Introduction to Combinatorial Methods

Testing Strategies for Artificial Intelligence Enabled Systems

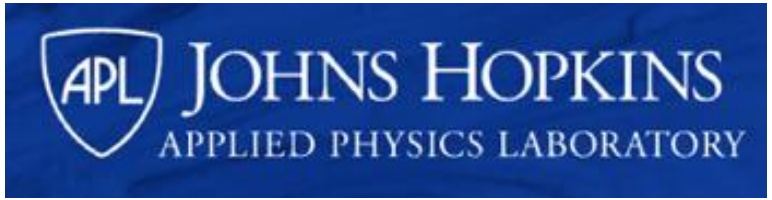
NIST/Virginia Tech, 4 Sept 2024

Rick Kuhn
National Institute of Standards and Technology
Gaithersburg, Maryland 20899
kuhn@nist.gov

NIST NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY U.S. DEPARTMENT OF COMMERCE

What is NIST and why are we doing this?

- US Government agency, which supports US industry through developing better measurement and test methods
- 3,000 scientists, engineers, and staff including 4 Nobel laureates
- Broad involvement with industry and academia



What are interaction faults?

- 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

pressure < 10 & volume > 300

(interaction between 2 factors)

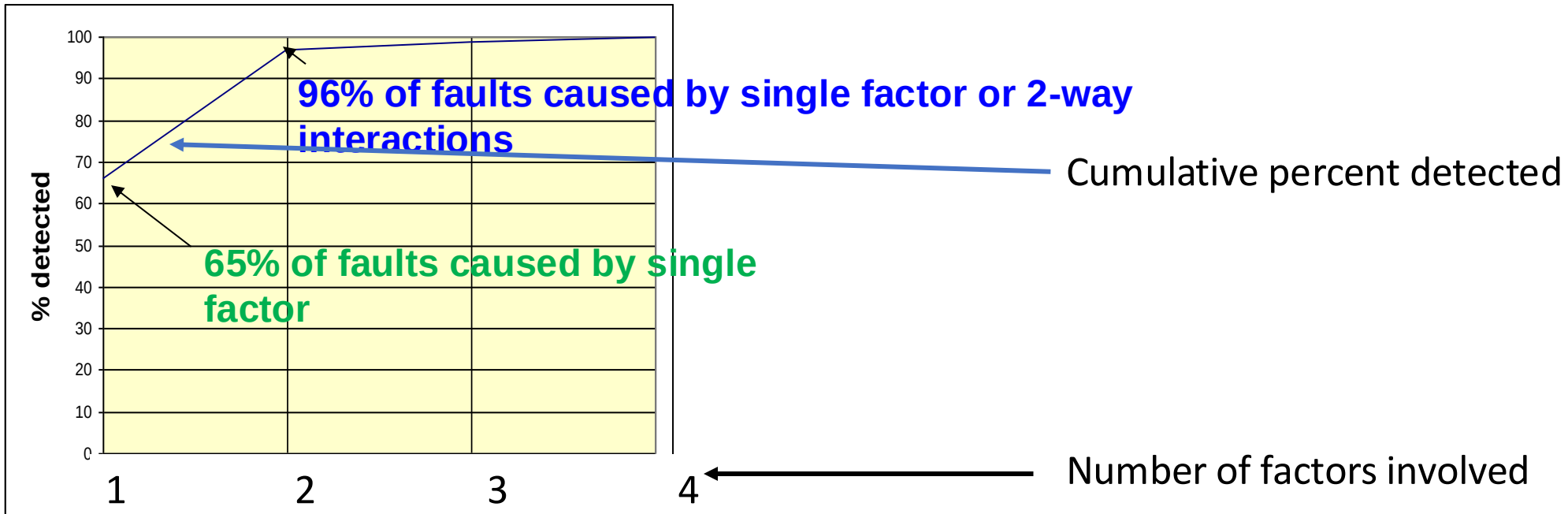
So this is a **2-way interaction**

=> testing all pairs of values can find this fault

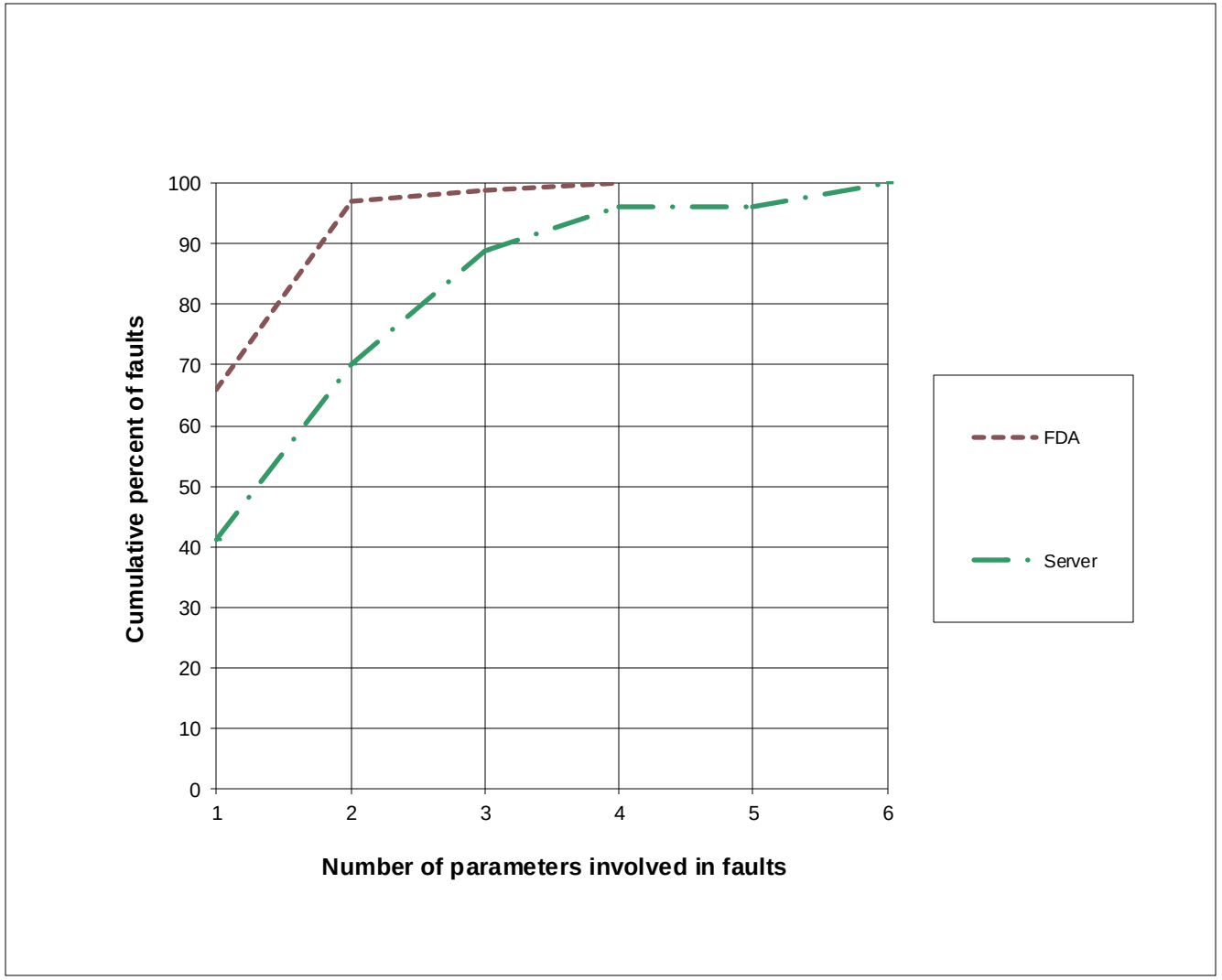
How are interaction faults distributed?

- 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



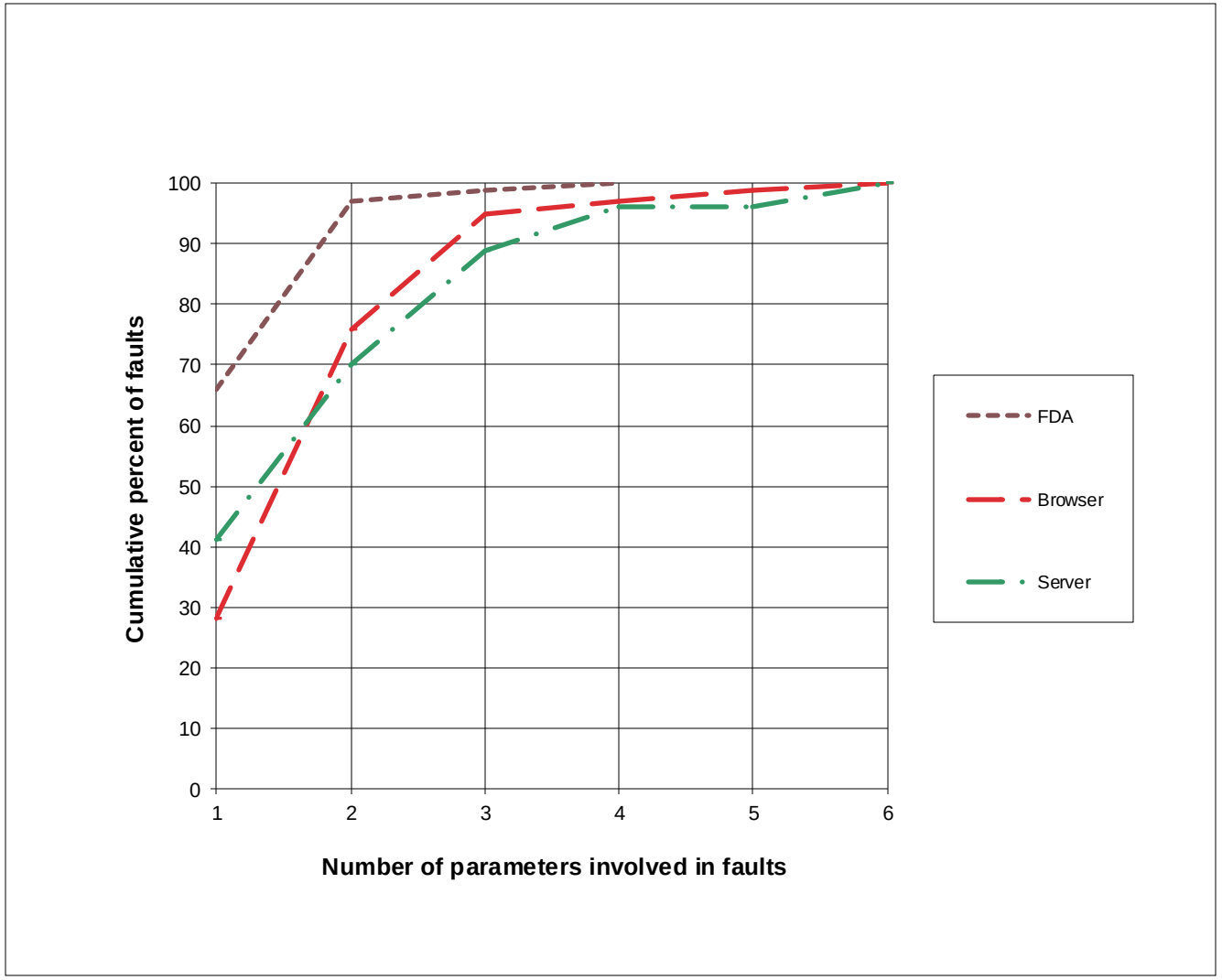
Server



These faults more complex than medical device software!!

