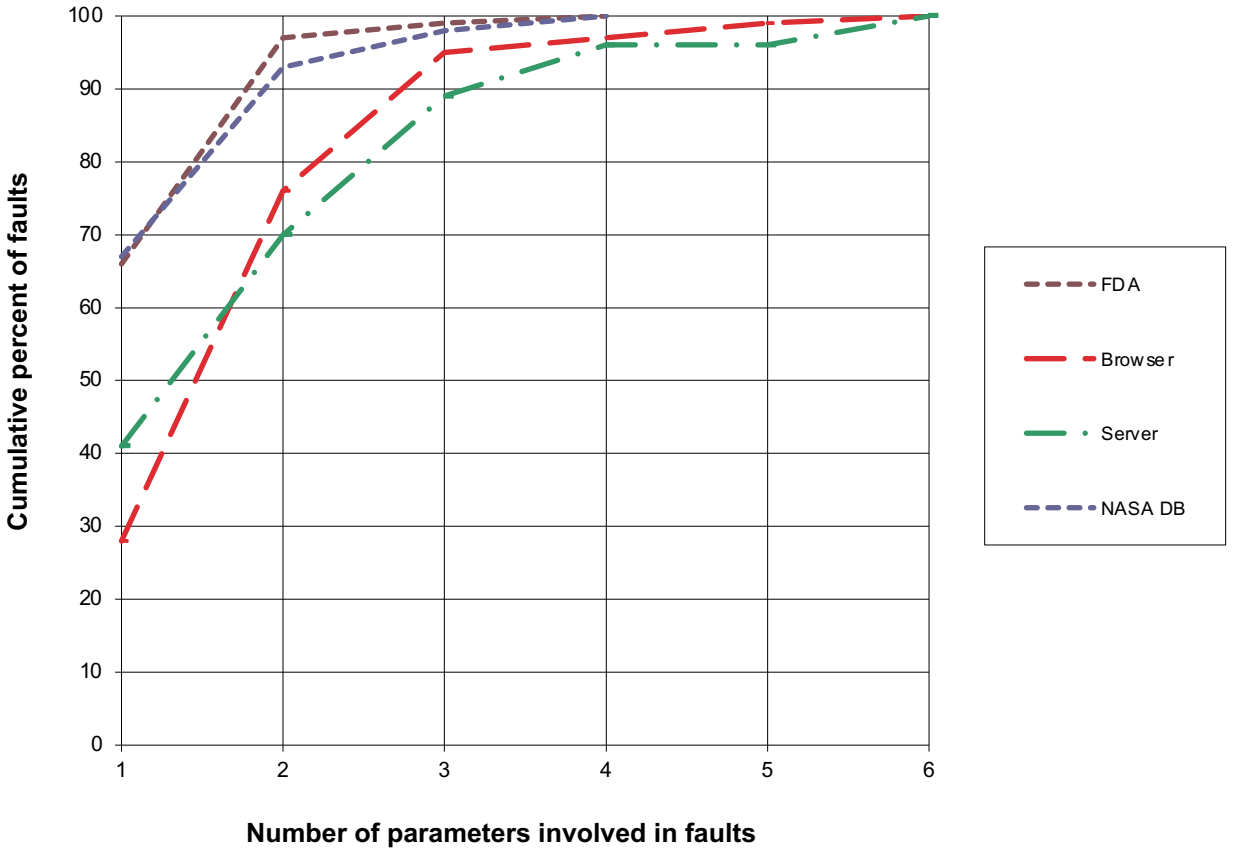
Why?

Browser



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

NASA distributed database

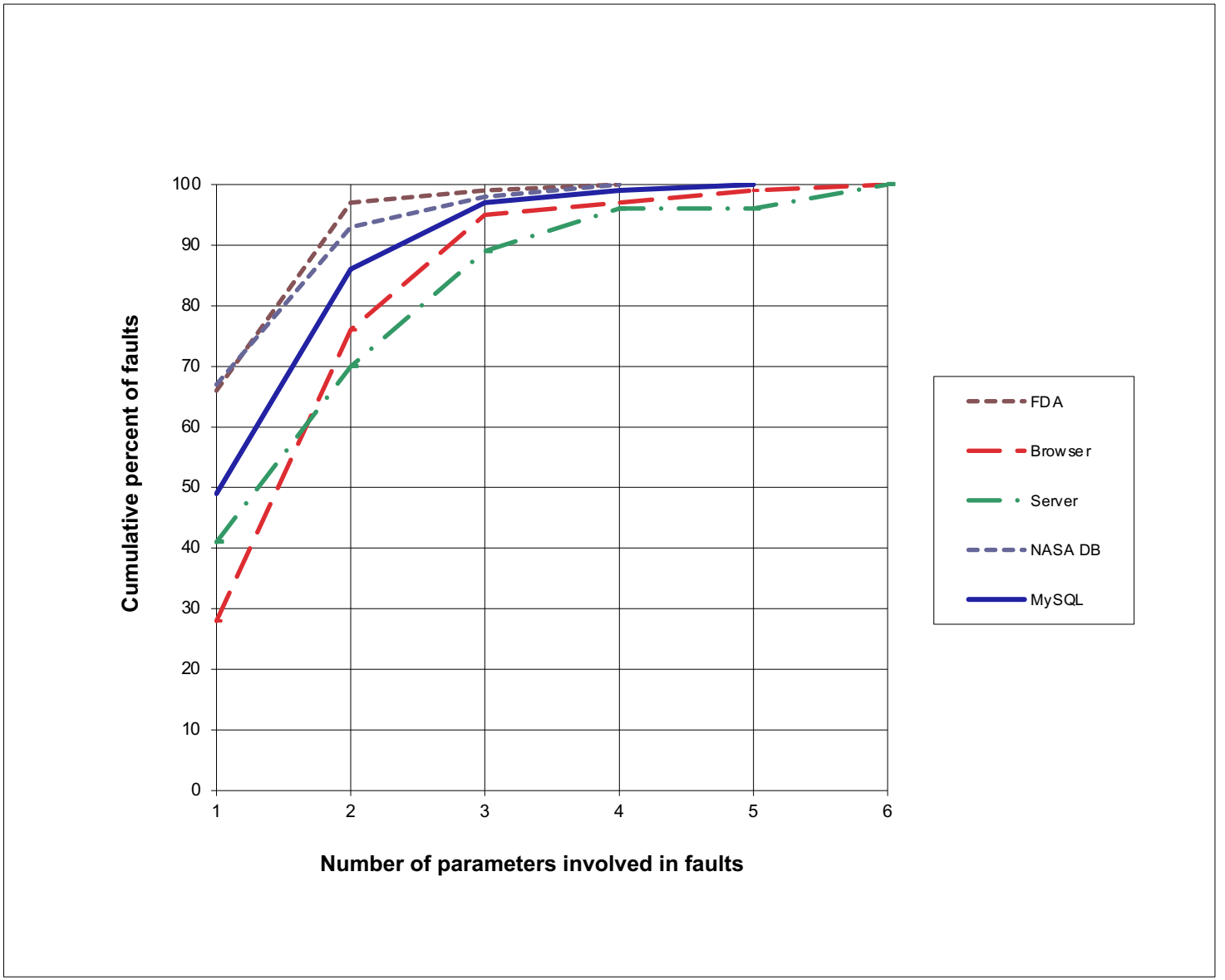


Note: initial testing

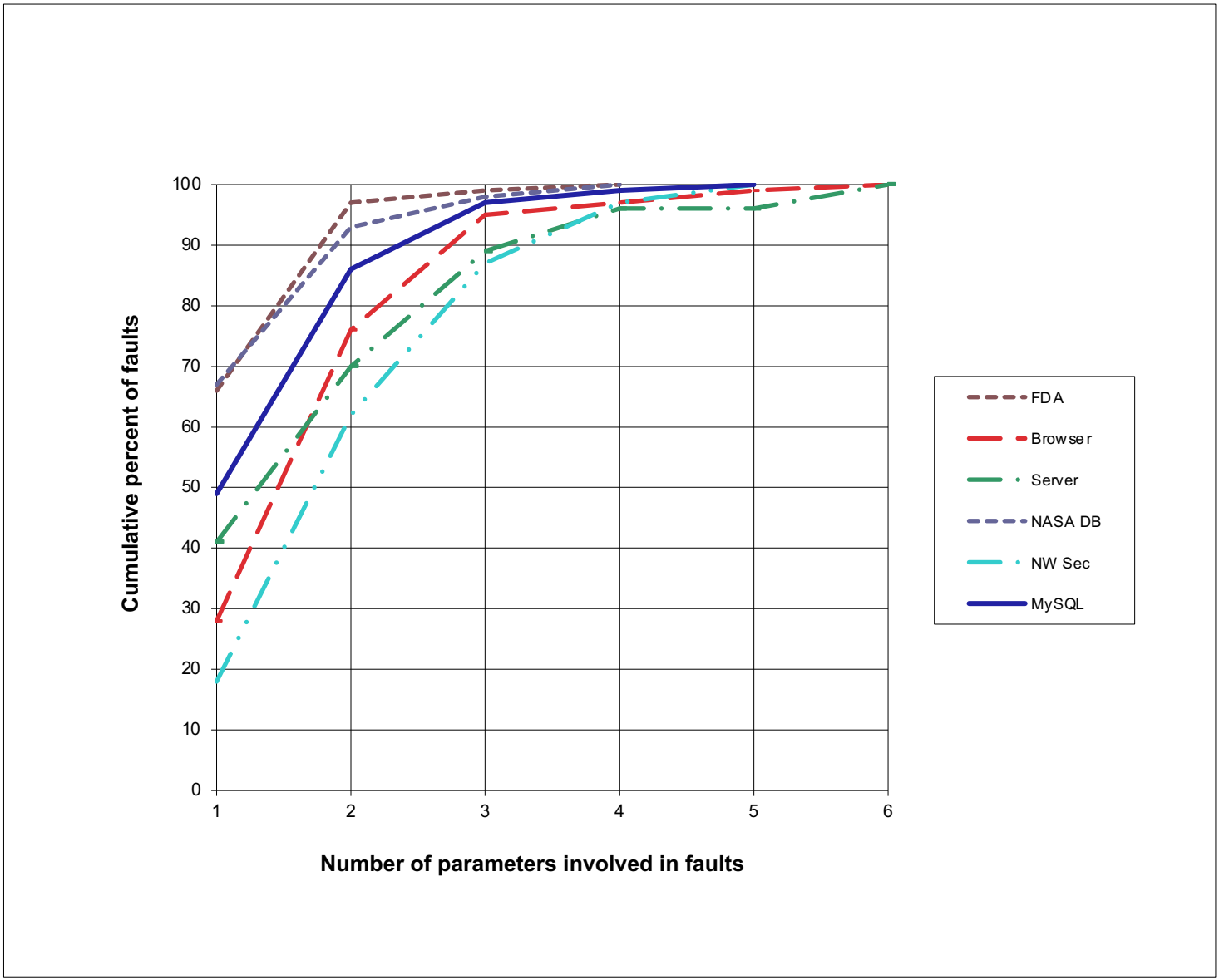
but ….

Fault profile better than medical devices!

