
Rick Kuhn, Dimitris E. Simos and Raghu Kacker
National Institute of Standards and Technology, USA
SBA Research, Austria
Overview

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** – **improve cost-benefit ratio for testing**
  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?

**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

![Graph showing 65% of faults caused by single factor and 96% caused by single factor or 2-way interactions.](image)
These faults more complex than medical device software!!

Why?
Curves appear to be similar across a variety of application domains.
NASA distributed database

Note: initial testing but ....
Fault profile better than medical devices!
Wait, there’s more

Cumulative proportion of faults for $t = 1..6$

- 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

• 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?

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, . . . )
Overview

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

<table>
<thead>
<tr>
<th>Test</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Each combination occurs same number of times

Example: P1, P2 = 1,2
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
   Brownlie et al (1992) AT&T

Generation of test suites using OAs
   OATS (Phadke, AT&T-BL)
What’s different about software?

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 **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**

<table>
<thead>
<tr>
<th><strong>orthogonal arrays:</strong></th>
<th><strong>covering arrays:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Combinations occur same number of times</td>
<td>• Combinations occur at least once</td>
</tr>
<tr>
<td>• Not always possible to find for a particular configuration</td>
<td>• <strong>Always possible to find</strong> for a particular configuration</td>
</tr>
<tr>
<td></td>
<td>• Size always $\leq$ orthogonal array</td>
</tr>
</tbody>
</table>
Let’s see how to use this in testing.
A simple example:

- 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 ...
How Many Tests Do We Need?

- There are $\binom{10}{3} = 120$ 3-way interactions.
- Each triple has $2^3 = 8$ settings: 000, 001, 010, 011, ...
- $120 \times 8 = 960$ combinations
- Each test exercises many triples:

0 1 1 0 0 0 0 1 1 0

OK, OK, what’s the smallest number of tests we need?
# A covering array of 13 tests

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

Each row is a test:

```
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 0 | 1 | 1 |
| 0 | 1 | 0 |
```

Each column is a parameter:

- Developed 1990s
- Extends Design of Experiments concept
- hard optimization problem but good algorithms now
Larger example - testing inputs, combinations of variable values

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 \( = 2^{34} = 1.7 \times 10^{10} \) 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
Will this be effective testing?

Cumulative proportion of faults for $t = 1..6$

- 33 tests for this (average) range of fault detection
- 85 tests for this (average) range of fault detection

That’s way better than 17 billion!
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)

<table>
<thead>
<tr>
<th>T-Way</th>
<th>NIST ACTS</th>
<th>ITCH (IBM)</th>
<th>Jenny (Open Source)</th>
<th>TConfig (U. Ottawa)</th>
<th>TVG (Open Source)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Time</td>
<td>Size</td>
<td>Time</td>
<td>Size</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>0.8</td>
<td>120</td>
<td>0.73</td>
<td>108</td>
</tr>
<tr>
<td>3</td>
<td>400</td>
<td>0.36</td>
<td>2388</td>
<td>1020</td>
<td>413</td>
</tr>
<tr>
<td>4</td>
<td>1363</td>
<td>3.05</td>
<td>1484</td>
<td>5400</td>
<td>1536</td>
</tr>
<tr>
<td>5</td>
<td>4226</td>
<td>18s</td>
<td>NA</td>
<td>&gt;1 day</td>
<td>4580</td>
</tr>
<tr>
<td>6</td>
<td>10941</td>
<td>65.03</td>
<td>NA</td>
<td>&gt;1 day</td>
<td>11625</td>
</tr>
</tbody>
</table>

Times in seconds

Traffic Collision Avoidance System (TCAS): $2^73^24^110^2$
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