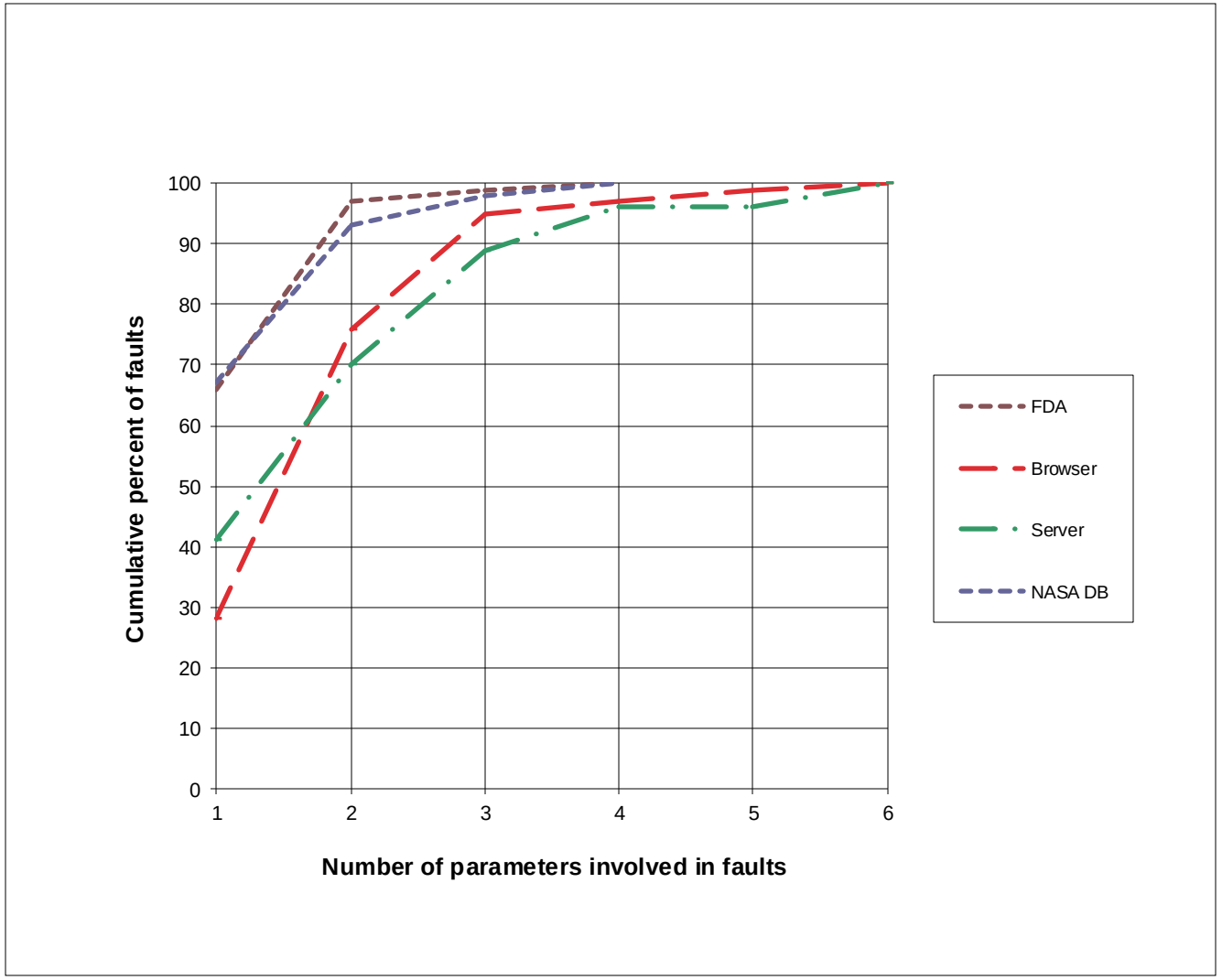
Why?

Browser



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

NASA distributed database

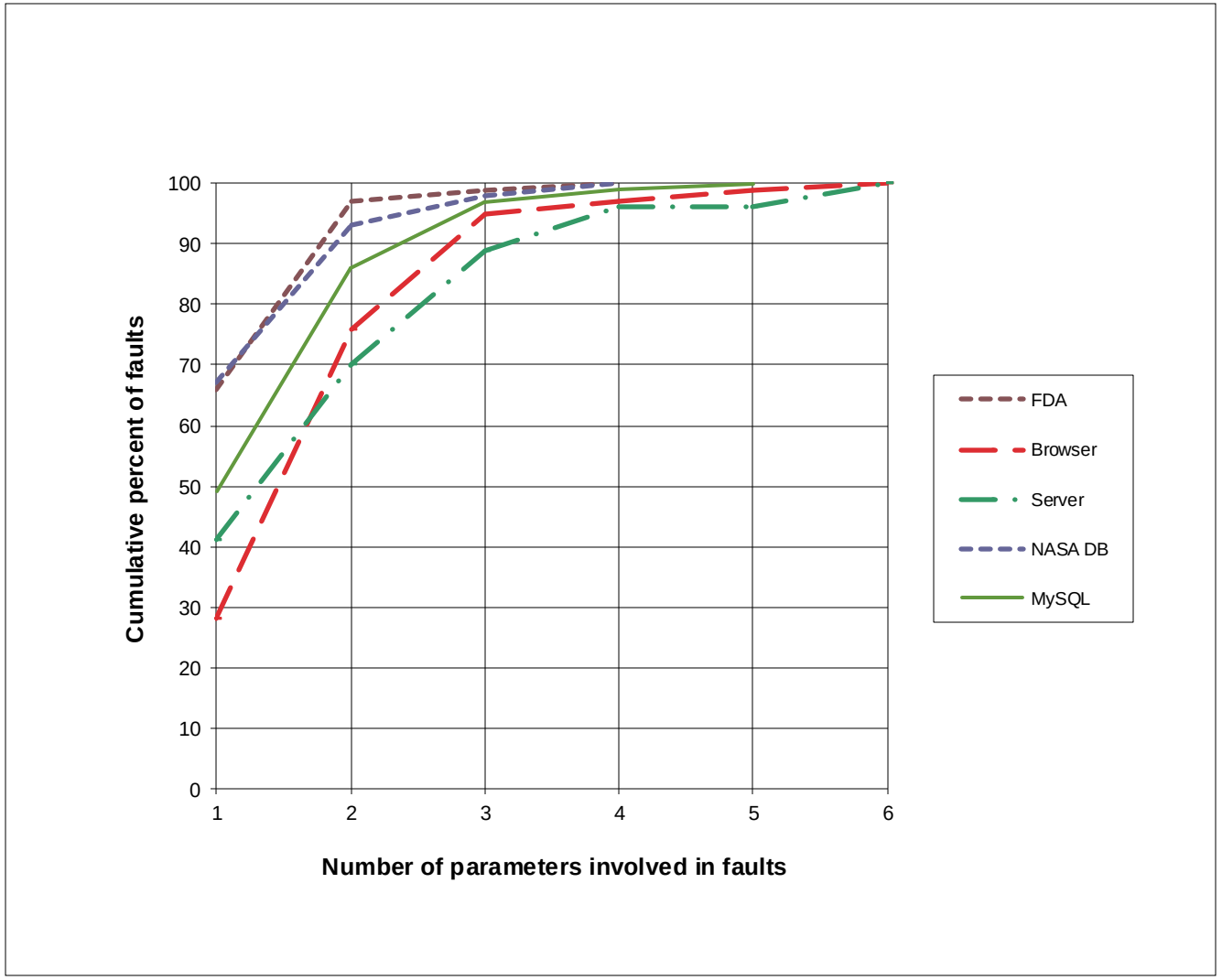


Note: initial testing

but ….