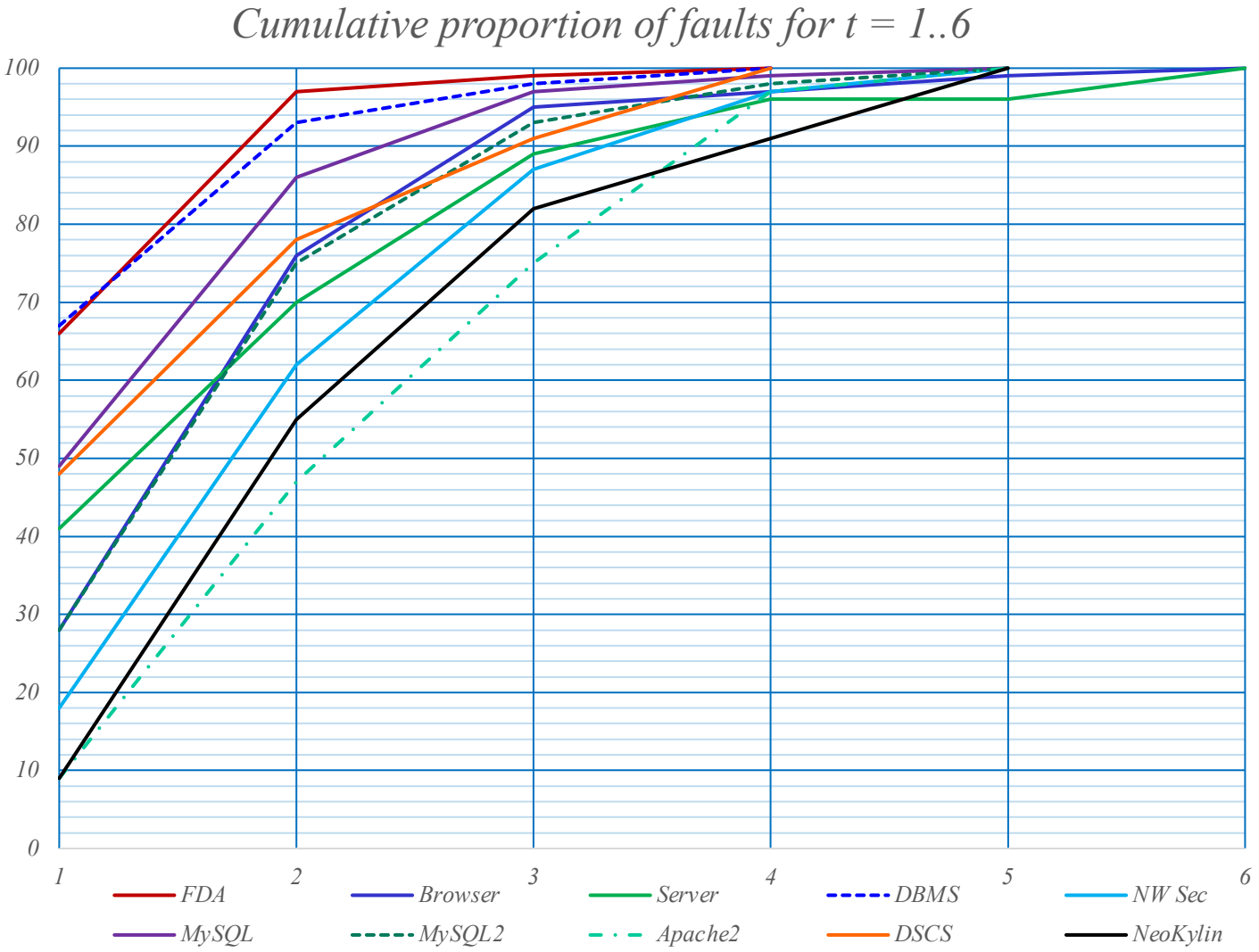
MySQL



TCP/IP

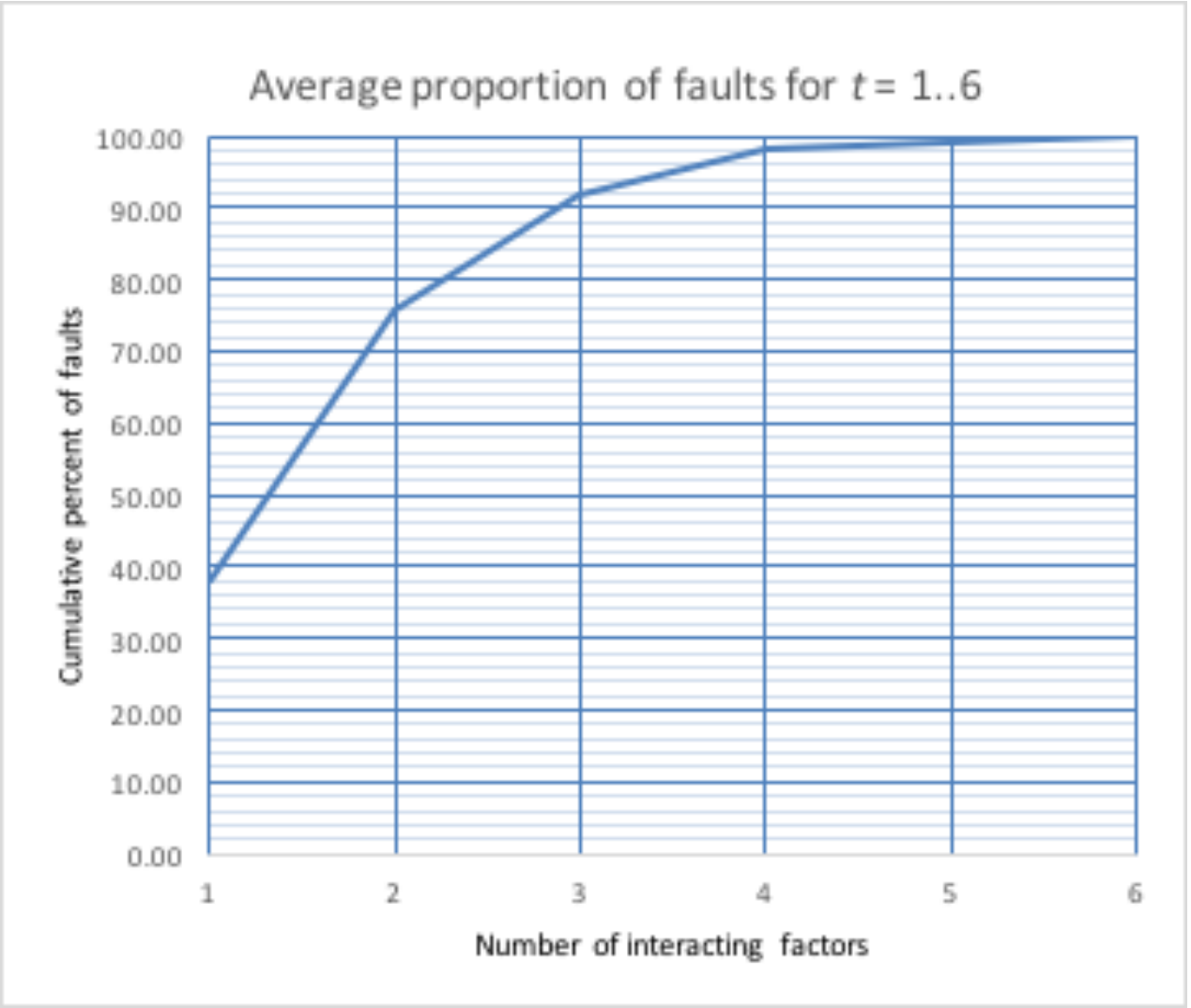


Various domains collected

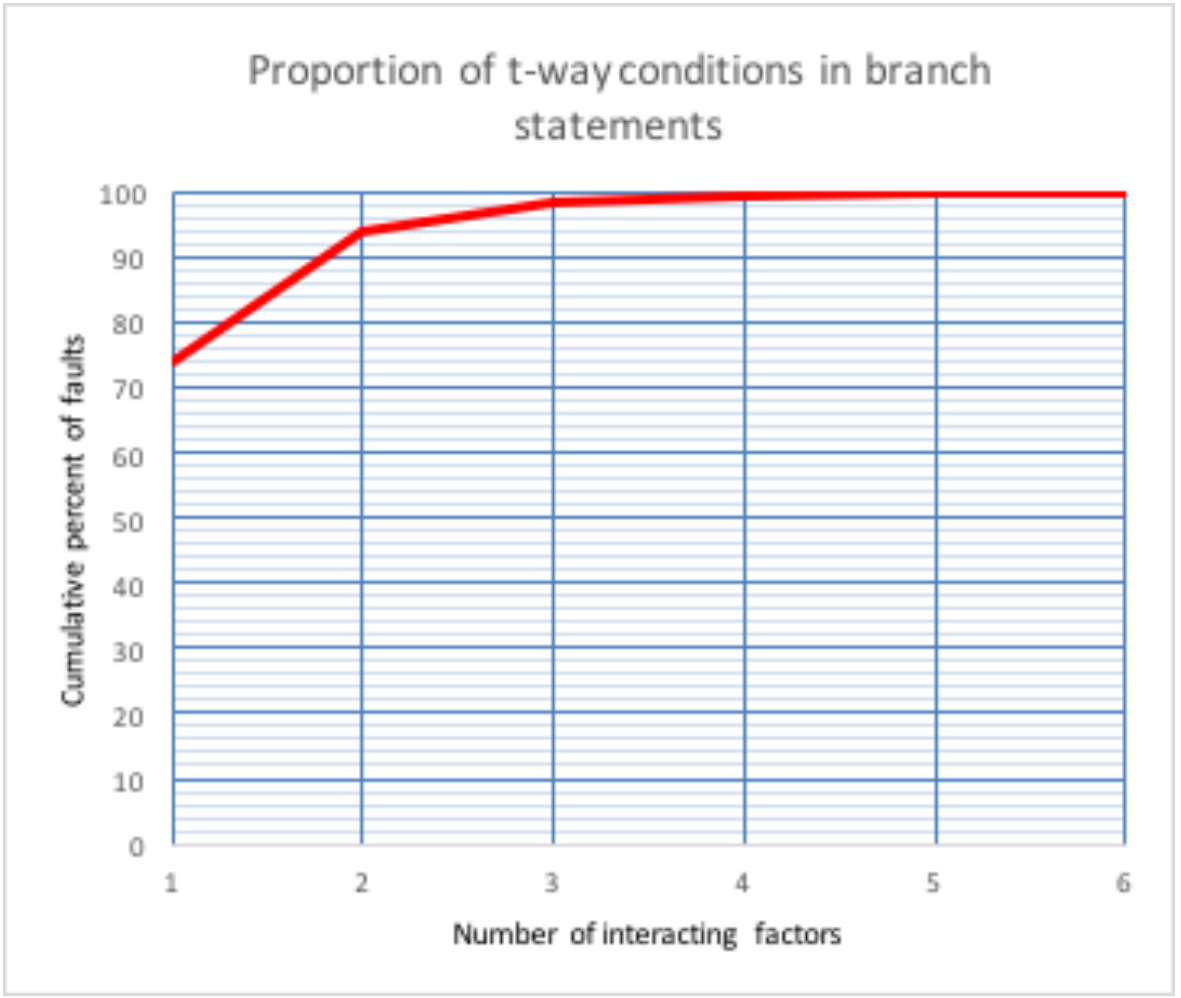


- Number of factors involved in failures is small
- No failure involving more than 6 variables has been seen

Average (unweighted)

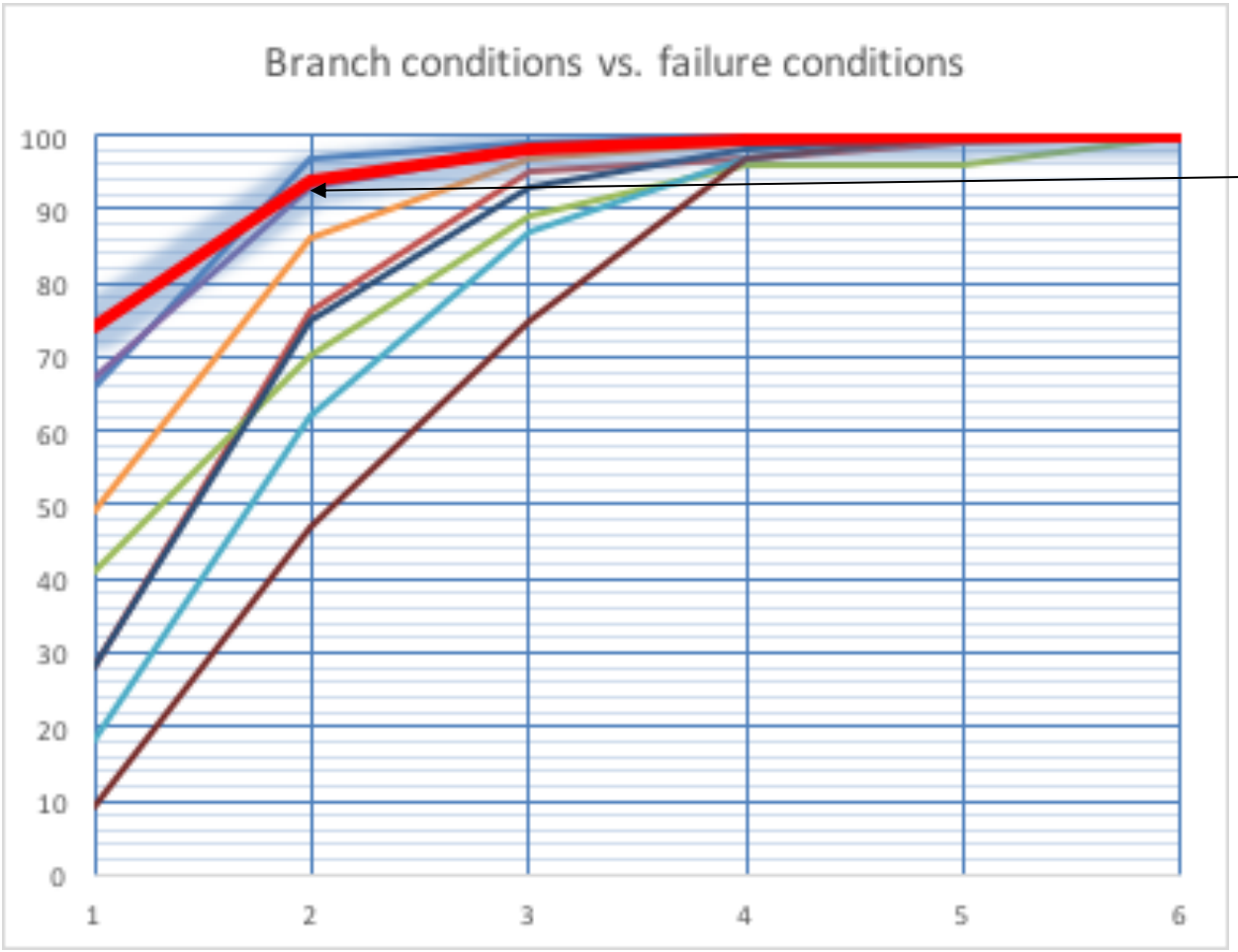


What causes this distribution?



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

Comparing with Failure Data



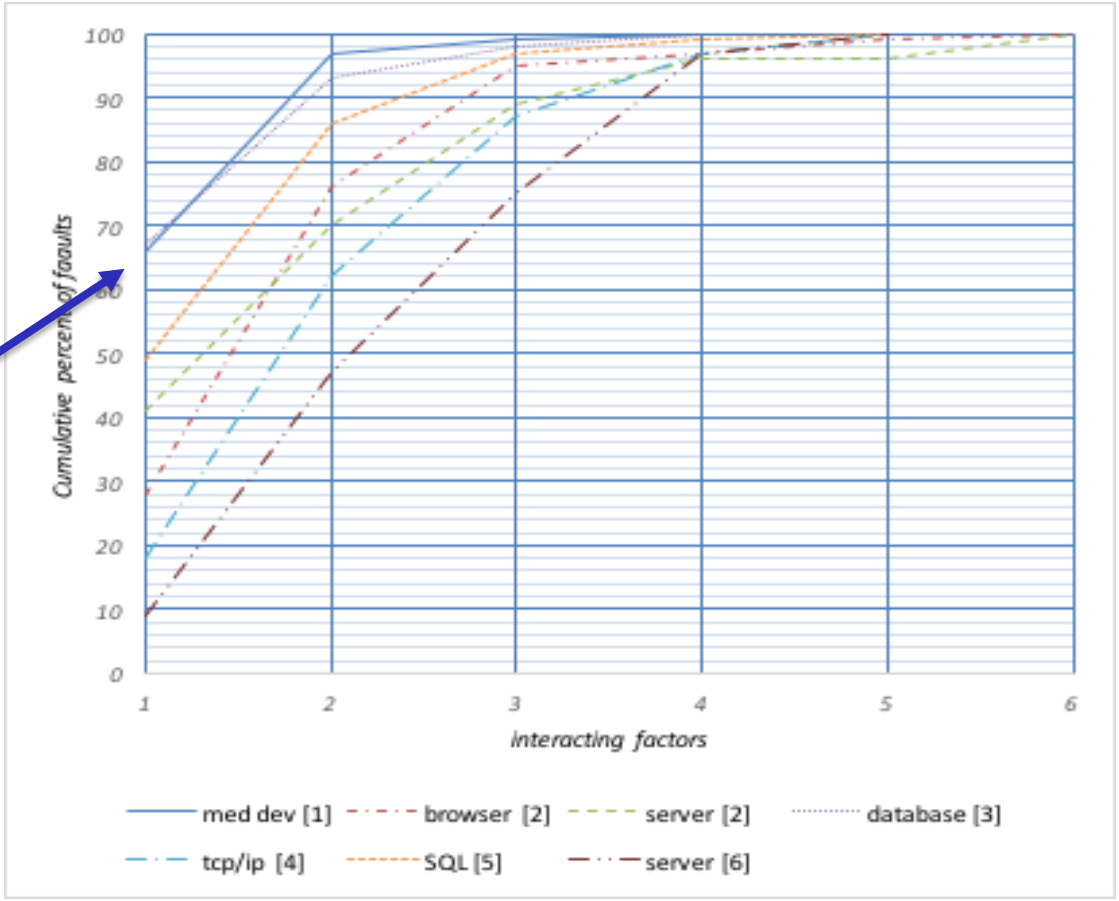
Branch statements

- 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

Distribution of failures by number of interacting variables

Interaction rule: most failures caused by one factor or two interacting; progressively fewer by ≥ 3 variables interacting

- No failures involving more than 6 variables among these
- Untested (database) or smaller user base applications (med devices) have simpler faults than heavily used applications (browser, server, SQL)



Why does this distribution occur?

Intuitively, simpler faults should be more common than complex faults; should take longer to find complex faults

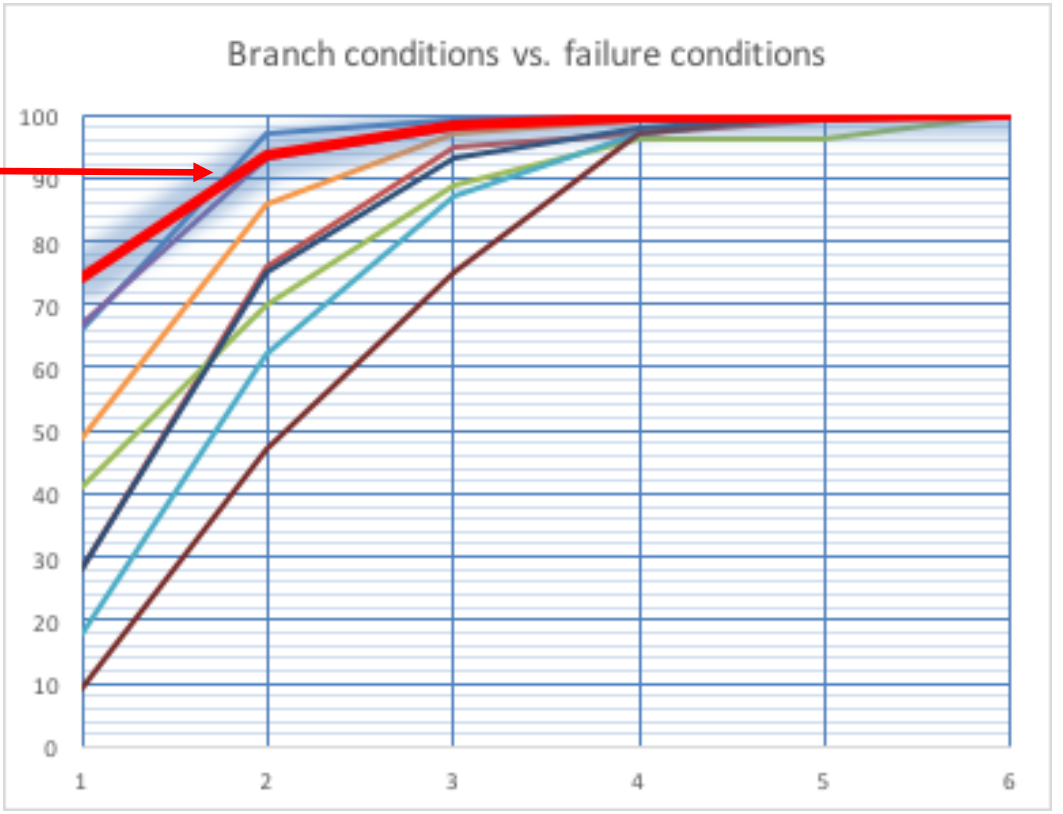
Can we develop a quantitative model?

- Start with two assumptions:
  - *t*-way faults occur in proportion to *t*-way conditions in code
  - *t*-way faults are removed in proportion to *t*-way combinations in inputs

- Do these assumptions reproduce the empirical data?

Branch condition data for *t*-way conditions in code

- 7,685 predicates from four avionics applications
- 400,811 predicates from 6.03 million lines of code in 63 Java open source applications
- does not consider effects of nesting



If *t*-way faults are removed in proportion to *t*-way combinations in inputs:

What will that look like over time as faults are removed?

t:	1	2	3	4	5	6	7	8
Avionics	74.1	19.6	4.5	1.2	.3	.1	.1	.1
Java	88.5	9.5	1.4	.4	.1	.1	0	0

Quantify the model parameters

Each set of inputs includes $C(n,t)$ combinations at each level of t, for n variables

- total number of combination settings is $v^t \times C(n,t)$, so each test or input set can cover at most $1/v^t$ of the total number of settings
- (t+1)-way combinations covered at rate of $v^{-(t+1)}/v^{-t} = 1/v$ of the proportion of t-way combinations
- if 1-way faults are removed at some rate r for some number of test sets, then the proportion remaining after k sets will be $(1-r)^k$

Now what happens with 2-way, 3-way, etc. faults?