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Baseline</th>
<th>Simultaneous</th>
<th>Skip</th>
<th>Partitioned</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity Reduction</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Skip fully covered combinations</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Search space pruning</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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 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 $2^13^44^2 = 2592$ 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:

```java
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

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

### Total possible configurations:

\[3 \times 3 \times 4 \times 3 \times 5 \times 4 \times 4 \times 5 \times 4 = 172,800\]
Number of configurations generated for $t$-way interaction testing, $t = 2..6$

<table>
<thead>
<tr>
<th>$t$</th>
<th># Configs</th>
<th>% of Exhaustive</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>29</td>
<td>0.02</td>
</tr>
<tr>
<td>3</td>
<td>137</td>
<td>0.08</td>
</tr>
<tr>
<td>4</td>
<td>625</td>
<td>0.4</td>
</tr>
<tr>
<td>5</td>
<td>2532</td>
<td>1.5</td>
</tr>
<tr>
<td>6</td>
<td>9168</td>
<td>5.3</td>
</tr>
</tbody>
</table>
ACTS - Defining a new system
Variable interaction strength

![New System Form](image)

- **Parameters**
  - Cur_Vertical_Sep
  - High_Confidence
  - Two_of_Three_Reports
  - Own_Tracked_Alt
  - Other_Track_Alt
  - Own_Tracked_Alt_Rate
  - Alt_Layer_Value
  - Up_Separation
  - Down_Separation
  - Other_RAC
  - Other_Capability
  - Climb_Inhibit

- **Strength**
  - 4

- **Parameter Names and Strength**
  - Cur_Vertical_Sep, High_Confidence, Two_of_Three_Reports, 2
  - Alt_Layer_Value, Up_Separation, Down_Separation, 3
Constraints
Covering array output
## Output options

### Mappable values

<table>
<thead>
<tr>
<th>Degree of interaction coverage: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of parameters: 12</td>
</tr>
<tr>
<td>Number of tests: 100</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Configuration #1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cur_Vertical_Sep=299</td>
</tr>
<tr>
<td>High_Confidence=true</td>
</tr>
<tr>
<td>Two_of_Three_Reports=true</td>
</tr>
<tr>
<td>Own_Tracked_Alt=1</td>
</tr>
<tr>
<td>Other_Tracked_Alt=1</td>
</tr>
<tr>
<td>Own_Tracked_Alt_Rate=600</td>
</tr>
<tr>
<td>Alt_Layer_Value=0</td>
</tr>
<tr>
<td>Up_Separation=0</td>
</tr>
<tr>
<td>Down_Separation=0</td>
</tr>
<tr>
<td>Other_RAC=NO_INTENT</td>
</tr>
<tr>
<td>Other_Capability=TCAS_CA</td>
</tr>
<tr>
<td>Climb_Inhibit=true</td>
</tr>
</tbody>
</table>

Etc.
### CAGen: A FIPO webUI tool

#### Input Parameter Model

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>scenario</td>
<td>a, b, c</td>
<td>3</td>
</tr>
<tr>
<td>protocol</td>
<td>tls, ssl, dtls</td>
<td>3</td>
</tr>
<tr>
<td>authenticate</td>
<td>true, false</td>
<td>2</td>
</tr>
<tr>
<td>retries</td>
<td>0, 1, 2, 3, 4</td>
<td>5</td>
</tr>
<tr>
<td>payload</td>
<td>1, 2, 3, 4, 5, 6</td>
<td>6</td>
</tr>
<tr>
<td>implementation</td>
<td>OPEN_SSL, GNU_TLS</td>
<td>2</td>
</tr>
</tbody>
</table>
# CAGen: Array Generation

## Array Generation

### Algorithm: FIPOG

<table>
<thead>
<tr>
<th>scenario</th>
<th>protocol</th>
<th>authenticate</th>
<th>retries</th>
<th>payload</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>tls</td>
<td>true</td>
<td>0</td>
<td>1</td>
<td>OPEN_SSL</td>
</tr>
<tr>
<td>b</td>
<td>ssl</td>
<td>false</td>
<td>1</td>
<td>2</td>
<td>GNU_TLS</td>
</tr>
<tr>
<td>c</td>
<td>dtls</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Showing rows 1-6
Available Tools

• **Covering array generator** – 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

- Information Technology
- Finance
- Defense
- Telecom
Overview

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

<table>
<thead>
<tr>
<th>Package</th>
<th>Number of test cases</th>
<th>Number of bugs found</th>
<th>Did CT find all original bugs?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package 1</td>
<td>Original 98</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>CT</td>
<td>49</td>
<td>6</td>
<td>Yes</td>
</tr>
<tr>
<td>Package 2</td>
<td>Original 102</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>CT</td>
<td>77</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>Package 3</td>
<td>Original 116</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>CT</td>
<td>80</td>
<td>7</td>
<td>Miss 1</td>
</tr>
<tr>
<td>Package 4</td>
<td>Original 122</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>CT</td>
<td>90</td>
<td>4</td>
<td>Yes</td>
</tr>
</tbody>
</table>
IoT example – smart house home assistant
Configuration testing for an IoT device