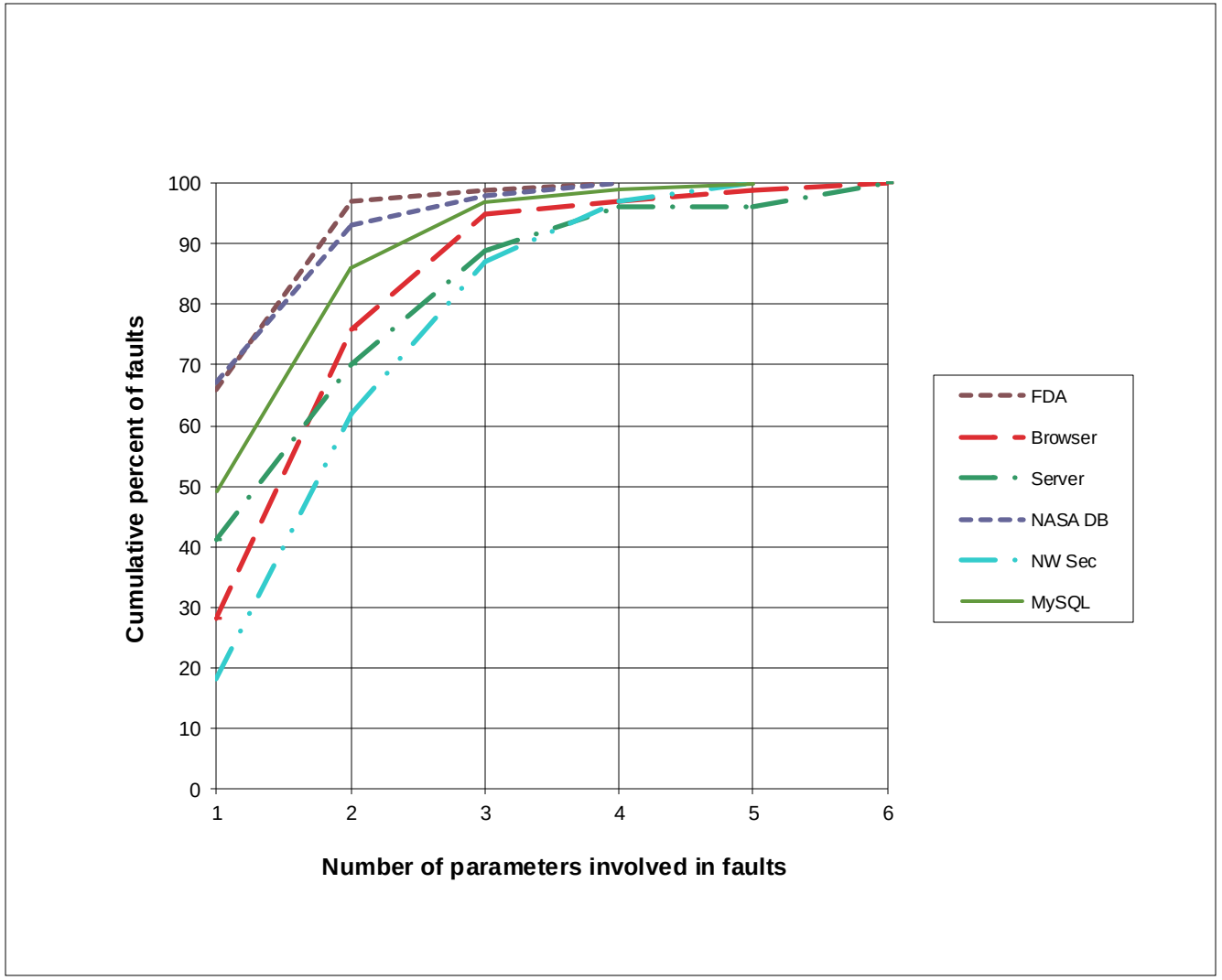
Fault profile better than medical devices!

MySQL database server



National Institute of Standards and Technology

TCP/IP

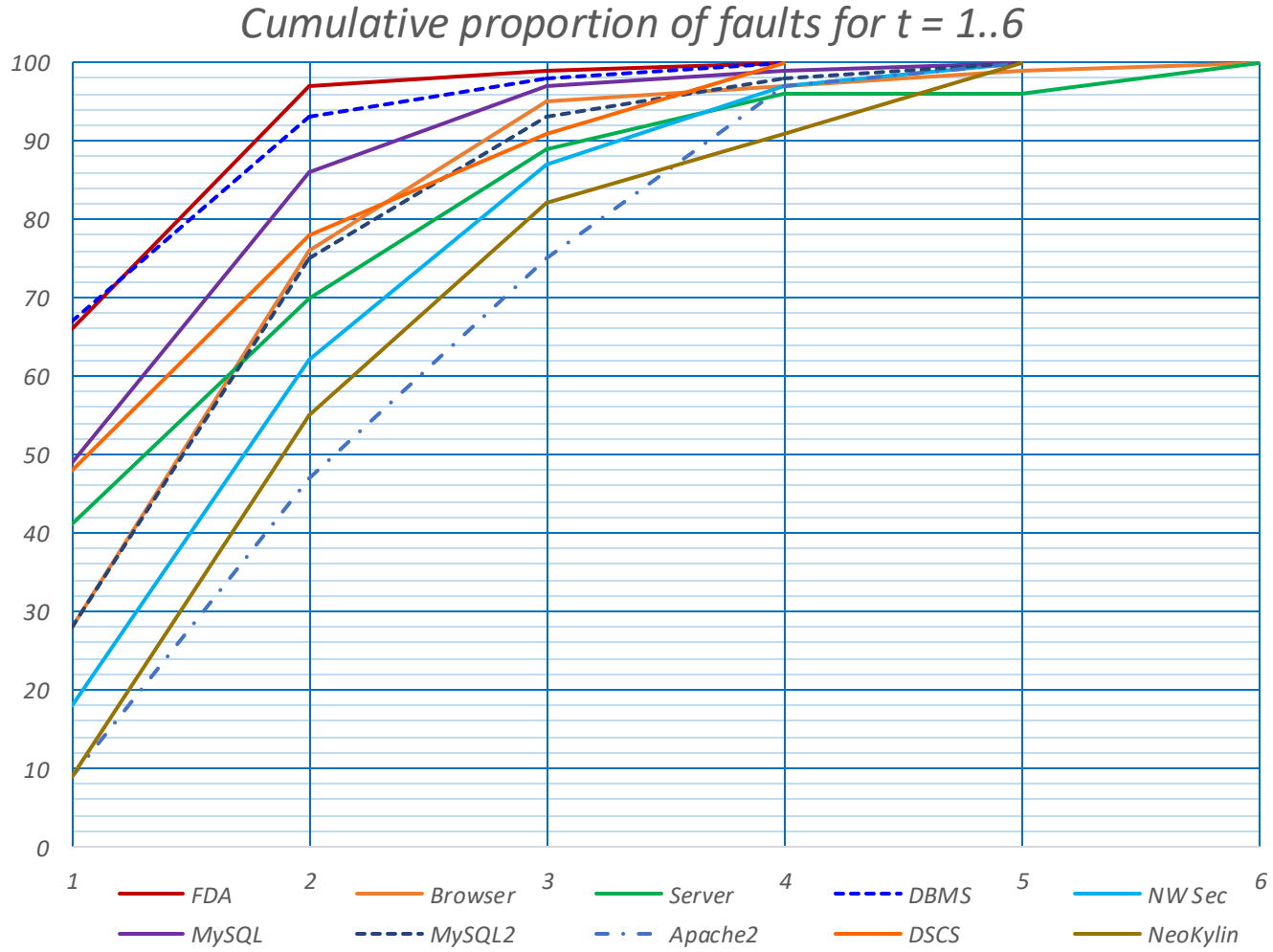

Cumulative percent of faults
100
90
80
70
60
50
40
30
20
10
0
1
2
3
4
5
6
Number of parameters involved in faults
FDA
Browser
Server
NASA DB
NW Sec
MySQL

Various domains collected

Wide variation in percent of failures caused by single factor

Variability decreases as number of factors increases

More testing or users => harder to find errors, fewer single factor failures



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

How does this knowledge help?

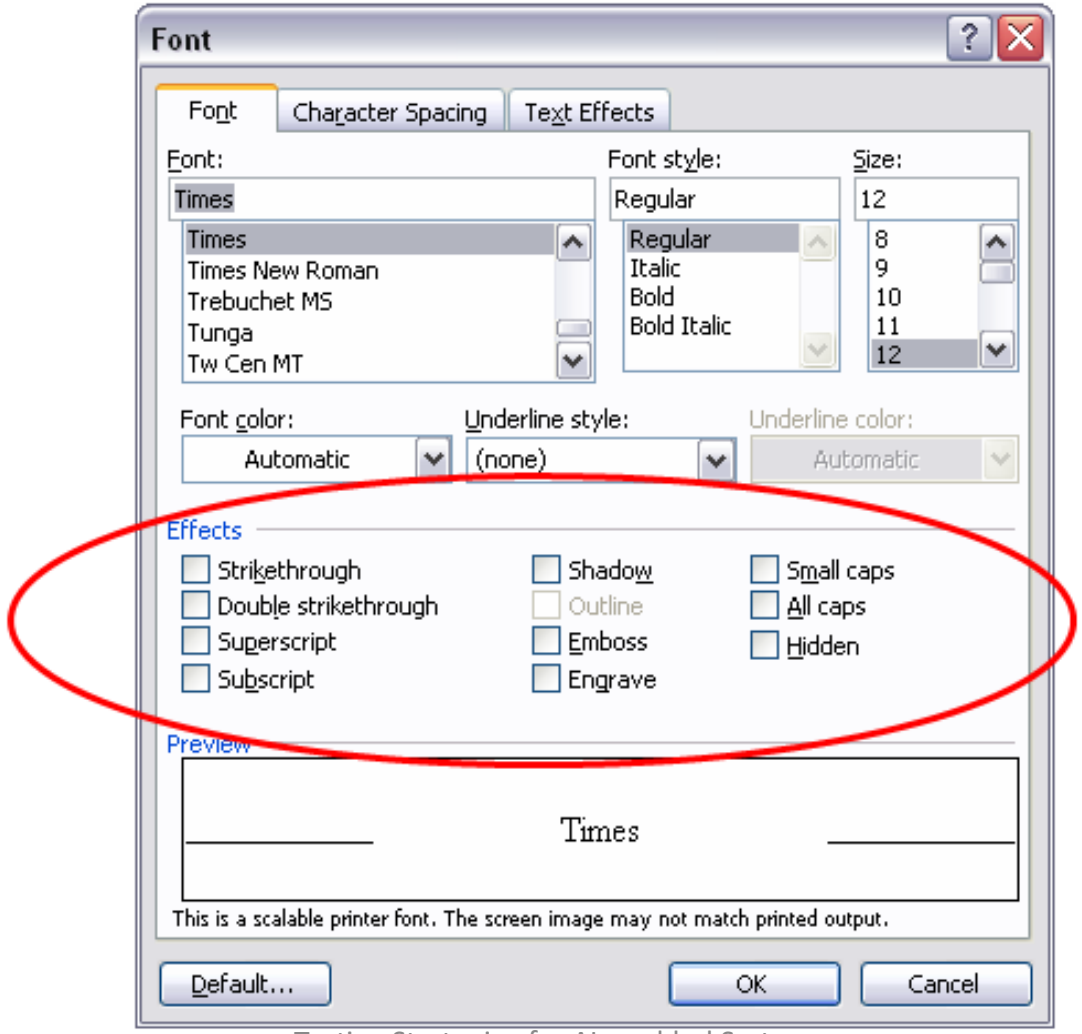
When all faults are triggered by the interaction of *t* or fewer variables, then testing all *t*-way combinations is *pseudo-exhaustive* and can provide strong assurance.