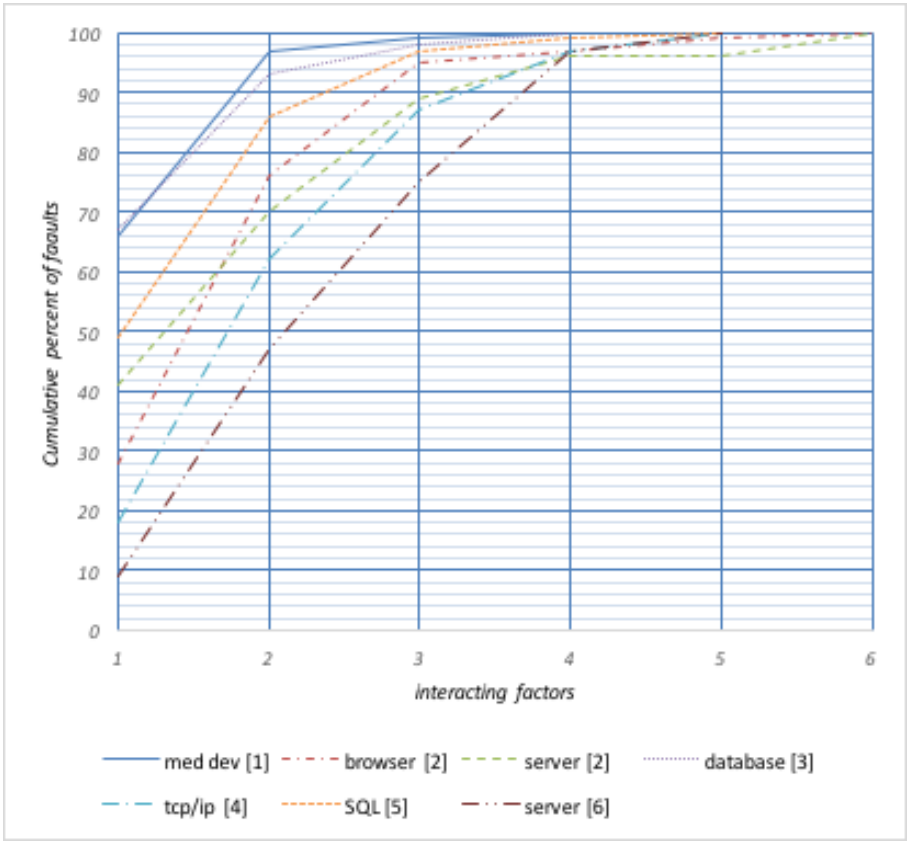
Discovery of a t-way fault depends on the presence of t-way combinations in input

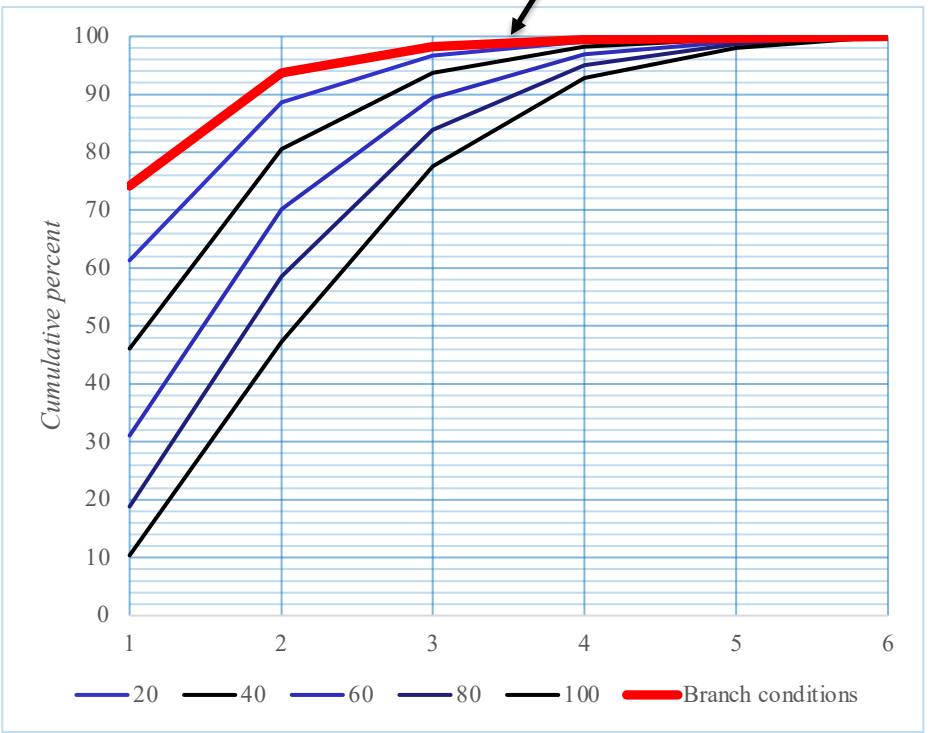
- (t+1)-way combinations 1/v of t-way combinations
- so 1-way fault discovery rate r will be reduced by this proportion, or r/v for 2-way, r/v^2 for 3-way, etc.
- minimum value of v is 2, and Boolean values are common
- so proportion of remaining t-way faults after k test sets $= (1 - r/2^{t-1})^k$

How does this distribution evolve as usage or number of test sets, k, increases?

Fault distribution as testing progresses

for testing cycles of *r*=.05; *k* = 20..100; starting from branch conditions;
curve moves down and to the right; close to empirical data

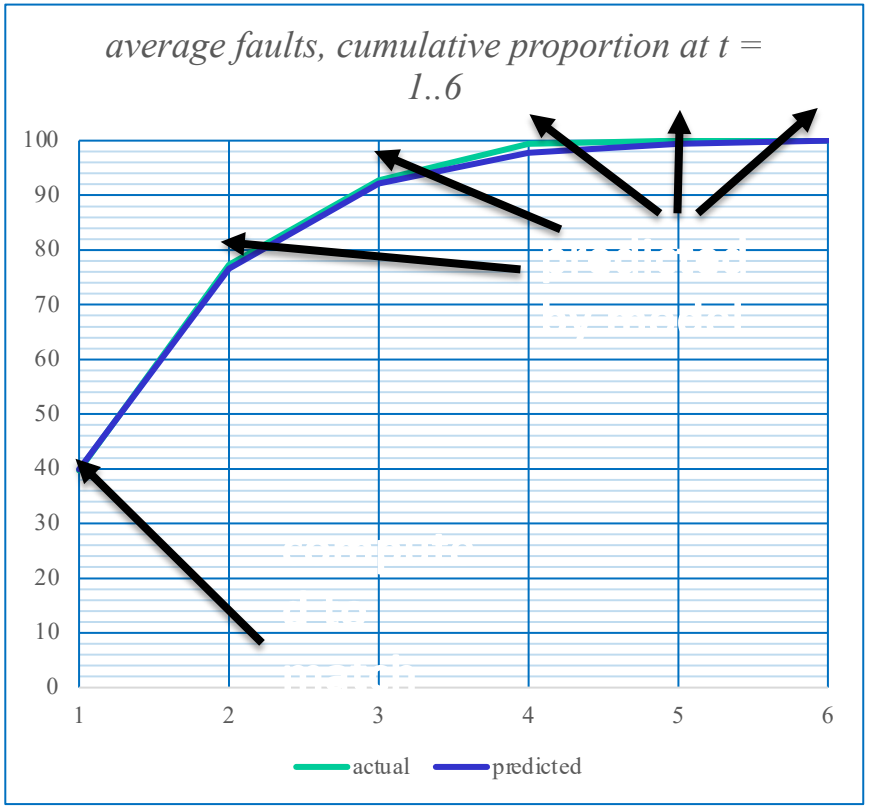




Actual vs. model - average

- Comparing model with average of reported *t*-way faults
- Determined k such that actual at *t* = 1 matches model at *t* = 1
- Values for *t* = 2 ..6 predicted by model
- Close match suggests assumptions are appropriate
- note this is approximately a Pareto distribution with $\alpha = 1.5$

t =	1	2	3	4	5	6
actual	39.7	77.3	92.8	99.4	100	100
predicted	39.9	76.6	92.2	97.8	99.4	100

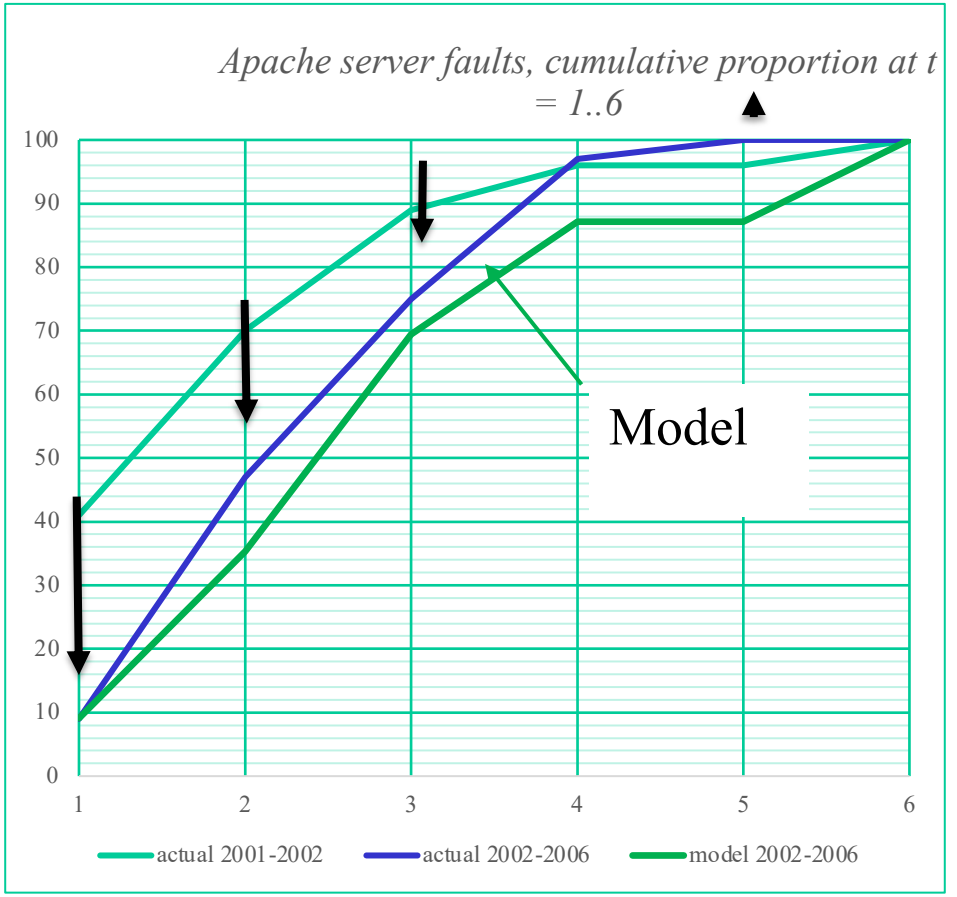


Actual vs. model – individual system

- Evolution of Apache server *t*-way faults for period 1 (2001-2002) through period 2 (2002-2006)

- Determined *k* tests such that actual for period 2 at *t* = 1 matches model at *t* = 1
- Values for *t* = 2 ..6 predicted by model

	1-way	2-way	3-way	4-way	5-way	6-way
Rpt - 2002	41	29	19	7	0	4
Rpt - 2006	9	38	28	22	3	0
k=54 test sets	9.1	26.2	34.1	17.8	0	13



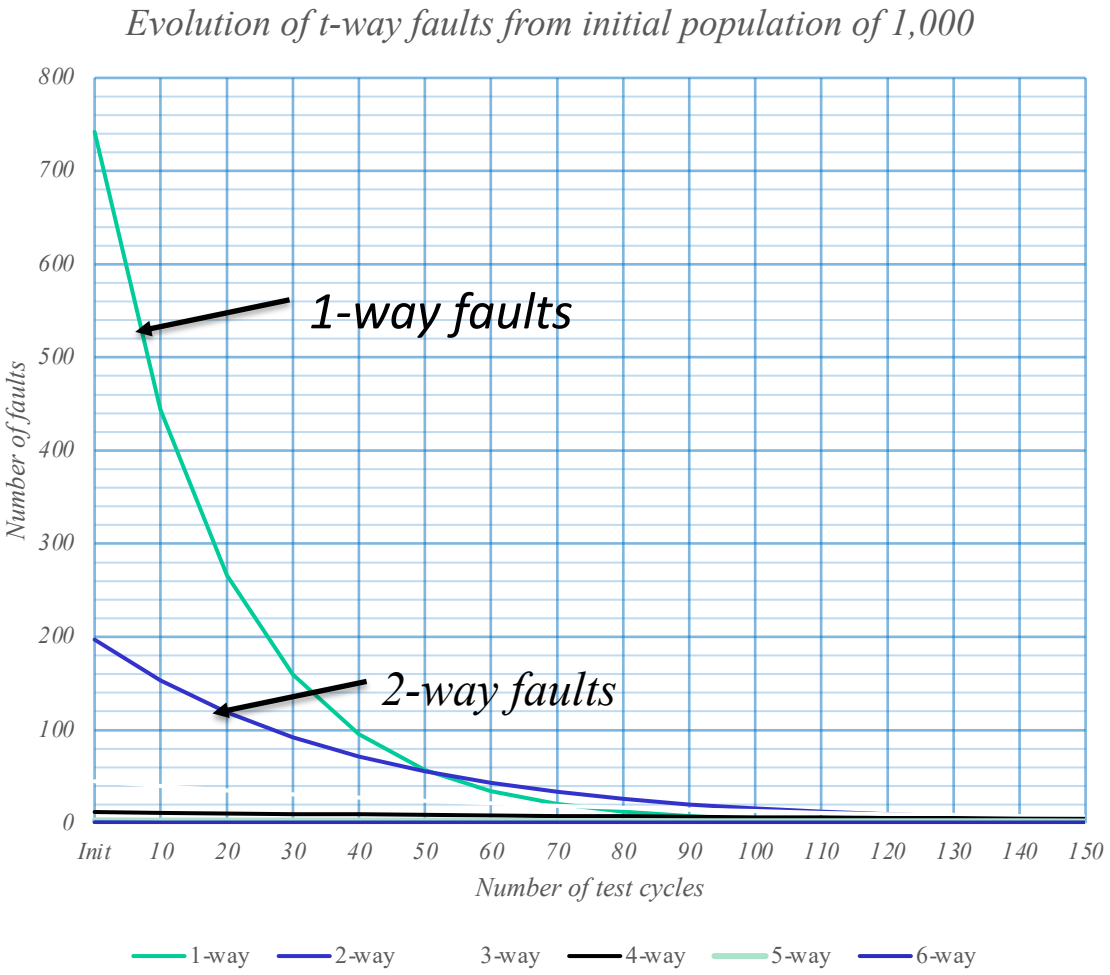
Model evolution of number of faults at *t*=1..6

- initial population of 1,000 faults
- initial distribution matches branch condition distribution
- simple faults decline much faster than more complex faults, as seen in practice

t-way faults = $F_t(1 - r/v^{t-1})^k$

for
v = 2
r = .05
k = 0..20

t	F_t %
1	74.1
2	19.6
3	4.5
4	1.2
5	0.3
6	0.1



Relationship with Reliability Growth Models

Goel-Okumoto, where

μ(*n*) = expected number of defects detected at time *n*:

$$\mu(n) = a(1 - e^{-bn})$$

for

a = initial number of defects

n = execution time

b = rate at which failure rate decreases

assumes: cumulative number of failures follows a Poisson process

How is this related to our model?

t-way fault evolution model: proportion of remaining *t*-way faults after *k* test sets

$\approx (1 - r/2^{t-1})^k$

- so if μ(*n*) is expected number of defects detected at time *n*
- then $\mu(n) = a(1 - (1 - r/2^{t-1})^k)$ in our model

- so for single-factor faults this reduces to

  $\mu(n) = a(1 - (1 - r)^k)$

  $\approx a(1 - e^{-kr})$ which is <u>equivalent to Goel-Okumoto</u>

- with $r_t = r/2^{t-1}$ we have basic exponential model for each level of *t*
- *so it reproduces standard reliability model without initial assumption of Poisson process*

Making this Knowledge Useful

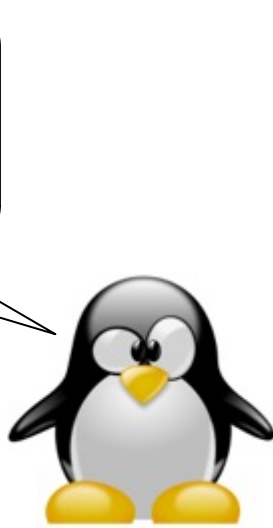
How does this knowledge help?

Interaction rule: 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, . . .)