```python
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

<table>
<thead>
<tr>
<th>switch00kitchen_lights</th>
<th>automation00music_mode</th>
<th>media_player00sound_system</th>
<th>group00living_space</th>
<th>switch00living_room_lights</th>
</tr>
</thead>
<tbody>
<tr>
<td>turn_off</td>
<td>trigger</td>
<td>clear_playlist</td>
<td>remove</td>
<td>turn_off</td>
</tr>
</tbody>
</table>

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

- ![kitchen lights icon](image1)

- ![kitchen lights icon](image2)

- ![kitchen lights icon](image3)

- **switch.kitchen_lights**  
  - on  
  - friendly_name: kitchen lights  
  - assumed_state: true
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:

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Param.</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abort</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Blur</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>Click</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>Change</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>dblClick</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>DOMActivate</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>DOMAttrModified</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>DOMCharacterDataModified</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>DOMElementNameChanged</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>DOMFocusIn</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>DOMFocusOut</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>DOMNodeInserted</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>DOMNodeInsertedIntoDocument</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>DOMNodeRemoved</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>DOMNodeRemovedFromDocument</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>DOMSubTreeModified</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>Error</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Focus</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>KeyDown</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>KeyUp</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>Load</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>MouseDown</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>MouseMove</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>MouseOut</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>MouseOver</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>MouseUp</td>
<td>15</td>
<td>4352</td>
</tr>
<tr>
<td>MouseWheel</td>
<td>14</td>
<td>1024</td>
</tr>
<tr>
<td>Reset</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Resize</td>
<td>5</td>
<td>48</td>
</tr>
<tr>
<td>Scroll</td>
<td>5</td>
<td>48</td>
</tr>
<tr>
<td>Select</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Submit</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>TextInput</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Unload</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>Wheel</td>
<td>15</td>
<td>4096</td>
</tr>
<tr>
<td>Total Tests</td>
<td></td>
<td>36626</td>
</tr>
</tbody>
</table>

Exhaustive testing of equivalence class values
## Document Object Model Events

### Combinatorial test set:

<table>
<thead>
<tr>
<th>t</th>
<th>Tests</th>
<th>% of Orig.</th>
<th>Test Results</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>702</td>
<td>1.92%</td>
<td>202</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1342</td>
<td>3.67%</td>
<td>786</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1818</td>
<td>4.96%</td>
<td>437</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2742</td>
<td>7.49%</td>
<td>908</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4227</td>
<td>11.54%</td>
<td>1803</td>
<td>72</td>
<td></td>
</tr>
</tbody>
</table>

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

LANTIRN = Low Altitude Navigation & Targeting Infrared for Night

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aircraft</td>
<td>15, 40</td>
</tr>
<tr>
<td>Altitude</td>
<td>5k, 10k, 15k, 20k, 30k, 40k, 50k</td>
</tr>
<tr>
<td>Maneuver</td>
<td>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, 360 nose roll</td>
</tr>
<tr>
<td>Mach (100th)</td>
<td>40, 50, 60, 70, 80, 90, 100, 110, 120</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 DIMENSIONS</td>
<td>1,2,4,6,8</td>
</tr>
<tr>
<td>2 NODOSDIM</td>
<td>2,4,6</td>
</tr>
<tr>
<td>3 NUMVIRT</td>
<td>1,2,3,8</td>
</tr>
<tr>
<td>4 NUMVIRTINJ</td>
<td>1,2,3,8</td>
</tr>
<tr>
<td>5 NUMVIRTEJE</td>
<td>1,2,3,8</td>
</tr>
<tr>
<td>6 LONBUFFER</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>7 NUMDIR</td>
<td>1,2</td>
</tr>
<tr>
<td>8 FORWARDING</td>
<td>0,1</td>
</tr>
<tr>
<td>9 PHYSICAL</td>
<td>true, false</td>
</tr>
<tr>
<td>10 ROUTING</td>
<td>0,1,2,3</td>
</tr>
<tr>
<td>11 DELFIFO</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>12 DELCROSS</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>13 DELCHANNEL</td>
<td>1,2,4,6</td>
</tr>
<tr>
<td>14 DELSWITCH</td>
<td>1,2,4,6</td>
</tr>
</tbody>
</table>