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

This discovery says we don't have to

(within reason; we still have value propagation issues, equivalence partitioning, timing issues, more complex interactions, . . .)

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



How Many Tests Would It Take?

- There are 10 effects, each can be on or off
- All combinations is 2^{10} = 1,024 tests
- What if our budget is too limited for these tests?
- Instead, let's look at all 3-way interactions …

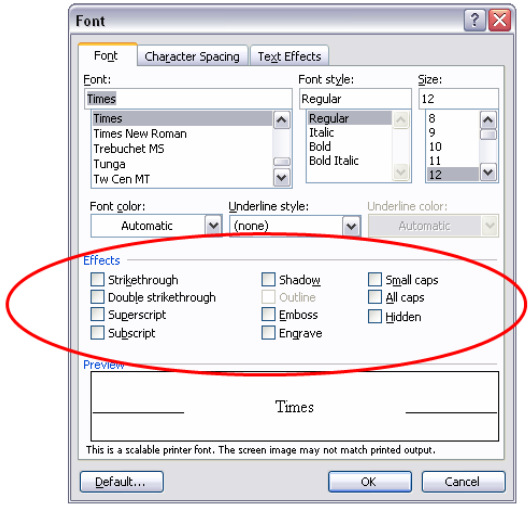
A covering array

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

A larger example

Suppose we have a system with 34 on-off switches.

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

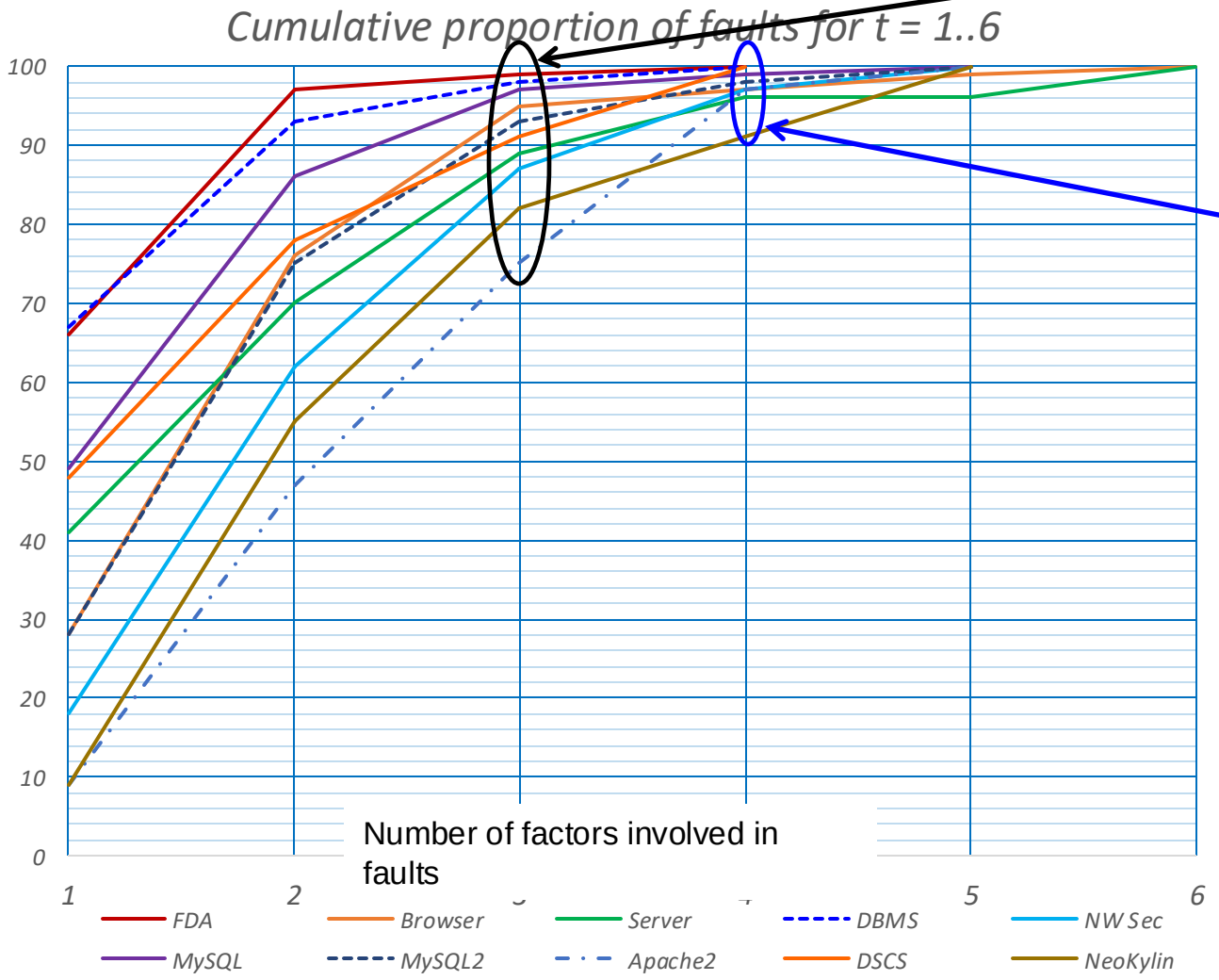
How do we test this?

34 switches = 2^{34} = 1.7 x 10^{10} possible inputs = 17 billion tests

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

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

Will this be effective testing?



33 tests for this (average) range of fault detection

85 tests for this (average) range of fault detection

Example: Cell phone app

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:

$3 \times 3 \times 4 \times 3 \times 5 \times 4 \times 4 \times 5 \times 4 = 3^3 4^4 5^2 = 172{,}800$

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

t	# tests	Fraction of 172,800 exhaustive
2	29	0.0002
3	137	0.0008
4	625	0.0040
5	2532	0.0150
6	9168	0.0530

Implications for testing:

- High strength testing can be done with a very small fraction of exhaustive tests
- Test set sizes can be practical for real-world testing

Example: abstract parameters – 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}

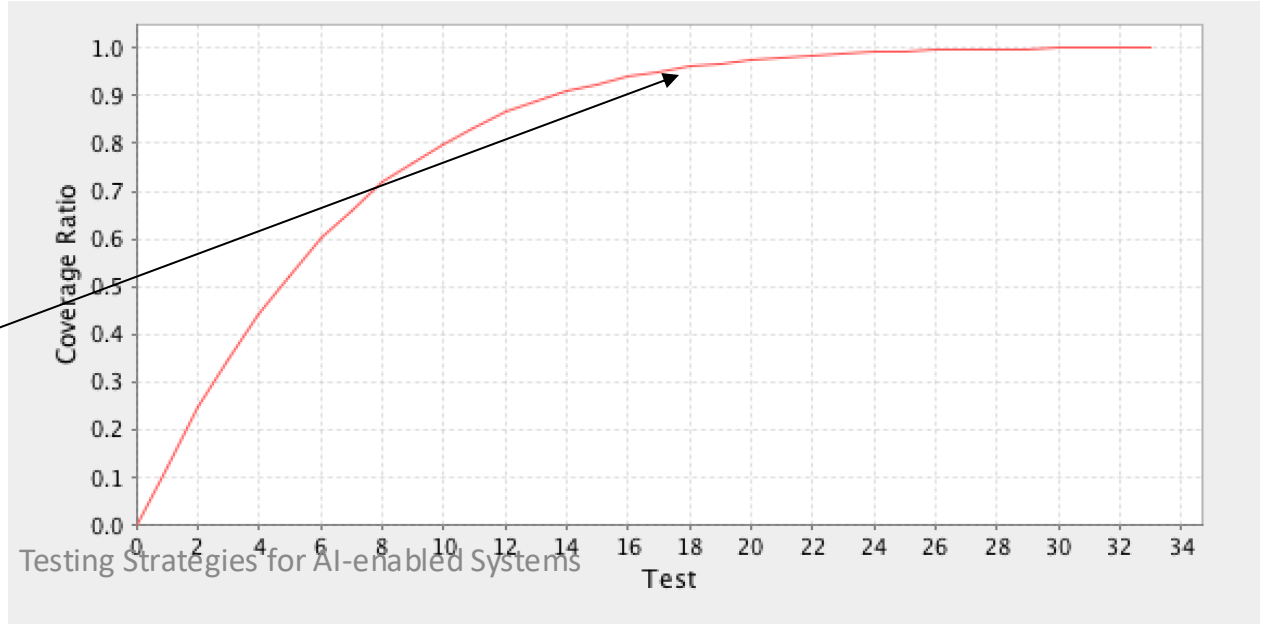
t	# tests	Fraction of $2^1 3^4 4^2$= 2,592 exhaustive
2	19	0.0073
3	67	0.0258
4	218	0.0841

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)