Testing Interactions: Design of Experiments

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
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	3
9	3	3	2

Each combination occurs same number of times, usually once.

Example: P1, P2 = 1,2

Orthogonal Arrays for Software 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)

Results good, but not great.

What's different about software?

National Institute of Standards and Technology

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

How do these differences affect interaction testing for software?

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

Key differences

orthogonal arrays:

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

covering arrays:

- Combinations occur at least once
- Always possible to find for a particular configuration
- Size always $\leq$ orthogonal array

NIST National Institute of Standards and Technology

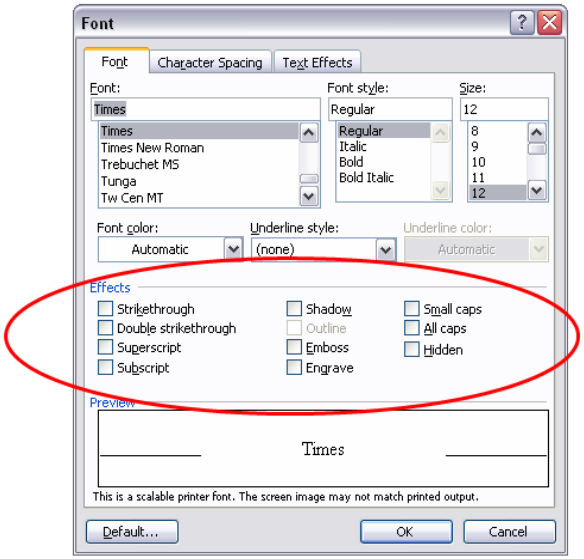
A covering array of 13 tests

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

Each row is a test:

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

Each column is a parameter:



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

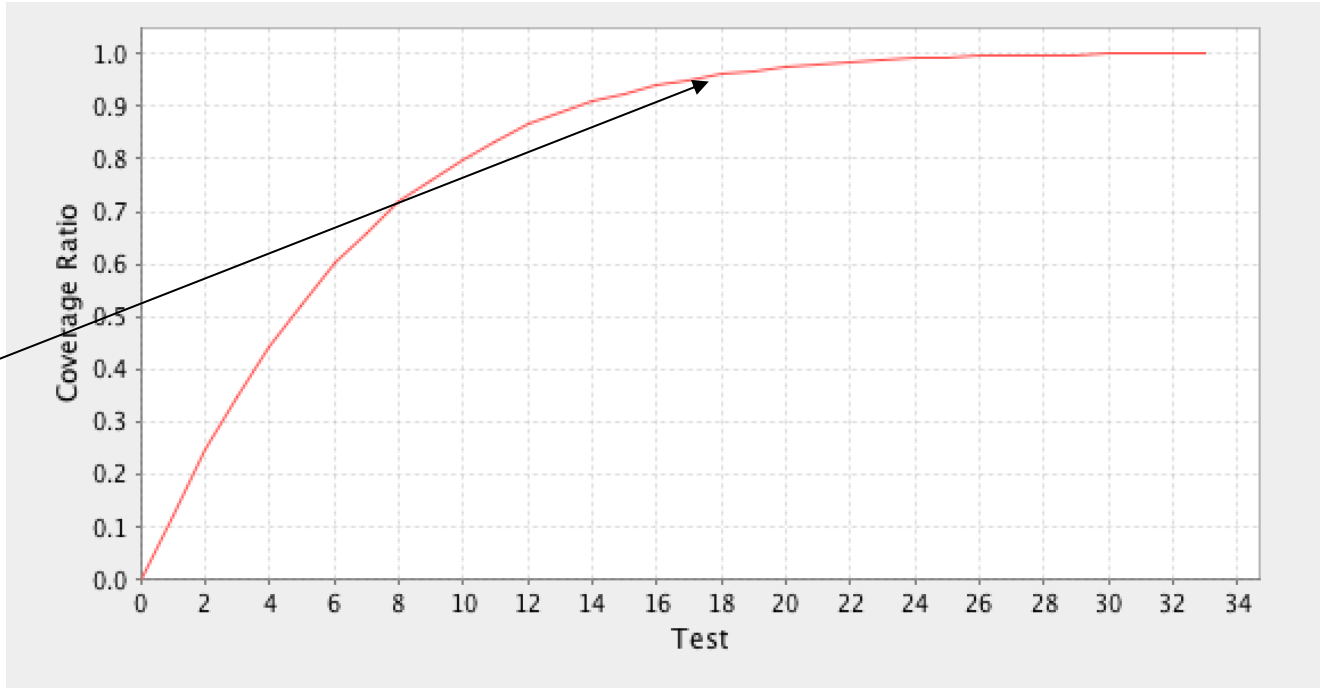
NIST National Institute of Standards and Technology

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 logarithmically with the number of parameters
  => even very large test problems are OK (e.g., 200 parameters)
- Bad news: increase exponentially with interaction strength t
  => select small number of representative values (but we always have to do this for any kind of testing)

However:

- coverage increases rapidly
- for 30 boolean variables
- 33 tests to cover all 3-way combinations
- but only 18 tests to cover about 95% of 3-way combinations



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 configurations – combinations of settings

- 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

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

NIST National Institute of Standards and Technology

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

Solving the oracle problem

- Problem: How do we determine the expected results for a set of inputs?
- Three approaches using combinatorial testing

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?
  - **Crash testing** server or other code to ensure it does not crash for any test input (like 'fuzz testing')
    - 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

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

1 - Embedded Assertions

Assertions check properties of expected result:

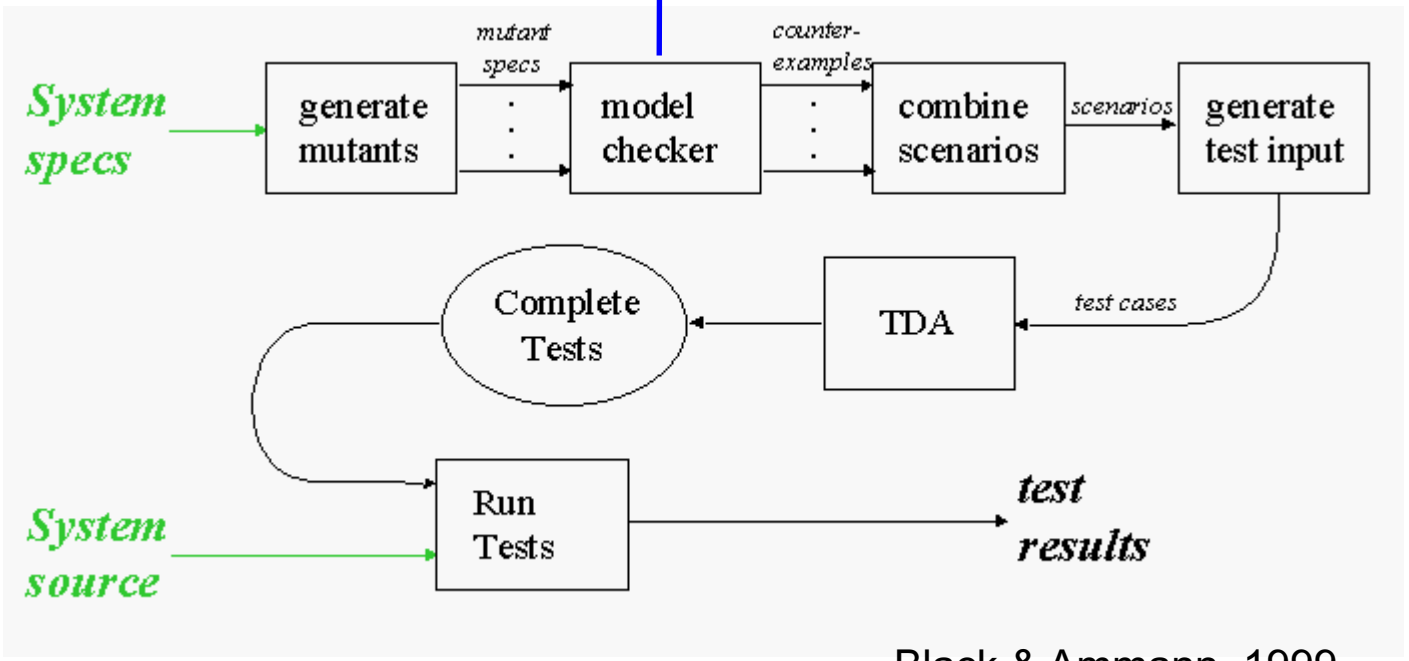
```
ensures balance  == \old(balance) - amount
&&  \result == balance;
```

•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

2 - model checking to produce tests



- Model-checker test production:
if assertion is not true, then a counterexample is generated.

- This can be converted to a test case.

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 4-value, two 10-value
  - All flaws found with 5-way coverage
  - Thousands of tests - generated by model checker in a few minutes

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

NIST National Institute of Standards and Technology

The usual logic operators,plus temporal:

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

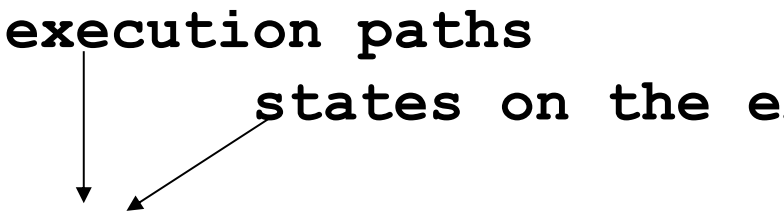
E φ - Exists: φ holds on some paths starting from the current state.

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]




```
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 -> AX !(1))`**
3. **`AG(v1 & v2 & ... & vk -> 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 -> 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!

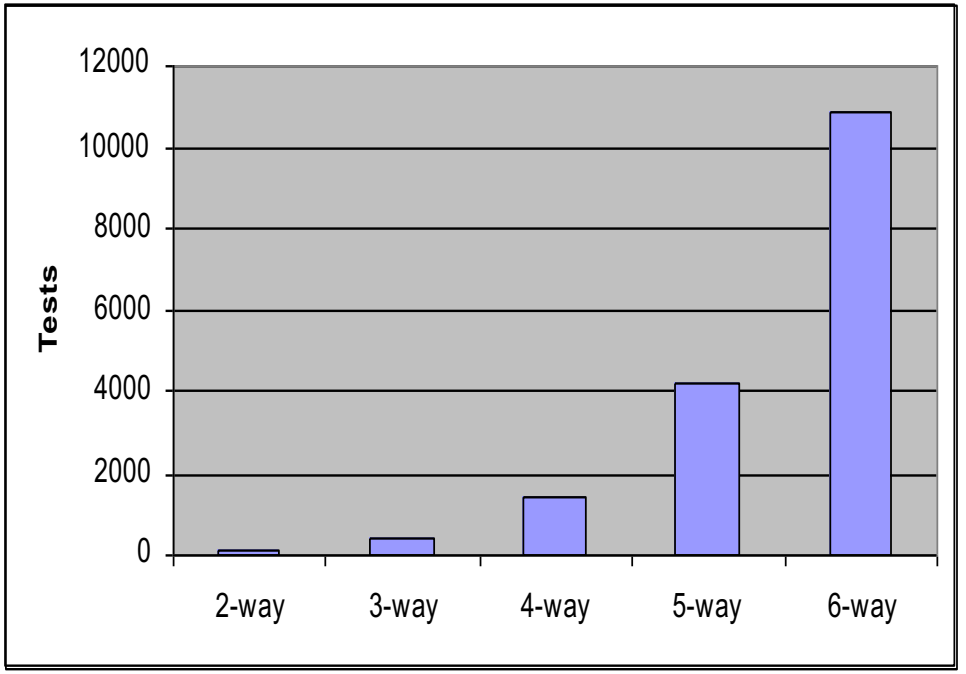
3. **`AG(v1 & v2 & ... & vk -> AX !(R))`**

   Forces production of a counterexample for each R.

   This is just right!

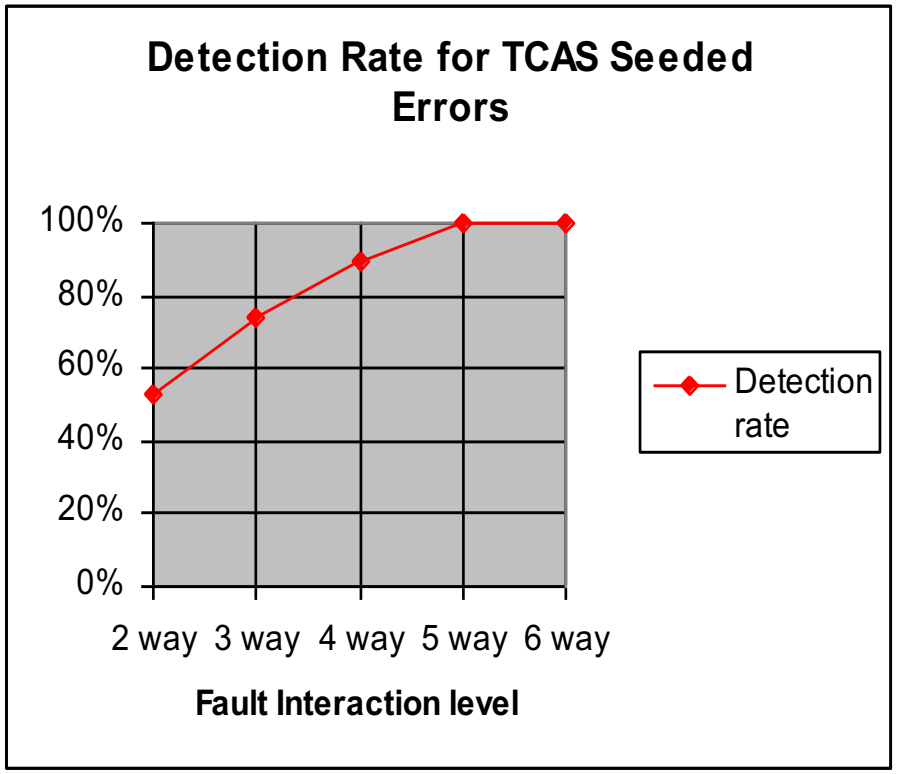
Tests generated

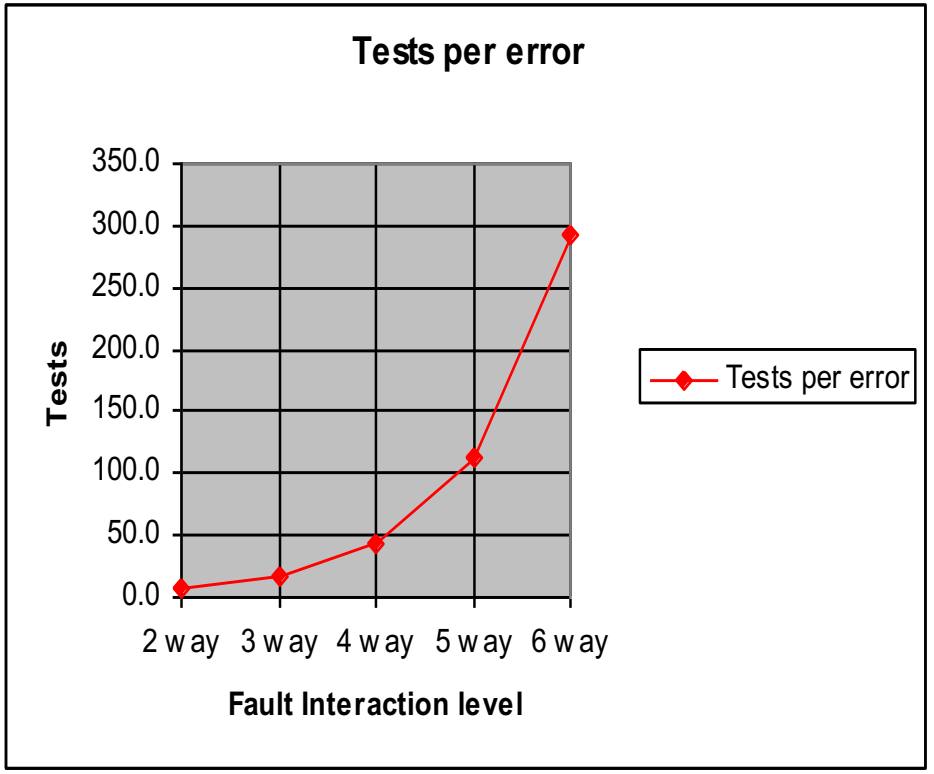
t	Test cases
2-way:	156
3-way:	461
4-way:	1,450
5-way:	4,309
6-way:	11,094



Results

- 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

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)

3 - Rule based systems testing

Background *k*-DNF form

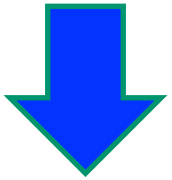
k-DNF = disjunctive normal form expression where no term contains more than *k* variables

For example, *abc* + *de* contains two terms, one with three literals and one with two, so the expression is in **3-DNF** form.

A 3-way covering array contains every possible setting of any 3 variables, as shown in previous slide

Example