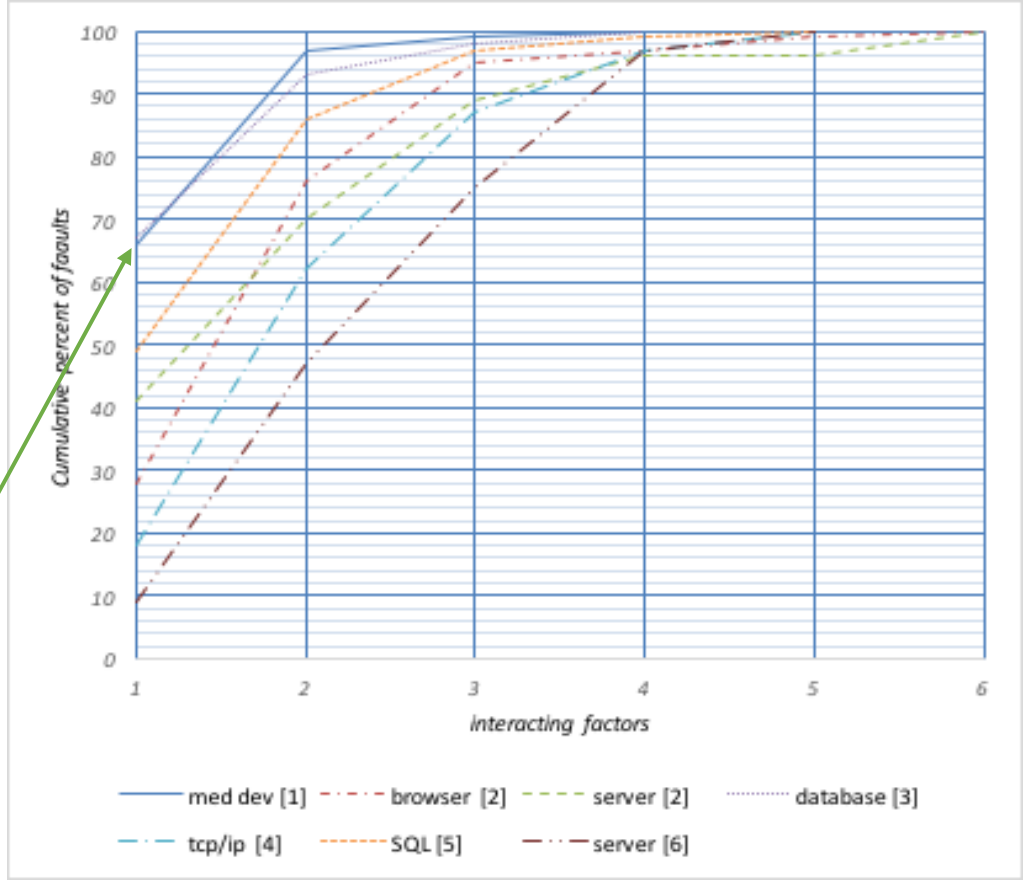
However:

- 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

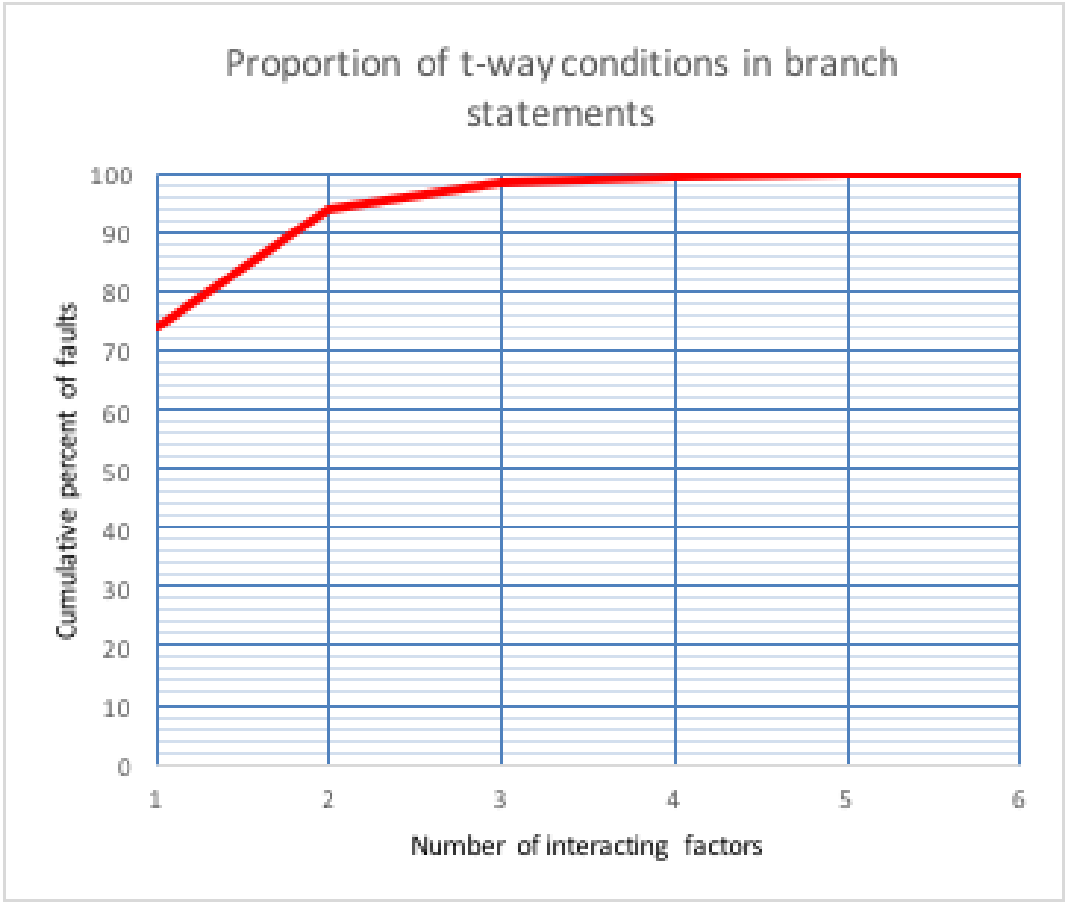


Covering Arrays and Reliability Models

- 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 fewer user applications (med devices) have simpler faults than heavily used applications (browser, server, SQL)



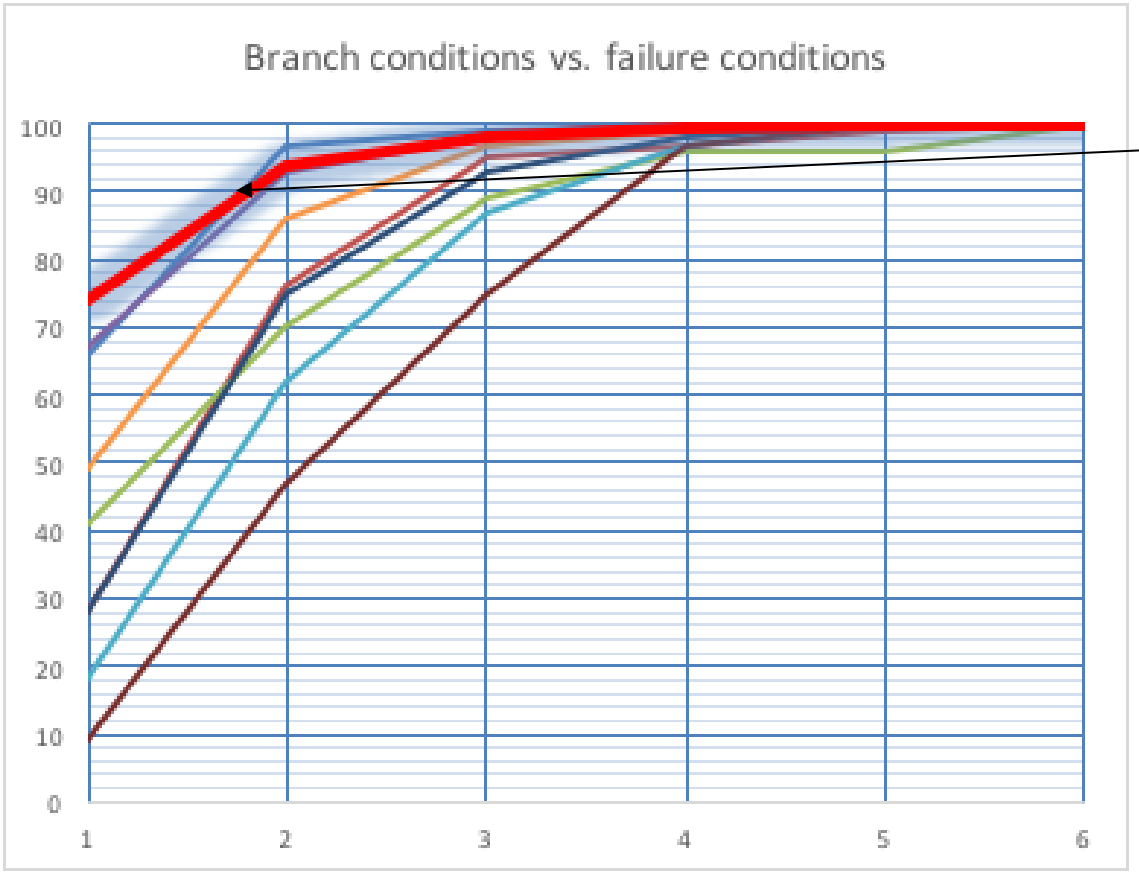
What causes this distribution?



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
- We have a mathematical model for this distribution



Why this distribution?

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

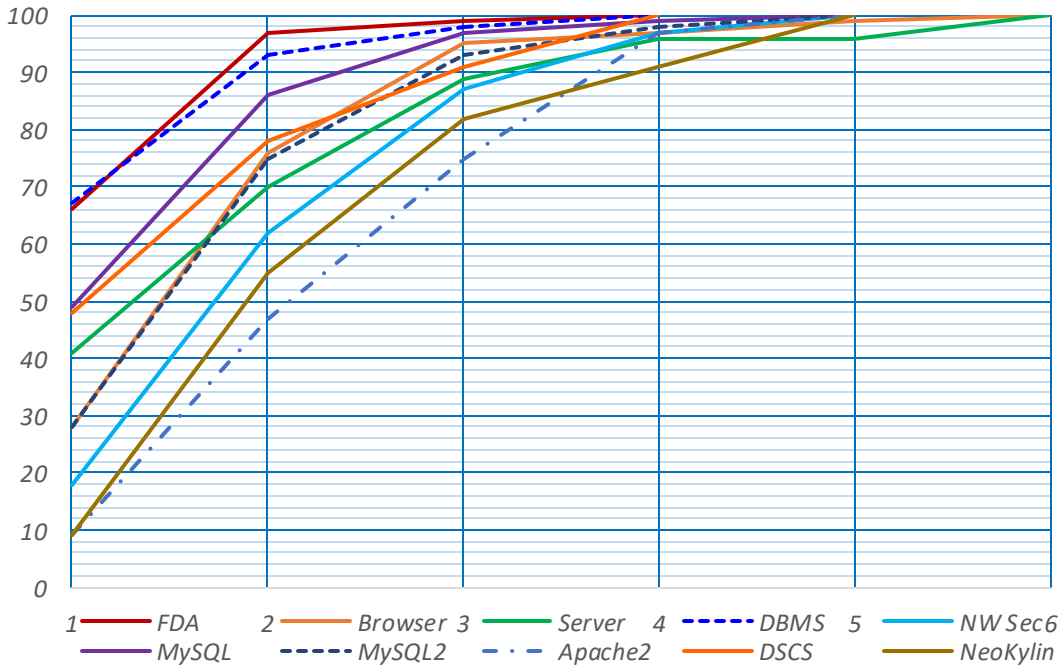
- Start with two assumptions:
  - *t*-way faults occur in proportion to *t*-way conditions in code
  - *t*-way faults are removed in proportion to *t*-way combinations in inputs