```
if (a && (c && !d ||e))  R1;
else if (!a && b && !c)  R2;
else exit();
```



Rules map to logic expressions

$(a(c\bar{d}+e) \rightarrow R_1)$ ← if (a && (c && !d ||e)) R1;

$(\bar{a}\, b\, \bar{c} \rightarrow R_2)$ ← else if (!a && b && !c) R2;

$((\sim(a(c\bar{d}+e)))(\sim(\bar{a}\, b\, \bar{c})) \rightarrow exit)$ ← else exit();

Example: where covering arrays come in

variables: *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:

$emp \;\&\&\; age > 18 \;\&\&\; (fa \;||\; emt \;||\; med) \rightarrow authorize \quad (result_1)$

$else \rightarrow deny \quad (result_2)$

$(emp \;\&\&\; age > 18 \;\&\&\; fa) \;||$
$(emp \;\&\&\; age > 18 \;\&\&\; emt) \;||$
$(emp \;\&\&\; age > 18 \;\&\&\; med)$

Rule Based System Testing

conventional:

- "use cases" verifying important or common situations
- often ad hoc
- may not be sufficiently thorough for high assurance

model-based:

- rules → formal model → model checker or other → test cases
- usually based on fault model; mutation testing

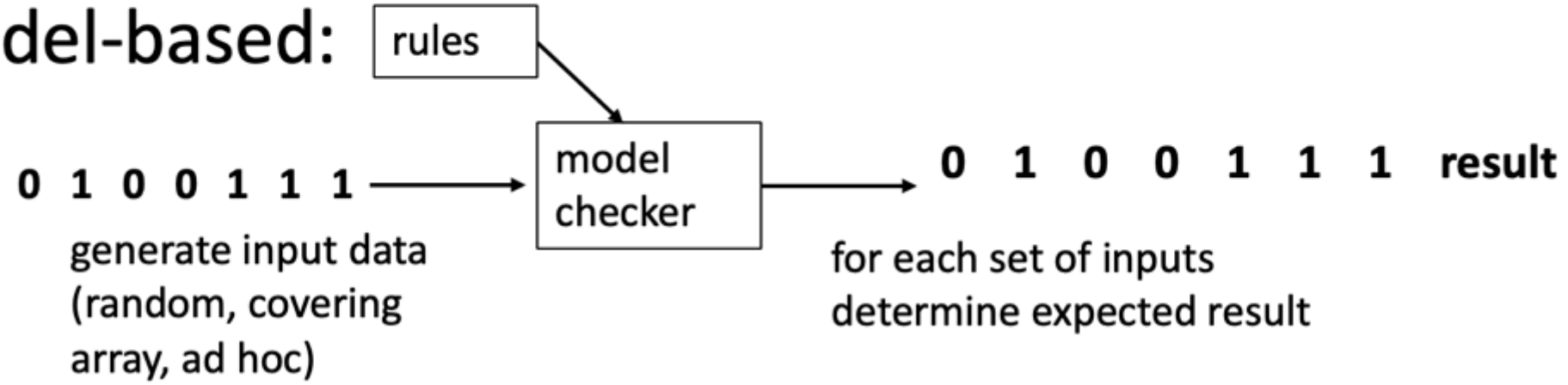
Pseudo-exhaustive testing of rules

Exhaustively test the inputs *on which an output is dependent*

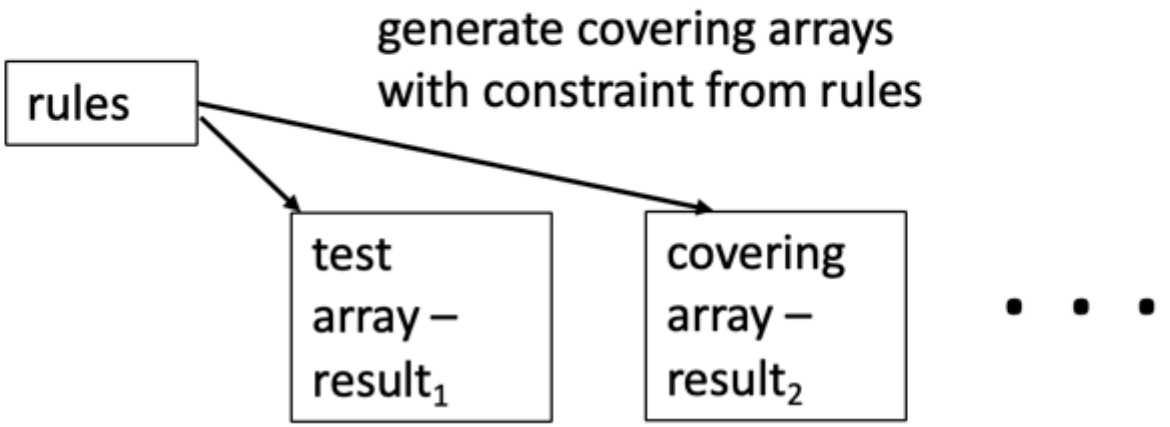
- convert rule antecedents to *k*-DNF form, producing sets of *k* or fewer attributes that will produce a particular result
- generate separate *k*-way covering arrays for combinations that should produce each result

Comparison

model-based:



our approach:



Positive testing (the easy part)

test set PTEST: every test should produce $result_i$

for any input where some combination of *k* input values matches a $result_i$ condition, a decision of $result_i$ is returned.

Construct test set PTEST with one test for each term of *R* :

$\text{PTEST}_i = T_i \bigwedge_{j \neq i} \sim T_j$

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 $result_i$, for *ab* + *cd* →$result_i$, enforce constraint ~(cd)

Negative testing (the hard part)

test set NTEST = covering array of strength *k*, for the set of attributes included in rules R_i for a particular $result_i$

constraints specified by ~ R_i

ensures testing of all conditions that do not produce $result_i$

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

Real world example

HIPAA text: "*(g)(1) Standard: Personal representatives. As specified in this paragraph, a covered entity must, except as provided in paragraphs (g)(3) and (g)(5) of this section, treat a personal representative as the individual for purposes of this subchapter.*
(2) Implementation specification: adults and emancipated minors. If under applicable law a person has authority to act on behalf of an individual who is an adult or an emancipated minor in making decisions related to health care, a covered entity must treat such person as a personal representative under this subchapter, with respect to protected health information relevant to such personal representation.
(3)(i) Implementation specification: unemancipated minors. If under applicable law a"
. . . etc. . . . for 349 pages . . .

mapped 324 words to rules and attributes (about 0.2% of total)

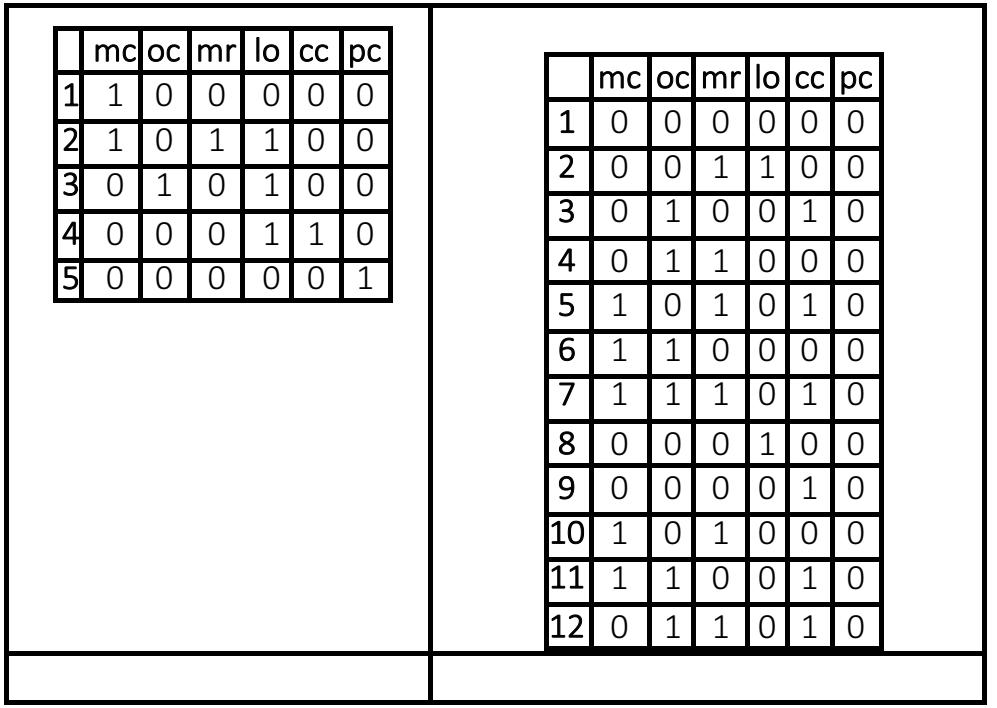
Text	Attributes
(A) The {minor consents : mc} to such health care service; no {other consent : oc} to such health care service is required by law, regardless of whether the consent of another person has also been obtained; and the minor has not {requested that such person : mr} be treated as the personal representative;	expression: mc && ~oc && ~mr attribute sets: {mc, ~oc, ~mr}
(B) The {minor may lawfully obtain : lo} such health care service without the consent of a parent, guardian, or other person acting in loco parentis, and the {minor : mc}, a {court : cc}, or {another person : oc} authorized by law consents to such health care service;	expression: lo && (mc||cc||oc) = lo && mc || lo && cc || lo && oc attribute sets: {lo, mc}, {lo, cc}, {lo, oc}
(C) A {parent, guardian, or other person acting in loco parentis assents to an agreement of confidentiality : pc}	expression: pc attribute sets: {pc}

Generating test sets

Rules: mc && ~oc && ~mr || lo && (mc || cc || oc) || pc → *grant*

To 3-DNF:

mc && ~oc && ~mr || lo&&mc || lo&&cc || lo&&oc || pc → *grant*

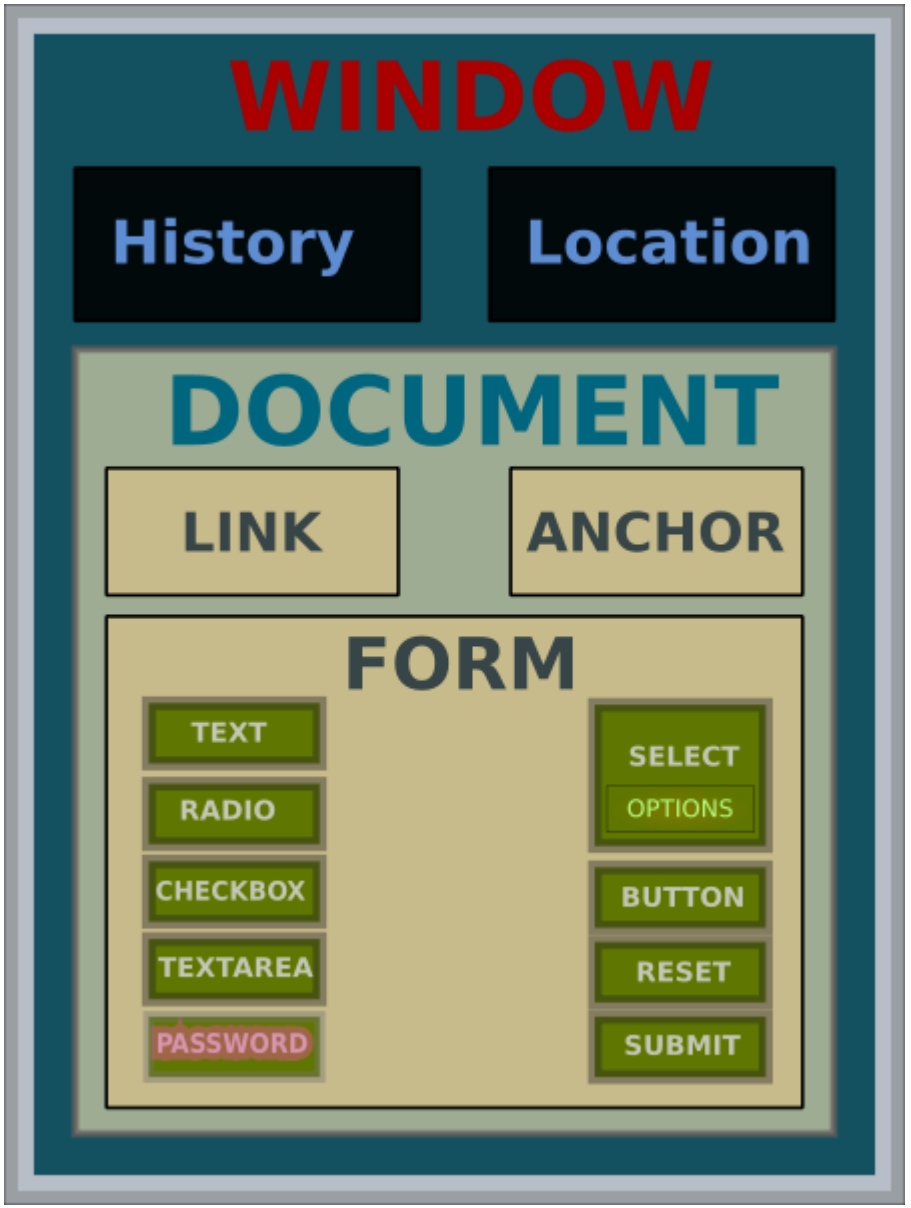


	mc	oc	mr	lo	cc	pc
1	1	0	0	0	0	0
2	1	0	1	1	0	0
3	0	1	0	1	0	0
4	0	0	0	1	1	0
5	0	0	0	0	0	1

	mc	oc	mr	lo	cc	pc
1	0	0	0	0	0	0
2	0	0	1	1	0	0
3	0	1	0	0	1	0
4	0	1	1	0	0	0
5	1	0	1	0	1	0
6	1	1	0	0	0	0
7	1	1	1	0	1	0
8	0	0	0	1	0	0
9	0	0	0	0	1	0
10	1	0	1	0	0	0
11	1	1	0	0	1	0
12	0	1	1	0	1	0

Real world Applications

Application - Web browser – validating 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?

NIST
National Institute of Standards and Technology

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
DOMCharacterDataModified	8	64
DOMElementNameChanged	6	8
DOMFocusIn	5	24
DOMFocusOut	5	24
DOMNodeInserted	8	128
DOMNodeInsertedIntoDocument	8	128
DOMNodeRemoved	8	128
DOMNodeRemovedFromDocument	8	128
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

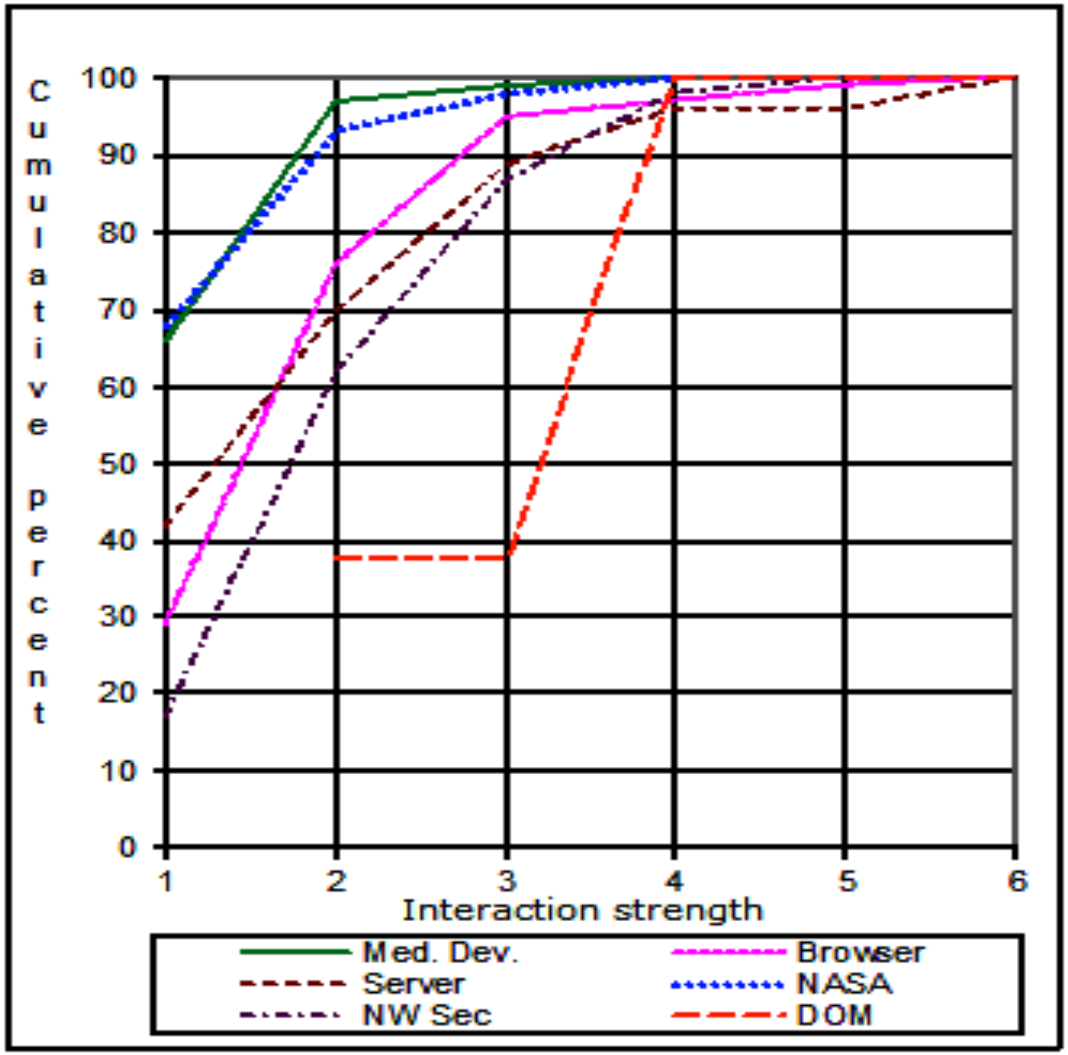
NIST National Institute of Standards and Technology

Document Object Model Test Results

Combinatorial test set:

t	Tests	% of Orig.	Test Results	
			Pass	Fail
2	702	1.92%	202	27
3	1342	3.67%	786	27
4	1818	4.96%	437	72
5	2742	7.49%	908	72
6	4227	11.54 %	1803	72

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

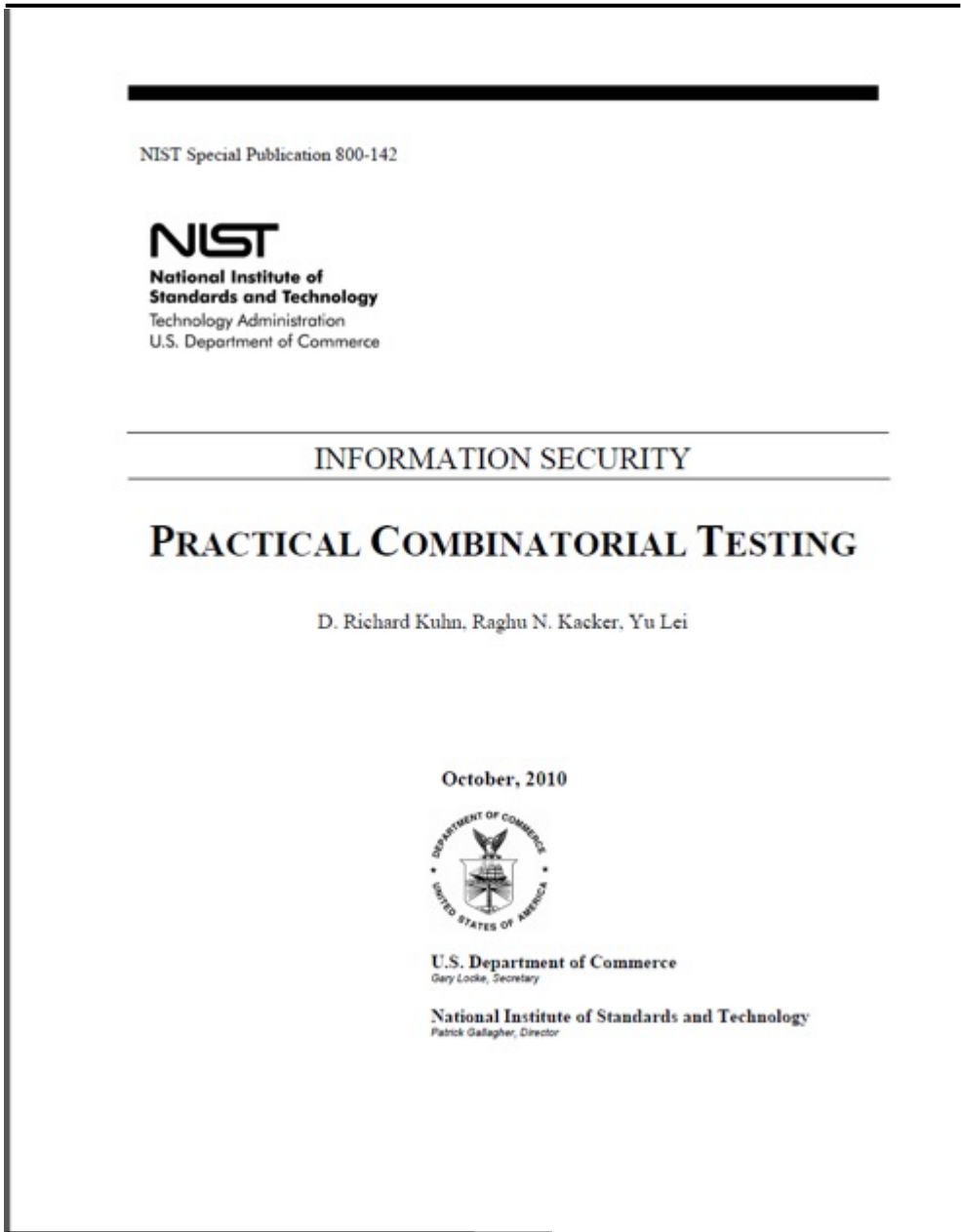


NIST
National Institute of Standards and Technology

Integrating Combinatorial Testing into Test Operations

- Test suite development
  - Generate covering arrays for tests OR
  - Measure coverage of existing tests and supplement

- Training
  - Testing textbooks – Ammann & Offutt, Mathur
  - Combinatorial testing tutorial →
  - User manuals
  - Worked examples
  - **Book – *Introduction to Combinatorial Testing* textbook**


NIST Special Publication 800-142

NIST
National Institute of Standards and Technology
Technology Administration
U.S. Department of Commerce

INFORMATION SECURITY

PRACTICAL COMBINATORIAL TESTING

D. Richard Kuhn, Raghu N. Kacker, Yu Lei

October, 2010

U.S. Department of Commerce
Gary Locke, Secretary

National Institute of Standards and Technology
Patrick Gallagher, Director

Combinatorial Testing 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)

Application - Modeling & Simulation

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)

NIST
National Institute of Standards and Technology

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?

NIST
National Institute of Standards and Technology

Network Deadlock Detection

Deadlocks Detected: combinatorial