- Given these very basic assumptions, we can derive a reliability model that is equivalent to standard exponential model in reliability theory
- → note: equivalent model but starting from simpler assumptions

Fault distribution as testing progresses

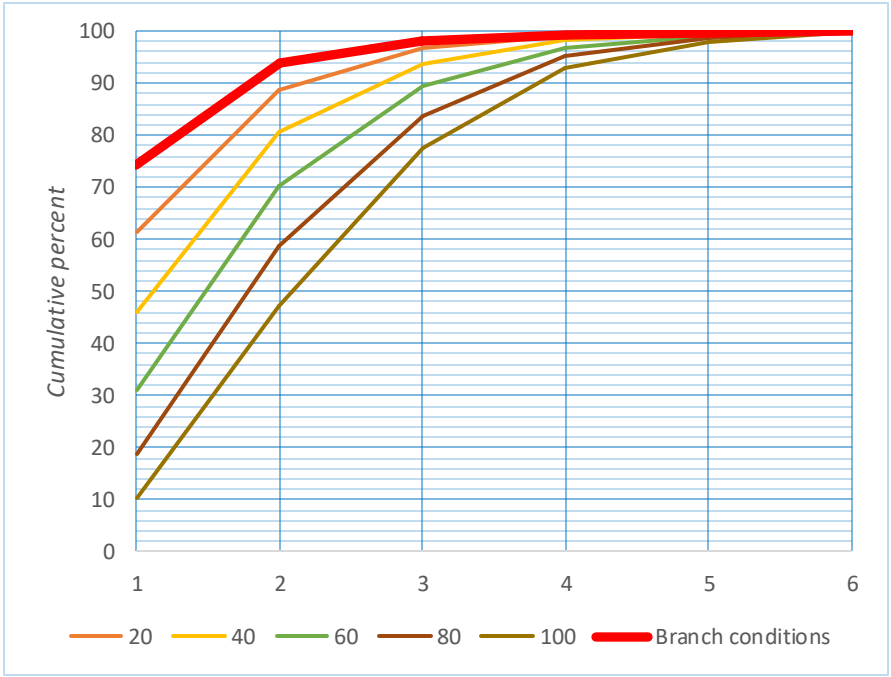
- for testing cycles, starting from distribution of branch conditions; curve moves down and to the right with more inputs/usage;
- close to empirical data

Empirical data



Empirical data

Model



Model

Implications for Testing

- *Uncertainty and variability in fault distribution:*
  greater at lower levels of *t* because they are found faster

- *Consistency with reliability growth models*:
  assumptions for the *t*-way fault distribution model lead to a basic exponential growth model equation at each level of *t*

- *Explains why 5-way, 6-way faults very rarely seen :*
  - w/ constant usage rate, time approx doubles with each increment in *t*
  - e.g., suppose it takes 2 months to find the 1-way faults
  - model predicts 16X time for 5-way, 32X time for 6-way
    => 2+ years for 5-way and 5+ years to find 6-way

Example: Subway control system

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

Original testing by company: 2 months

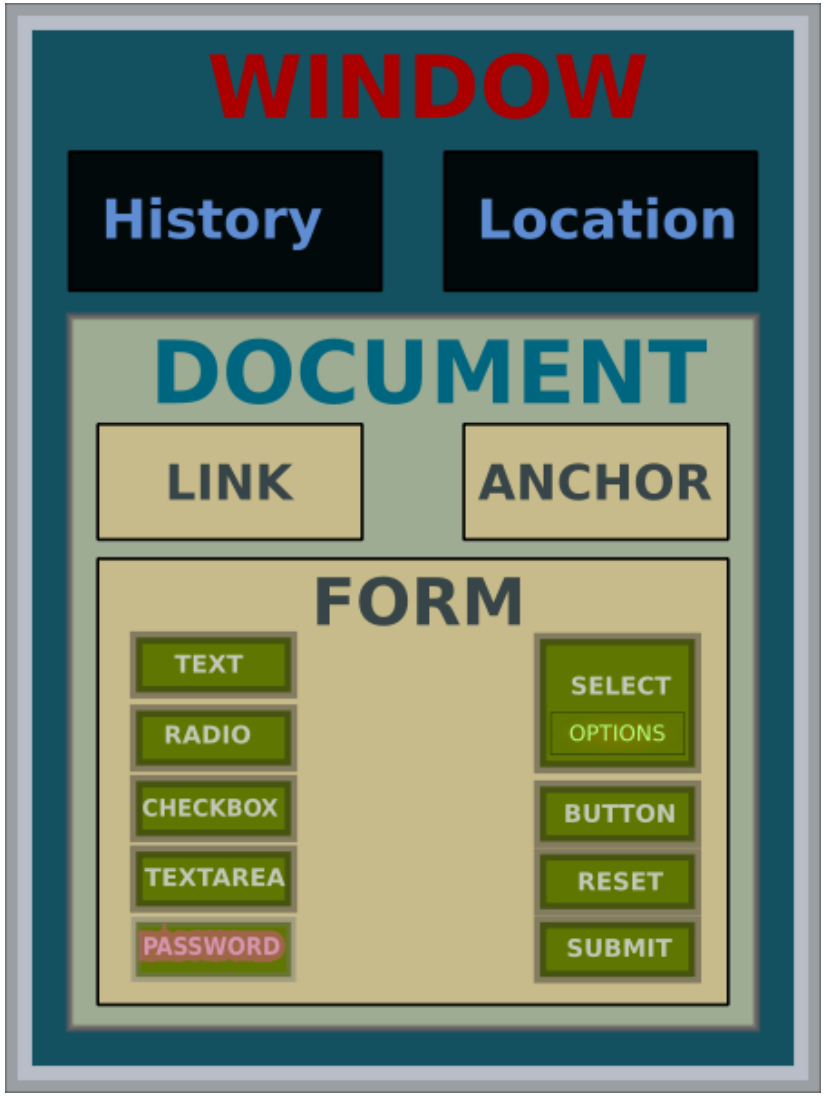
Combinatorial testing by U. Texas students: 2 weeks

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

Testing data

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

Example: Document Object Model Events



- 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
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
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

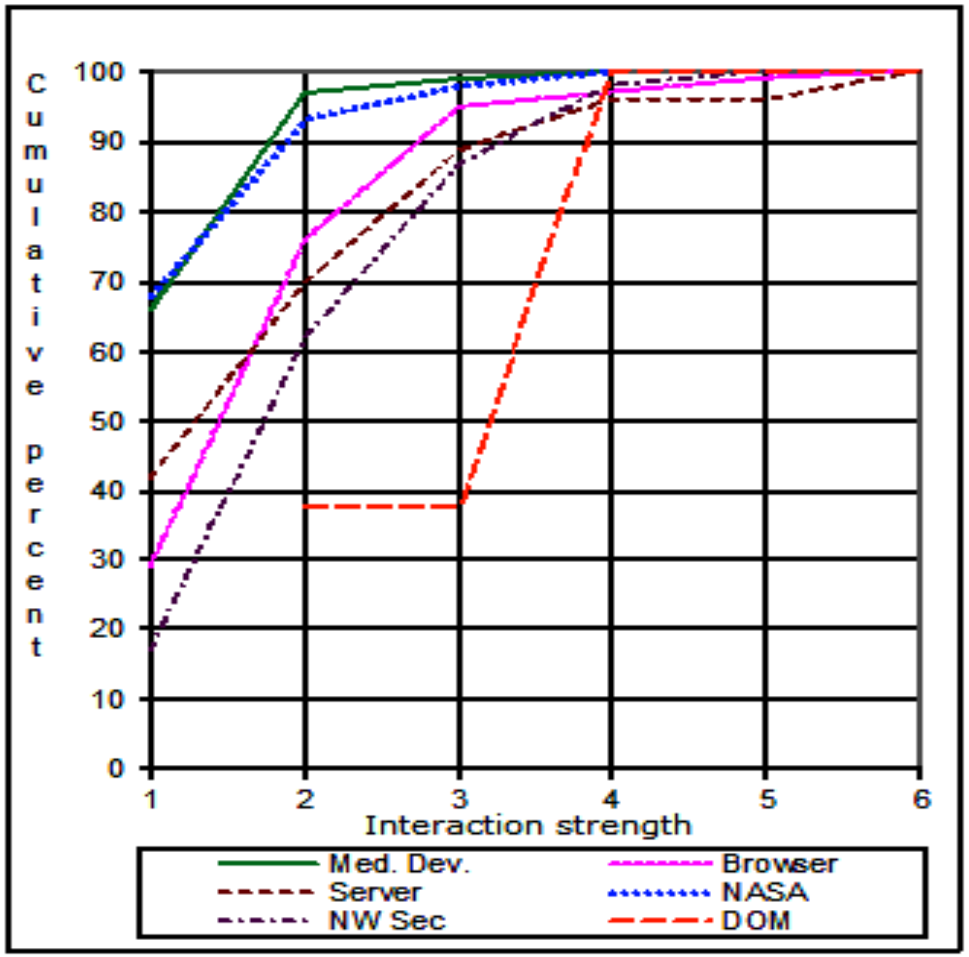
Exhaustive testing of equivalence class values

Document Object Model Events

Combinatorial test set:

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



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

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

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

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**

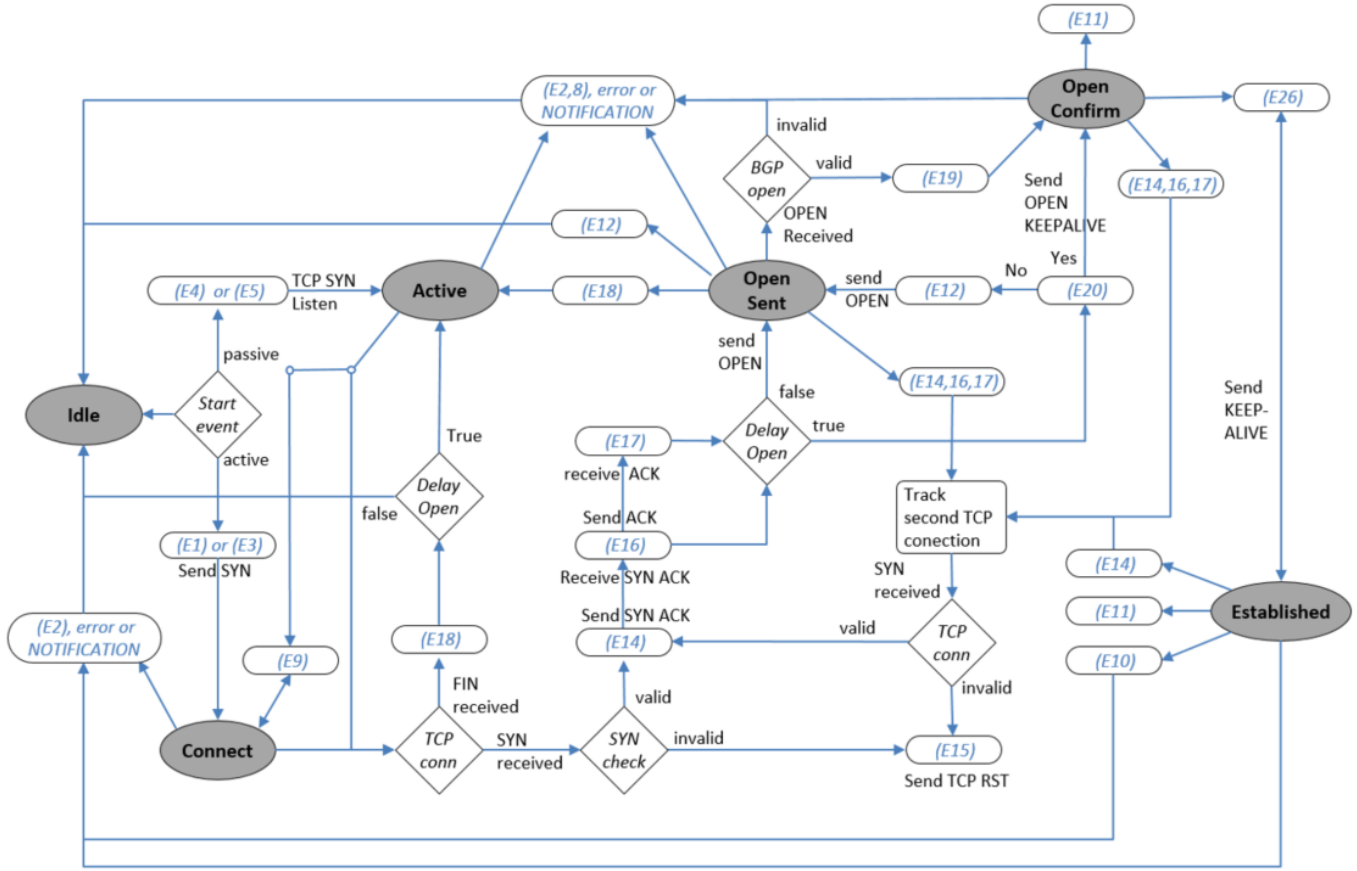
(because they are looking for it)

State changes

Covering arrays are highly effective

However – most *program behavior depends on more than input combinations*

How to test complex state machine?

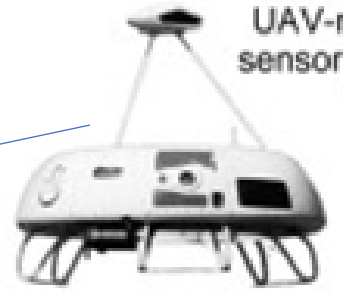


Response to inputs will change depending on ***inputs* and *system state***

How can we apply combinatorial methods to improve the testing of software that is not a pure function?

Sequences of Events

Problem: connect many peripherals, order of connection may affect application



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
a	connect range finder
b	connect telecom
c	connect satellite link
d	connect GPS
e	connect video
f	connect UAV

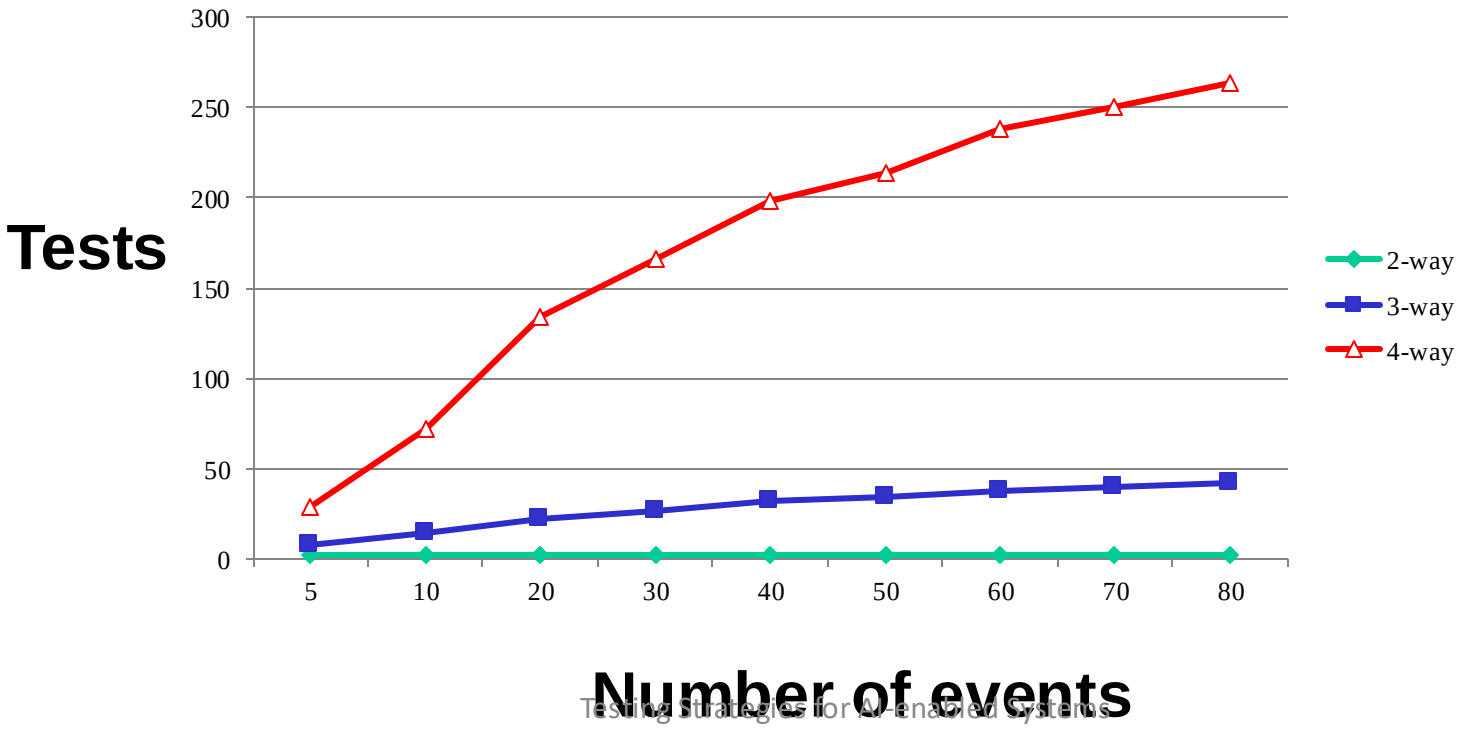
Sequence Covering Array

- 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
- Example: .*c.*f.*b.* covered. Any such 3-way seq covered.

Test	Sequence					
1	a	b	c	d	e	f
2	f	e	d	c	b	a
3	d	e	f	a	b	c
4	c	b	a	f	e	d
5	b	f	a	d	c	e
6	e	c	d	a	f	b
7	a	e	f	c	b	d
8	d	b	c	f	e	a
9	c	e	a	d	b	f
10	f	b	d	a	e	c

Event 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



Sequences of Input Combinations

- Covering arrays are great, but sequence of inputs can affect results when state is maintained by the system (nearly all)
- Sequence covering arrays handle sequences of events, but events may be complex and involve multiple parameters, combinations
- States change according to inputs, combinations of input values
- So we want to consider the order of inputs of combinations in test set

What are ordered combinations?