t	Tests	500 pkts	1000 pkts	2000 pkts	4000 pkts	8000 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

t	Tests	500 pkts	1000 pkts	2000 pkts	4000 pkts	8000 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

NIST
National Institute of Standards and Technology

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 might never have been found with random testing

Why do this testing? Risks:

- accidental deadlock configuration: low
- deadlock config discovered by attacker: **much higher**

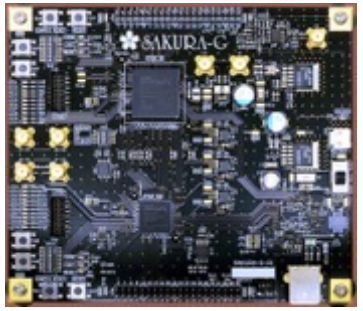
(because they are looking for it)

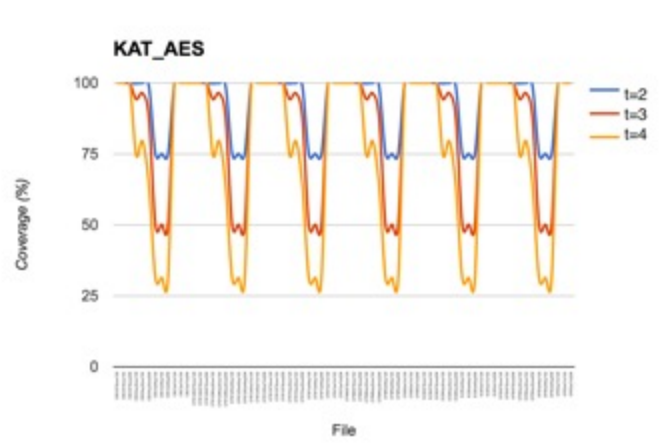
Application - Combinatorial Security Testing

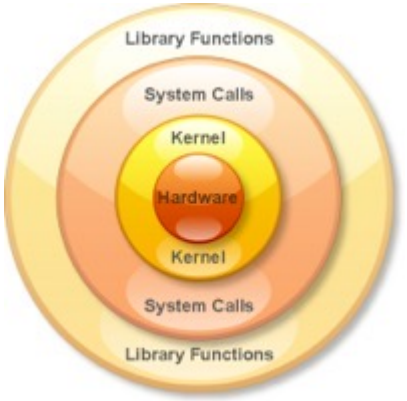
Large scale automated software testing for security

- 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







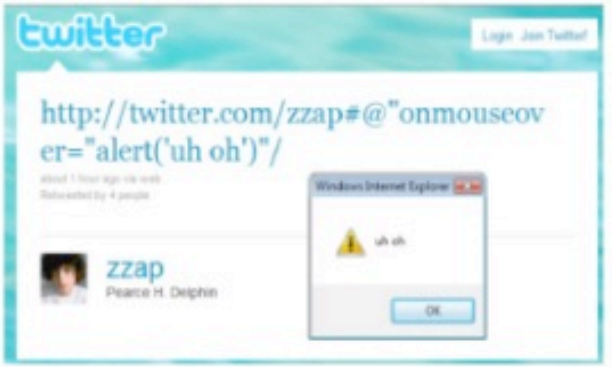
NIST National Institute of Standards and Technology

SBA Research

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

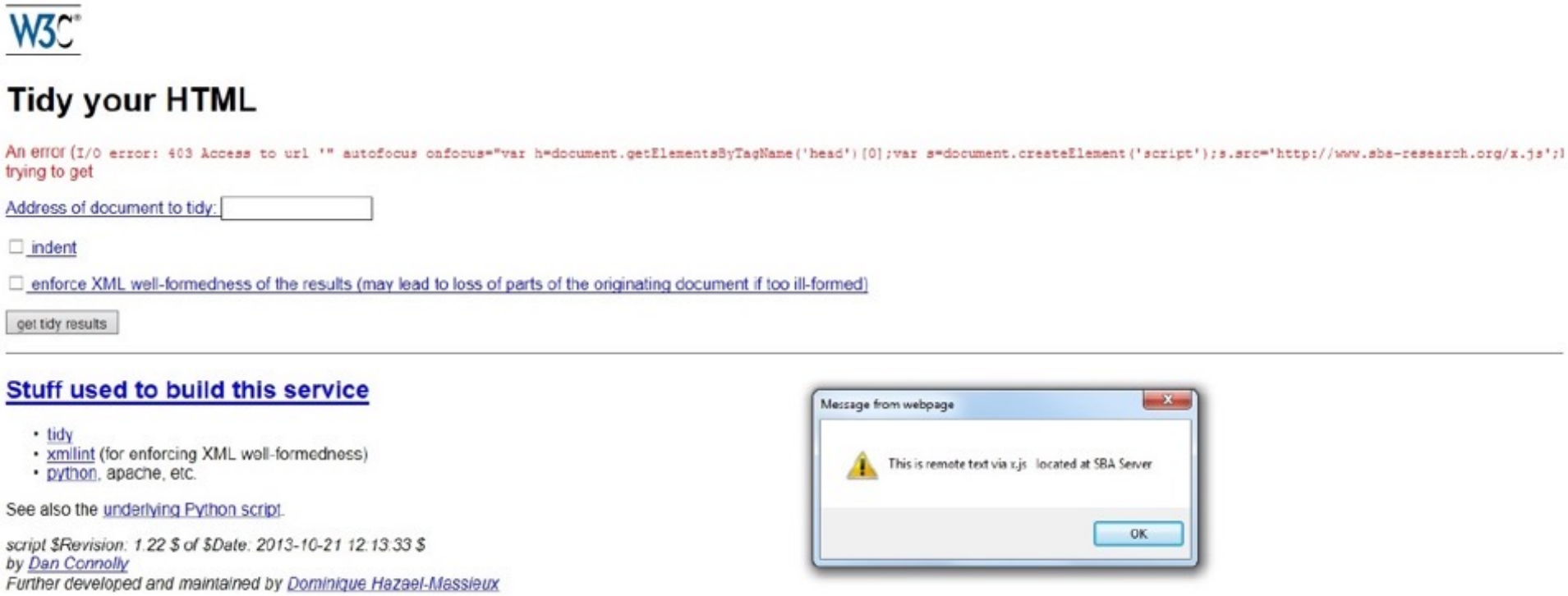


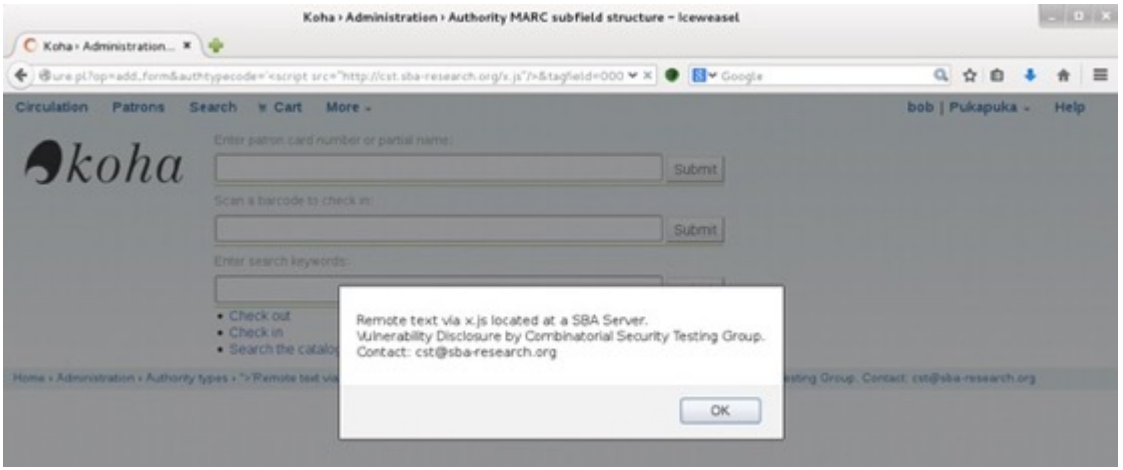
Difference from Classical CT: Modelling Attack Vectors

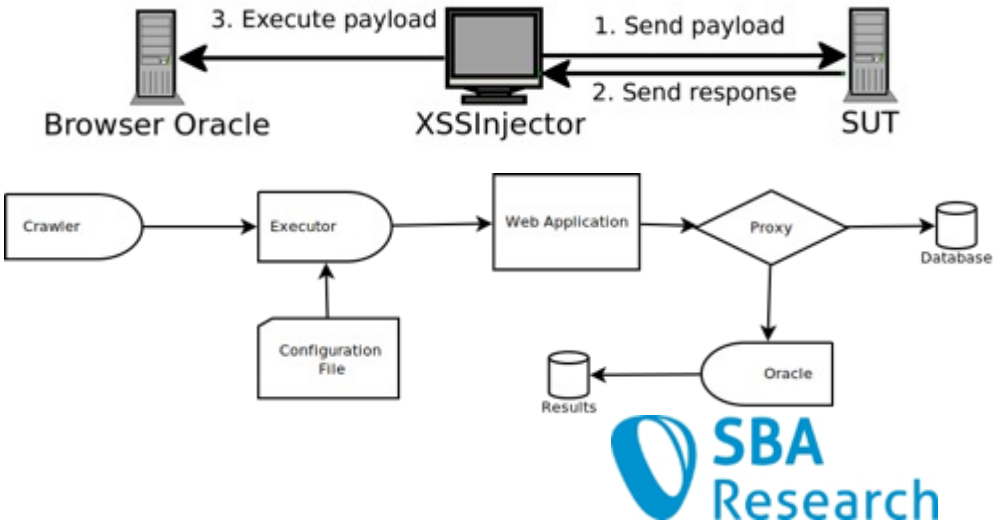
- Attacker injects client-side script in parameter msg:
  `http://www.foo.com/error.php?msg=<script>alert(1)</script>`

SBA Research

Sample of XSS and SQLi vulnerabilities found







SBA Research

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>	␣	’;	␣	onError=	␣	alert(1)	␣	’>	␣	\>
"><script>	␣	’>	␣	onError=	␣	alert(1)	␣	’>	␣	\>
"><script>	␣	’;	␣	onError=	␣	src="invalid"	␣	’>	␣	\>
"><script>	␣	’>	␣	onError=	␣	src="invalid"	␣	’>	␣	\>

Retrieving the Root Cause of Security Vulnerabilities

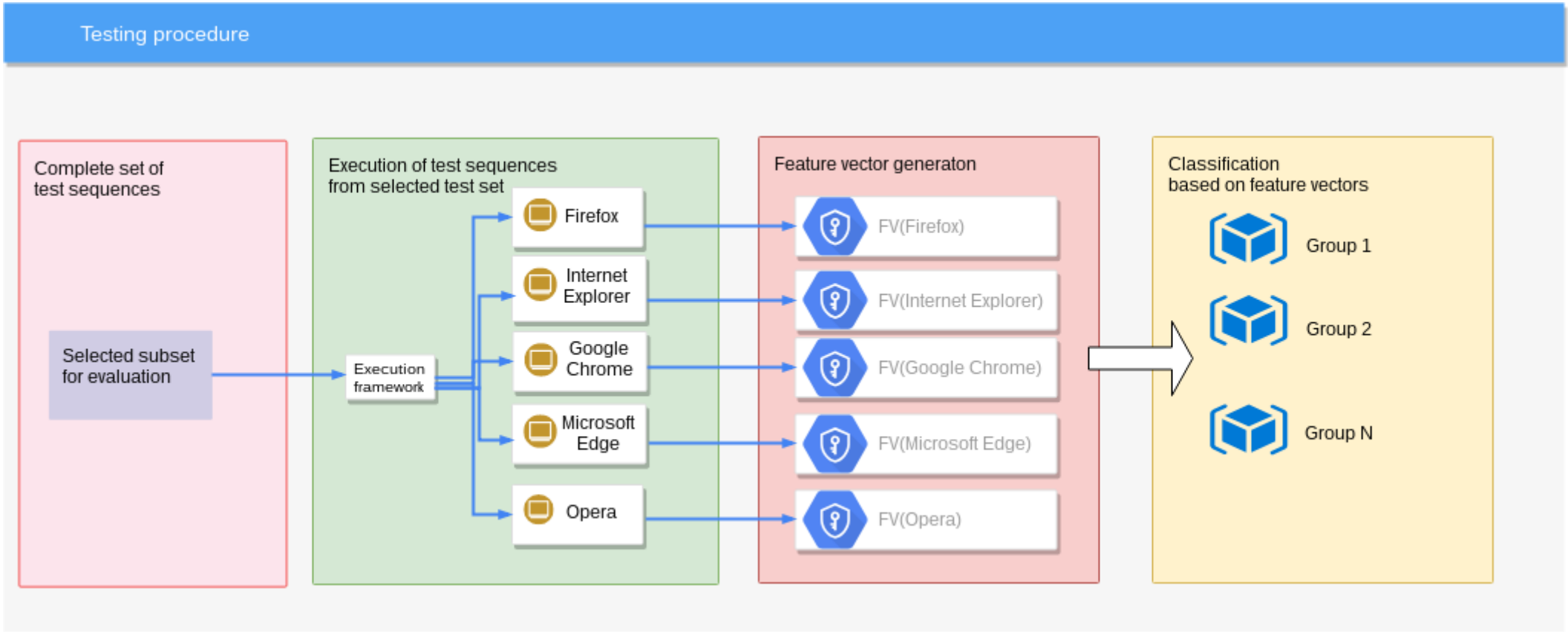
- Analysis revealed common structure for successful XSS Vectors
- E.g. all contain the following 2-tuple: ("><script>, onError=)

UNIVERSITY OF TEXAS ARLINGTON

SBA Research

Application - 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



National Institute of Standards and Technology

SBA Research

SCAs for browser fingerprinting: evaluation

Complete test sequence set: $\mathcal{S}$ with $|\mathcal{S}| = 1956$

Browsers

1. Mozilla Firefox, version 64.0.0.6914;
2. Google Chrome, version 71.0.3578.98;
3. Microsoft Internet Explorer, version 11.0.17134.1;
4. Microsoft Edge, version 11.00.17134.471.
5. Opera, version 57.0.3098.106;

1. {Firefox},
2. {Google Chrome, Opera},
3. {Microsoft Internet Explorer, Microsoft Edge}

New Areas and Next Steps

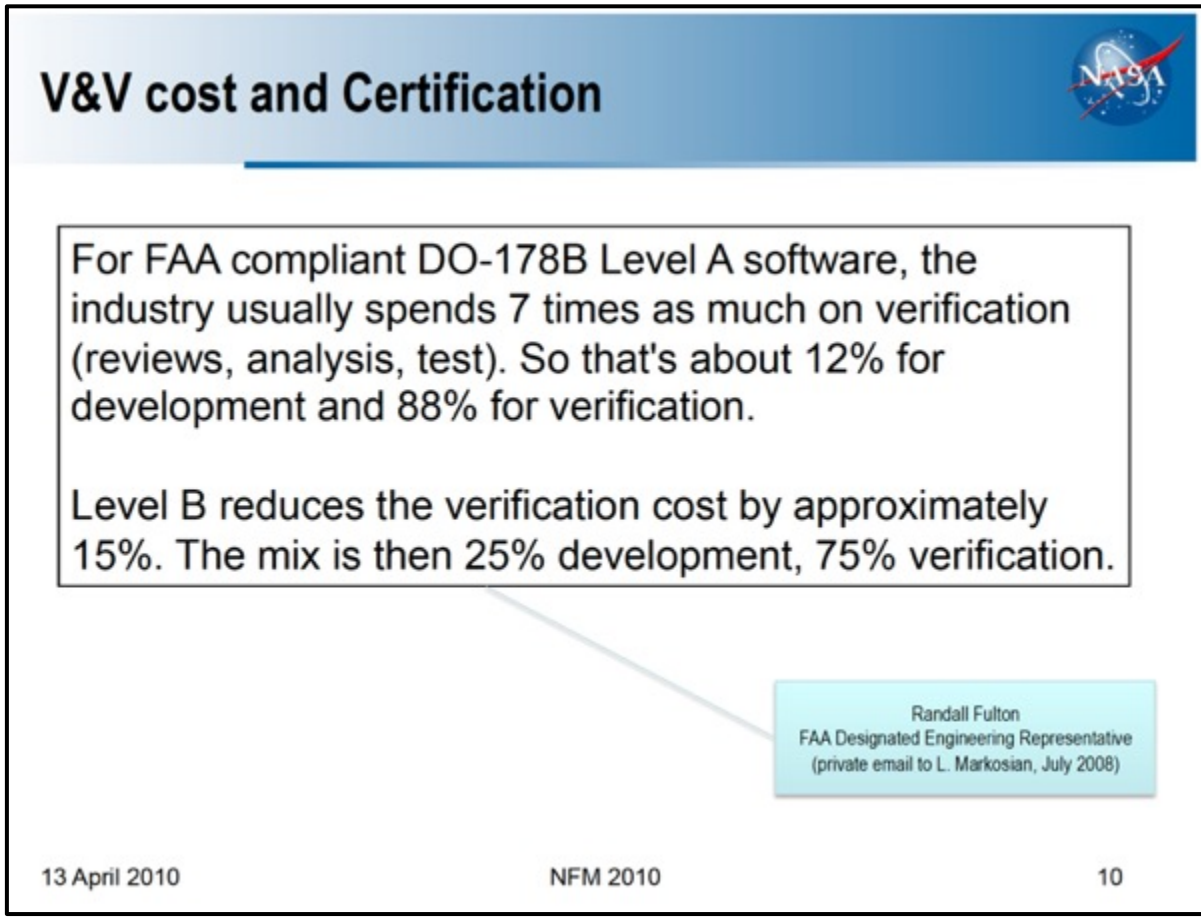
Application - Autonomous Systems

Software safety assurance is already *very* expensive

Consumer level software cost: about 50% code development, 50% verification

For aviation life-critical, 12% code development, 88% verification
(Software is about 30% of cost for new civilian aircraft, higher for military)

Autonomy makes the problem even harder!



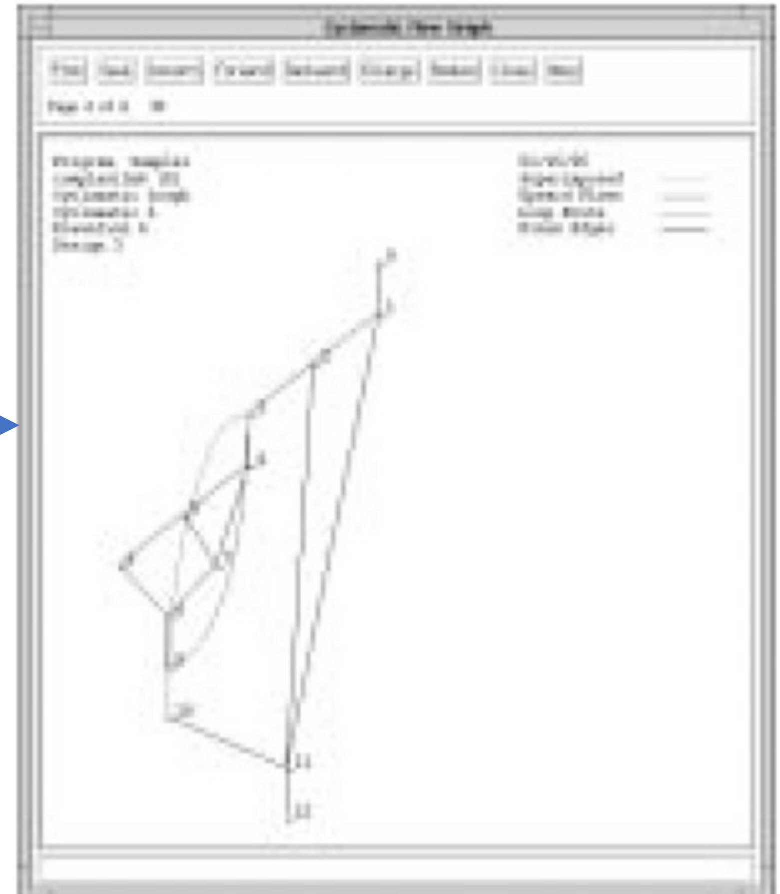
Why can't we use same processes as other safety-critical software ?

- Nearly all conventional software testing is based on structural coverage – ensuring that statements, decisions, paths are covered in testing
- Life-critical aviation software requires MCDC testing, white-box criterion that cannot be used for neural nets and other black-box methods

Code coverage works well - for conventional software

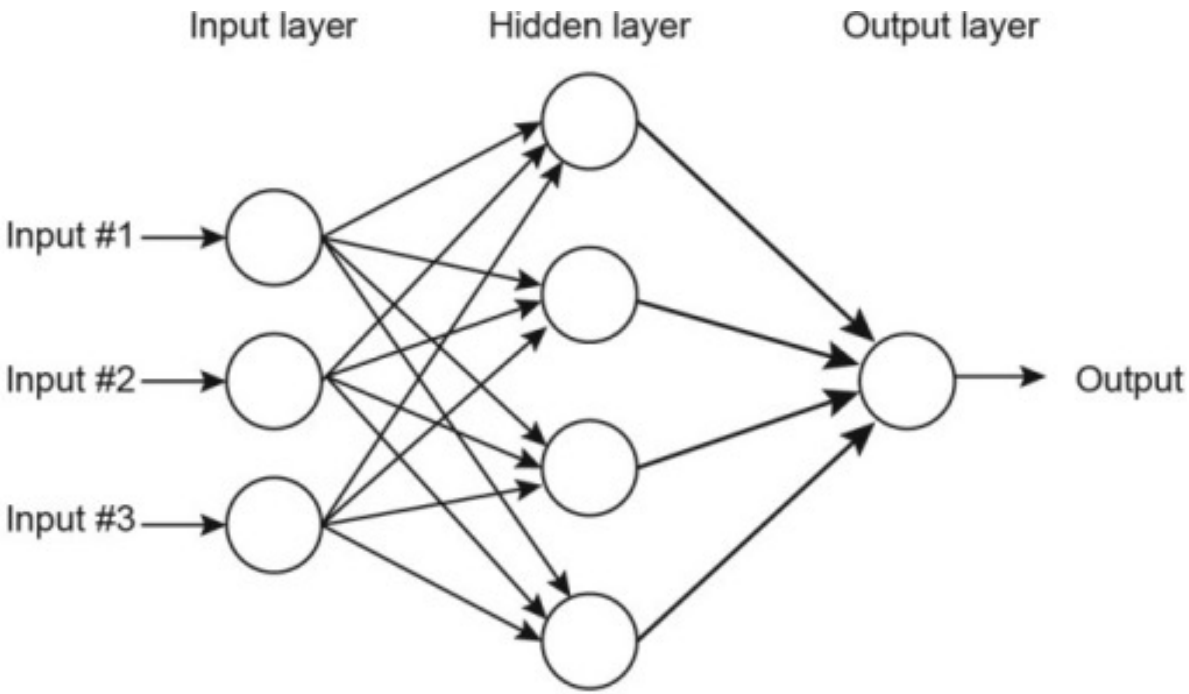
Annotated Source Listing

```
                                                              01/

                                                           Start  N
                                  v(G)   ev(G)   iv(G)    Line   L
-------------------------------------------------------  -----  -
                                   6       6       3        40

  complexity6(int i, int j)
  {/* Sample module with complexity 6 */
      if (i > 0 && j > 0) {
          while (i > j) {
              if (i % 2 && j % 2)
                  printf("%d\n", i);
              else
                  printf("%d\n", j);
              i--;
          }
      }
  }
```



Can we use code coverage for machine learning?

- Much of AI/ML depends on various neural nets
- Algorithm and code stays the same
- Connections and weights vary
- Behavior changes depending on inputs used in training



To monitor and guard input space, need to measure

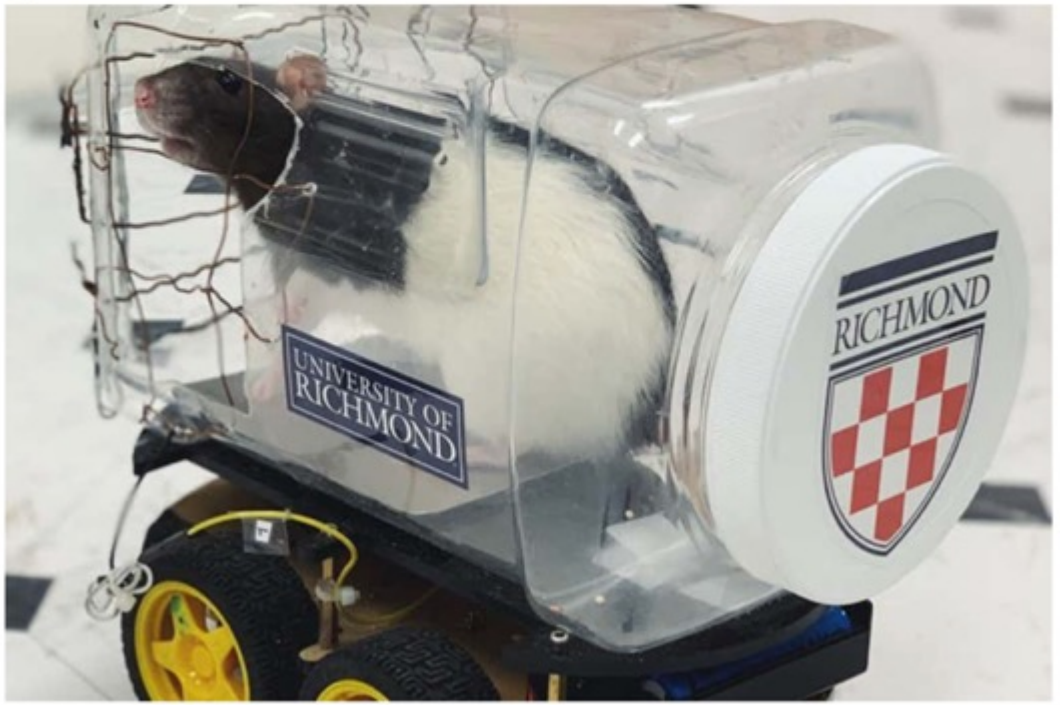
- Gold standard of assurance and verification of life-critical software can't be used for much of new life-critical autonomy software
- We can measure "neuron coverage", but indirect measure and not clear how closely related to accuracy and ability to correctly process all of the input space
- Measure the input space directly
- Then see if the AI system handles all of it correctly

NewScientist

Scientists have trained rats to drive tiny cars to collect food

LIFE 22 October 2019

By Alice Klein



It doesn't take much intelligence to drive a car.

Even rats can do it!

But can they do it under <u>all kinds of conditions</u> ?

The problem is harder outside of a constrained environment

Things get tricky as the scene becomes complex

- Multiple conditions involved in accidents
  - "The camera failed to recognize the white truck against a bright sky”
  - "The sensors failed to pick up street signs, lane markings, and even pedestrians due to the angle of the car shifting in rain and the direction of the sun”
- We need to understand what combinations of conditions are included in testing

Combinatorial value coverage

a	b	c	d
0	0	0	0
0	1	1	0
1	0	0	1
0	1	1	1

Vars	Combination values	Coverage
a b	00, 01, 10	.75
a c	00, 01, 10	.75
a d	00, 01, 11	.75
b c	00, 11	.50
b d	00, 01, 10, 11	1.0
c d	00, 01, 10, 11	1.0

19 combinations included in test set

100% coverage of 33% of combinations
75% coverage of half of combinations
50% coverage of 16% of combinations

Kuhn, D. R., Mendoza, I. D., Kacker, R. N., & Lei, Y. (2013). Combinatorial coverage measurement concepts and applications. *2013 IEEE Sixth Intl Conference on Software Testing, Verification and Validation Workshops*

NIST National Institute of Standards and Technology

Vars	Combination values	Coverage
a b	00, 01, 10	.75
a c	00, 01, 10	.75
a d	00, 01, 11	.75
b c	00, 11	.50
b d	00, 01, 10, 11	1.0
c d	00, 01, 10, 11	1.0

Total possible 2-way combinations = $2^2 \binom{4}{2} = 24$

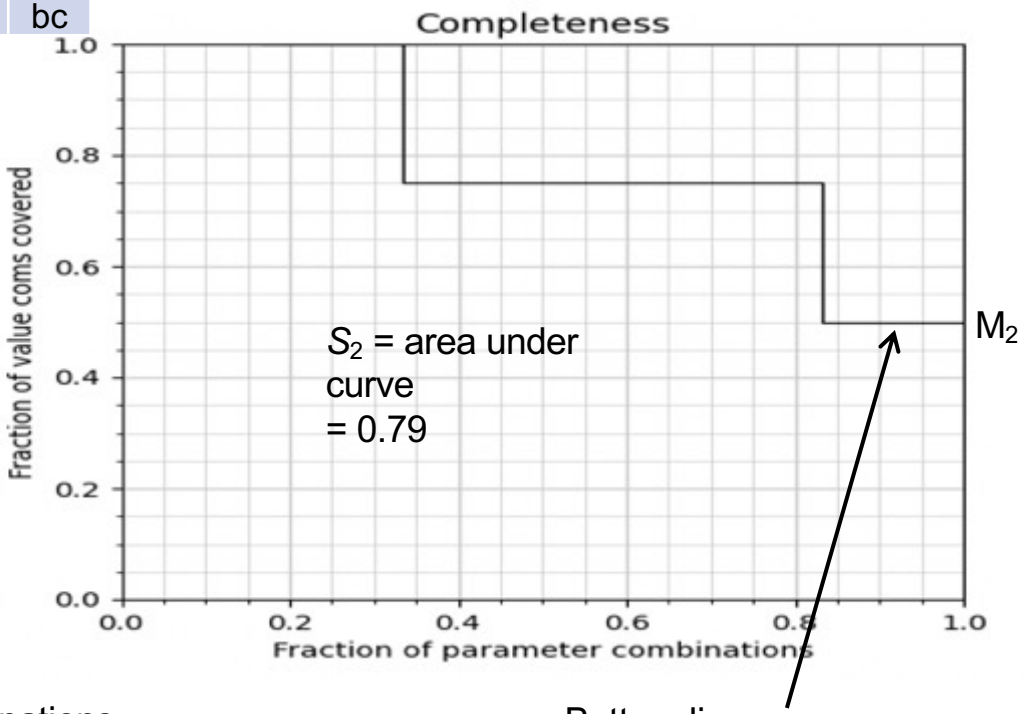
S_2 = fraction of 2-way combinations covered = 19/24
= 0.79

Rearranging the table:

1.00	00	00				
.75	01	01	00	00	00	
.50	10	10	01	01	01	00
.25	11	11	10	10	11	11
	bd	cd	ab	ac	ad	bc

Graphing Coverage Measurement

1.00	00	00				
.75	01	01	00	00	00	
.50	10	10	01	01	01	00
.25	11	11	10	10	11	11
	bd	cd	ab	ac	ad	bc



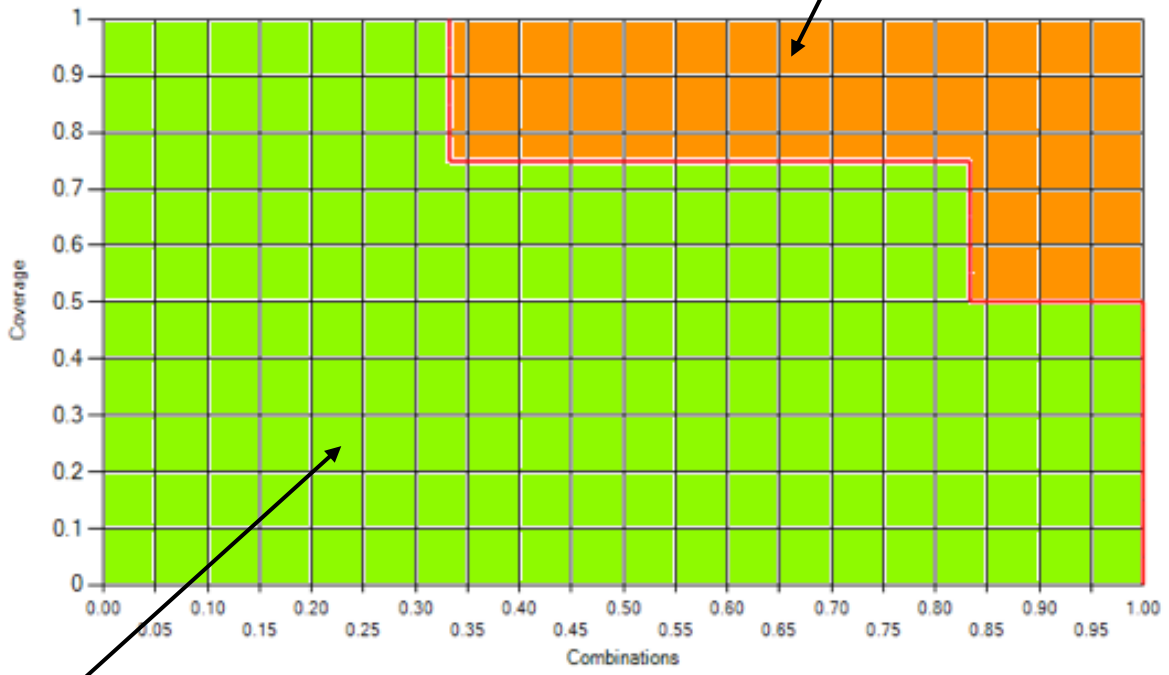
100% coverage of .33 of combinations
75% coverage of .50 of combinations
50% coverage of .16 of combinations

Bottom line:
All combinations covered to at least .50

What else does this chart show?

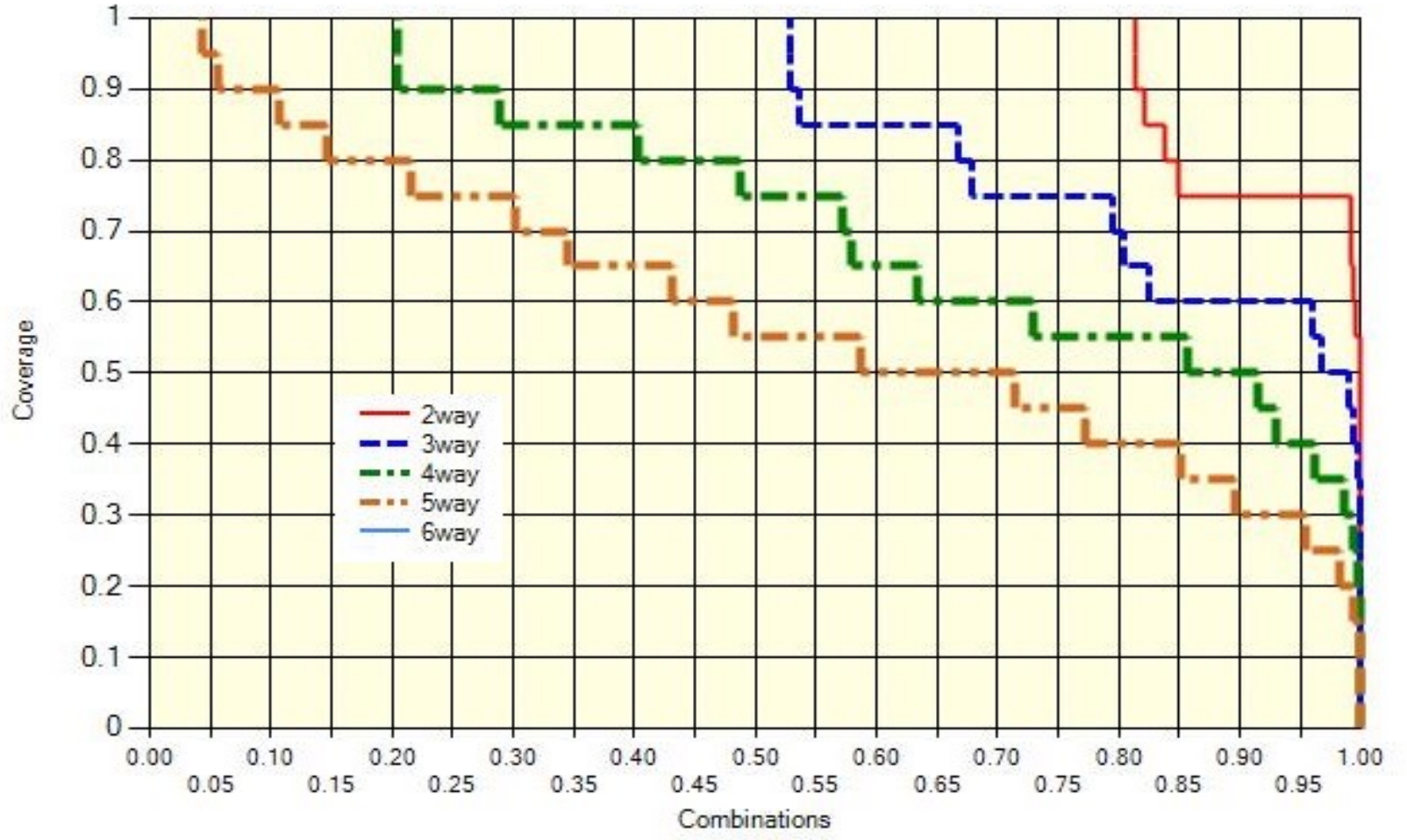
1 - S_t = **Untested combinations**

(look for problems here)



S_t = Tested combinations => code works for these

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



NIST
National Institute of Standards and Technology

Application - Transfer learning what is the problem?

- Differences inevitably exist between training data sets, test data sets, and real-world application data
- Further differences exist between data from two or more different environments
- How do we predict performance of a model trained on one data set when applied to another?
  - New environment
  - Changed environment
  - Additional possible values
  - etc.

Lanus, E., Freeman, L. J., Kuhn, D. R., & Kacker, R. N. (2021, April). Combinatorial Testing Metrics for Machine Learning. In *2021 IEEE Intl Conference on Software Testing, Verification and Validation Workshops (ICSTW)*

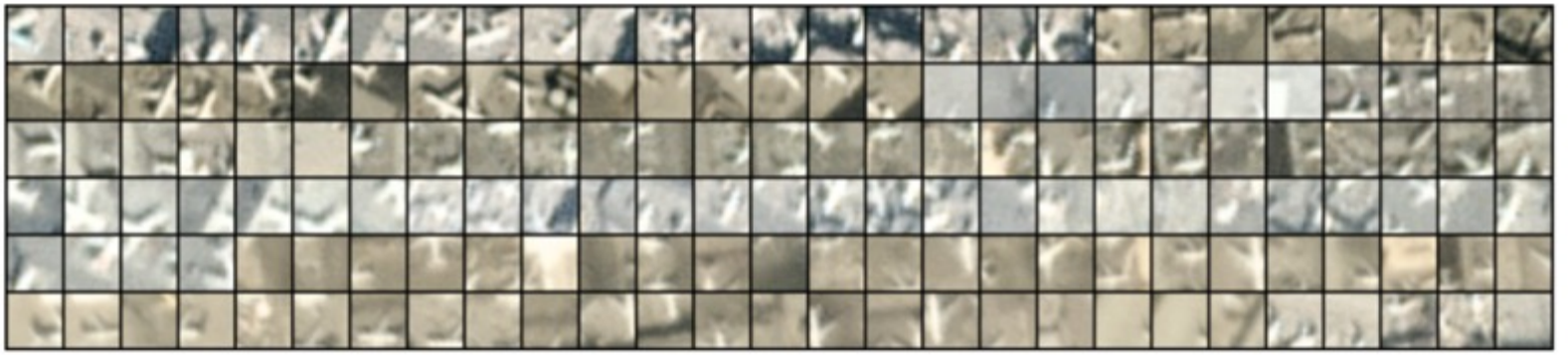
Transfer learning – conventional practice

- Randomized selection – but will randomization be sufficient, especially with smaller data sets?
- Ensure at least one of each object type – but this may not be representative of object attribute distributions

- Interactions are critical to consider in most ML problems, especially for safety, but conventional practice does little to ensure data sets are adequately representative of interactions

Example – image analysis

- Planes in satellite imagery – Kaggle ML data set – determine if image contains or does not contain an airplane
- Two data sets – Southern California (SoCal, 21,151 images) or Northern California (NorCal, 10,849 images)
- 12 features, each discretized into 3 equal range bins



Transfer learning problem

- Train model on one set, apply to the other set
- Problem –
  - Model trained on larger, SoCal data applied to smaller, NorCal data → performance drop
  - Model trained on smaller, NorCal data applied to larger, SoCal data → NO performance drop
- This seems backwards!
- Isn't it better to have more data?
- Can we explain this and predict it next time?

Density of combinations in one but not the other data set, 2-way

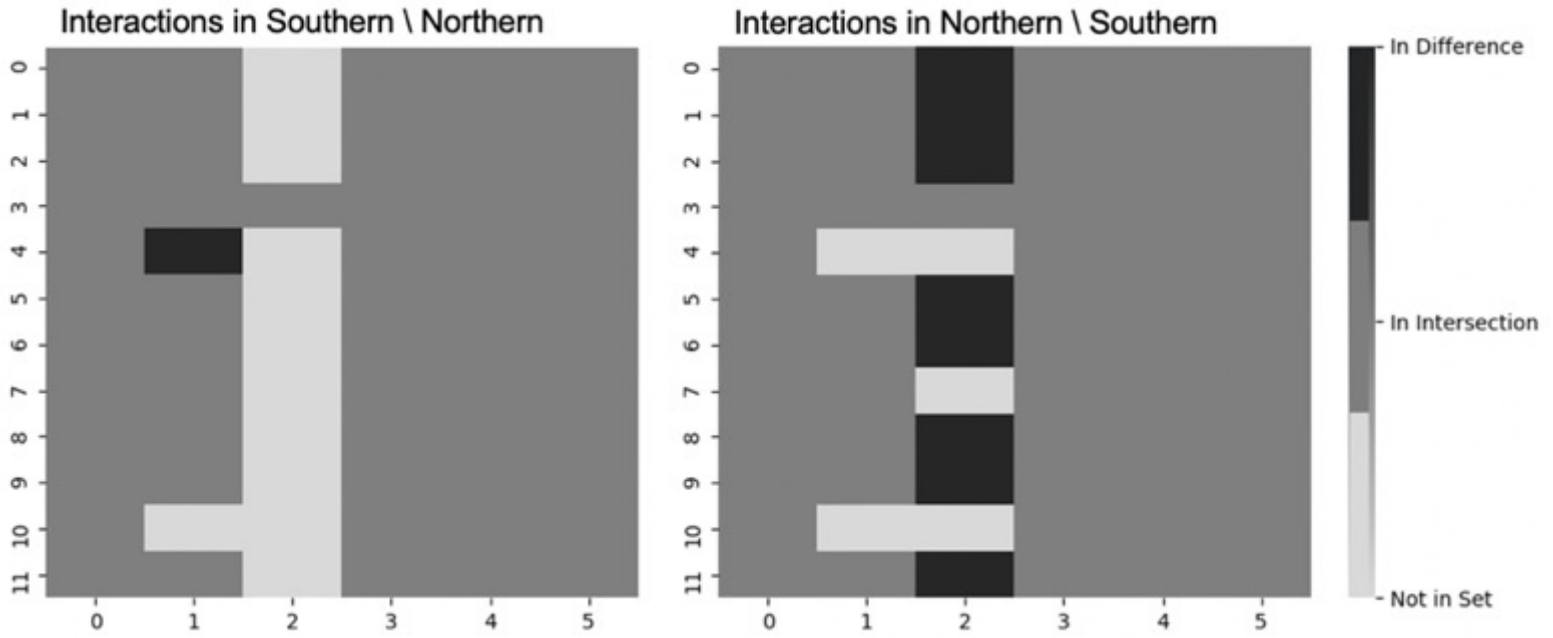


Image from Combinatorial Testing Metrics for Machine Learning, Lanus, Freeman, Kuhn, Kacker, IWCT 2021

For C = SoCal, N = NorCal,
|C\N| / |C| = 0.02
|N\C| / |N| = 0.12

→ The NorCal data set has fewer "never seen" combinations, even with half as many observations

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 software
- 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, M.S. Raunak
{kuhn, raghu.kacker, raunak}@nist.gov

http://csrc.nist.gov/acts

NIST
National Institute of Standards and Technology