Test	p0	p1	p2	p3
1	**a**	d	**b**	**c**
2	b	a	c	d
3	b	d	c	a
4	c	a	b	d
5	c	d	b	a
6	d	a	c	b

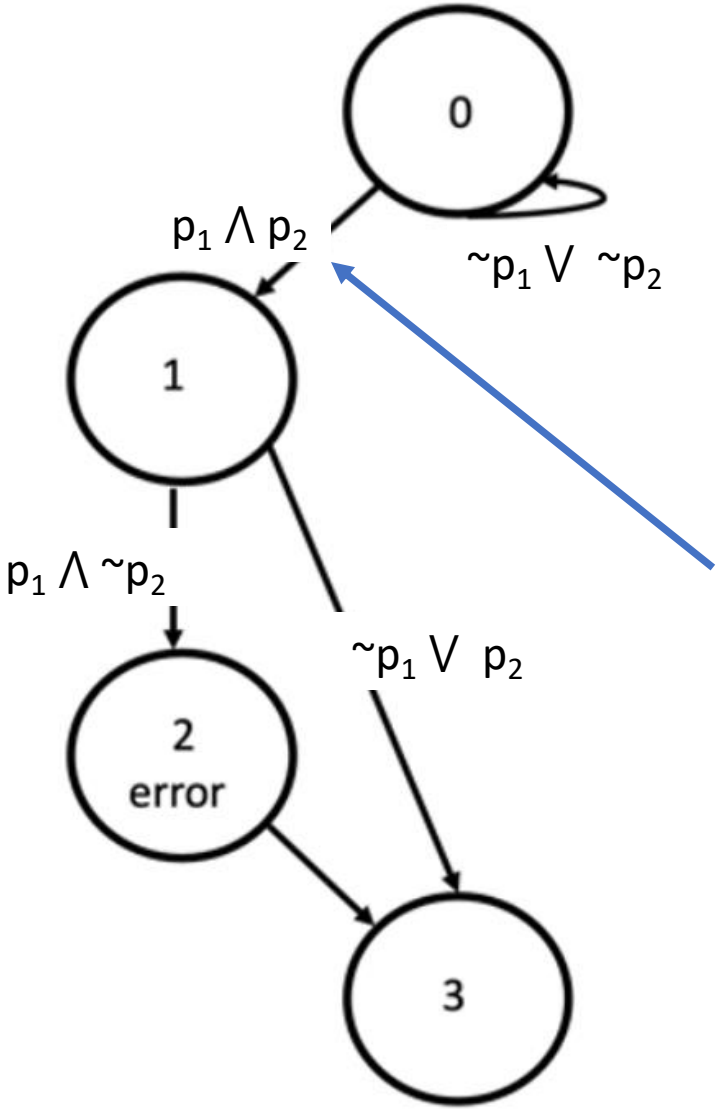
- Sequence covering, of events *a, b, c, d*
- Sequence covering array has all sequences of events for some specified length, non-repeating
- Each row is one test sequence

Test	p0	p1	p2	p3
1	**a**	**d**	b	c
2	b	a	c	d
3	**b**	d	c	**a**
4	c	a	b	d
5	c	d	b	a
6	d	**a**	c	**b**

Ordered combinations

- Combination order c_1 *→ c_2 *→ ...*→ c_s of *s* combinations of *t* parameter values, abbreviated *s*-order, is a set of *t*-way combinations in *s* rows
- Each row is one set of test inputs
- Ordering of combinations as rows entered sequentially
- Example: *p0p1* = *ad* *→ *p2p3* = *cb*

Order of covering array tests affects error detection



Test	p_1	p_2	p_3	p_4
1	0	0	1	0
2	0	1	0	1
3	1	0	0	1
4	1	1	1	0
5	0	1	0	0
6	0	0	1	1

Example:

$p_1 \Lambda p_2$

not followed by

$p_1 \Lambda \sim p_2$

Reordering tests to:

1
2
4
3
5
6

solves the problem

Ordered combination cover

- An ordered combination cover, designated OCC(*N, s, t, p, v*), covers all *s*-orders of *t*-way combinations of the *v* values of *p* parameters, where *t* is the number of parameters in combinations and *s* is the number of combinations in an ordered series.
- Number of combination order tuples to cover, for *s*-orders of *t*-way combinations of *p* parameters with *v* values each:

$$v^t C(p,t)^s$$

How can we find these ordered combination covers (OCC) efficiently?

Test	p0	p1	p2	p3
1	**a**	**d**	b	c
2	b	a	c	d
3	**b**	d	c	**a**
4	c	a	b	d
5	c	d	b	a
6	d	**a**	c	**b**

Generating ordered combination covers (OCC) ?

- The problem turns out to be easy!
- Theorem (OCC Coverage). *A test set covers s-orders of t-way combinations if and only if it includes an ordered series containing a total of s covering arrays, each of strength t.*

So,

1. make a *t*-way covering array
2. write *s* copies of it

Ordered coverage of adjacent combinations

- An adjacent combination order $c_1 \rightarrow c_2 \rightarrow \ldots \rightarrow c_s$ of *t*-way combinations, abbreviated *s*-order, is a set of *t*-way combinations in *s* consecutive rows.
- No interleaving between the ordered combinations, i.e.,
  Ordered: $c_1 \ ^*\!\rightarrow c_2$ c_1 is *eventually* followed by c_2
  Adjacent ordered : $c_1 \rightarrow c_2$ c_1 is *immediately* followed by c_2
- Ordered combinations with added constraint that rows are adjacent, i.e., for $c_1 \rightarrow c_2$ where c_1 and c_2 are in consecutive rows
- We need to produce an ordering of combinations such that every *t*-length permutation of combinations occurs as tests (rows) are input sequentially

This can be done with a deBruijn sequence

deBruijn sequences

- Studied in early 20th century, many properties proved by deBruijn
- For a given set *S* of *k* symbols, a deBruijn sequence D(*k*, *n*) includes every *n*-length permutation of the symbols in *S*
- length of a deBruijn sequence is k^n, and no shorter length sequence covering all the *n*-length permutations is possible
- Probably re-invented by every hacker on the planet (to crack key code locks)

D(3,2) = 00102112200102…

9 digits

key codes length 2: 18 digits

00,01,02,10,11,12,20,21,22

00102112 2

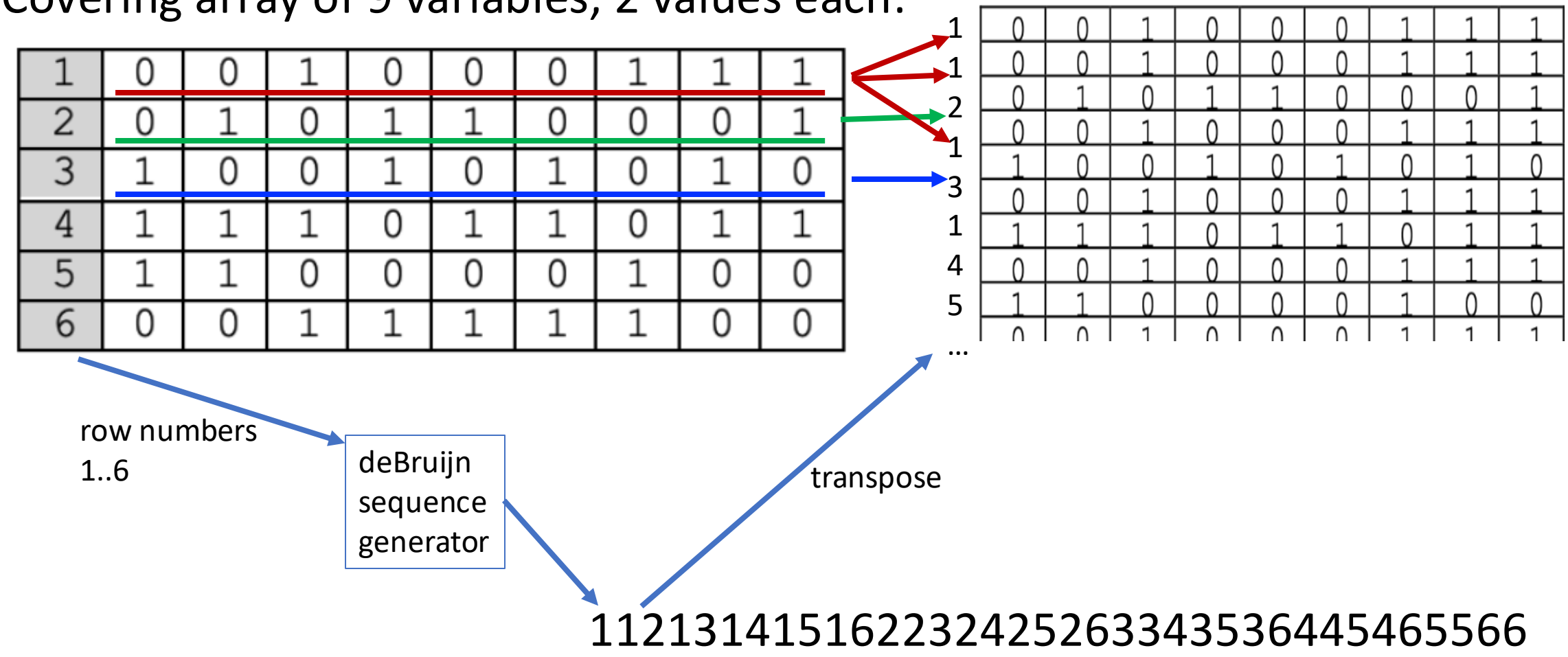
No ‘enter’ key:
628 key presses
instead of
3,125 for 4-digit code

Generating adjacent **ordered combination covers**

1. Generate a covering array of desired strength for the input model of the system under test.
2. Number the rows of the covering array sequentially, from 1 to *k*, for a covering array with *k* rows.
3. Generate a deBruijn sequence D(*k,s*) of the *k* row indices.
4. For each row index *i* in the sequence, write row *i* from the covering array. After the last row, append the initial *s* – 1 rows of the covering array, resulting in $N = k^s + s - 1$ rows.

Example

- Covering array of 9 variables, 2 values each:



Using adjacent ordered combinations

- Therac-25 example - radiation therapy machine fatal errors, 1985-1987 - widely known in software safety
- Multiple bugs and safety failures
- Critical, fatal race condition - error occurs if X-ray beam selected, changed to electron without min time between selections

Testing to detect error

p_1 = min time between option selections
p_2 = X-ray beam selected
p_3 = electron beam selected
p_4 = start beam

Test	p_1	p_2	p_3	p_4
t_1	1	1	0	0
t_2	1	0	1	0
t_3	1	0	0	1
t_4	0	0	1	0
t_5	0	1	0	1
...				
t_m	1	1	0	0
t_n	0	0	1	1
...				

Then, error only detected if test set contains a sequence of:

$p_1p_2p_3p_4$ = 1100 (X-ray beam selected)

followed by

$p_1p_2p_3p_4$ = 0011 (not min time so X-ray still on, electron selected & beam started)

OCCa guaranteed to contain this sequence.
Unlikely that other test set would, even if very large

Learning and Applying Combinatorial Methods

- Tutorial (English & Español) on using combinatorial testing for real-world software
- Key concepts and methods, explains use of software tools for combinatorial testing
- Costs and practical considerations
- Advanced topics such as the use of formal models and test oracle generation
- Designed for testers or undergraduate students of computer science or engineering

t-way to higher strength

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

a 3-way combination: **abc**
contains three 2-way combinations:
ab ac bc

So 10 *3-way combinations* will be
30 *2-way combinations*

Another way of looking at this:
30 2-way combinations only gives us
10 3-way combinations