5x3x4x4x4x4x2x2x2x4x4x4x4x4 = 31,457,280 configurations

Are any of them dangerous? If so, how many? Which ones?
# Network Deadlock Detection

## Deadlocks Detected: combinatorial

<table>
<thead>
<tr>
<th>t</th>
<th>Tests</th>
<th>500 pkts</th>
<th>1000 pkts</th>
<th>2000 pkts</th>
<th>4000 pkts</th>
<th>8000 pkts</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>161</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>752</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

## Average Deadlocks Detected: random

<table>
<thead>
<tr>
<th>t</th>
<th>Tests</th>
<th>500 pkts</th>
<th>1000 pkts</th>
<th>2000 pkts</th>
<th>4000 pkts</th>
<th>8000 pkts</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>28</td>
<td>0.63</td>
<td>0.25</td>
<td>0.75</td>
<td>0.50</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>161</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>752</td>
<td>10.13</td>
<td>11.75</td>
<td>10.38</td>
<td>13</td>
<td>13.25</td>
</tr>
</tbody>
</table>
Network Deadlock Detection

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

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)
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)?

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>connect range finder</td>
</tr>
<tr>
<td>b</td>
<td>connect telecom</td>
</tr>
<tr>
<td>c</td>
<td>connect satellite link</td>
</tr>
<tr>
<td>d</td>
<td>connect GPS</td>
</tr>
<tr>
<td>e</td>
<td>connect video</td>
</tr>
<tr>
<td>f</td>
<td>connect UAV</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Test</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a b c d e f</td>
</tr>
<tr>
<td>2</td>
<td>f e d c b a</td>
</tr>
<tr>
<td>3</td>
<td>d e f a b c</td>
</tr>
<tr>
<td>4</td>
<td>c b a f e d</td>
</tr>
<tr>
<td>5</td>
<td>b f a d c e</td>
</tr>
<tr>
<td>6</td>
<td>e c d a f b</td>
</tr>
<tr>
<td>7</td>
<td>a e f c b d</td>
</tr>
<tr>
<td>8</td>
<td>d b c f e a</td>
</tr>
<tr>
<td>9</td>
<td>c e a d b f</td>
</tr>
<tr>
<td>10</td>
<td>f b d a e c</td>
</tr>
</tbody>
</table>
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

Test coverage has traditionally been defined using graph-based structural coverage criteria:

- statement (weak)
- branch (better)
- etc.

Based on paths through the code

Subsumption relationships of structural coverage criteria

What about the data?
<table>
<thead>
<tr>
<th>Tests</th>
<th>Variables</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable pairs</th>
<th>Variable-value combinations covered</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab</td>
<td>00, 01, 10</td>
<td>.75</td>
</tr>
<tr>
<td>ac</td>
<td>00, 01, 10</td>
<td>.75</td>
</tr>
<tr>
<td>ad</td>
<td>00, 01, 11</td>
<td>.75</td>
</tr>
<tr>
<td>bc</td>
<td>00, 11</td>
<td>.50</td>
</tr>
<tr>
<td>bd</td>
<td>00, 01, 10, 11</td>
<td>1.0</td>
</tr>
<tr>
<td>cd</td>
<td>00, 01, 10, 11</td>
<td>1.0</td>
</tr>
</tbody>
</table>

100% coverage of 33% of combinations
75% coverage of half of combinations
50% coverage of 16% of combinations
<table>
<thead>
<tr>
<th>Variable pairs</th>
<th>Variable-value combinations covered</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab</td>
<td>00, 01, 10</td>
<td>.75</td>
</tr>
<tr>
<td>ac</td>
<td>00, 01, 10</td>
<td>.75</td>
</tr>
<tr>
<td>ad</td>
<td>00, 01, 11</td>
<td>.75</td>
</tr>
<tr>
<td>bc</td>
<td>00, 11</td>
<td>.50</td>
</tr>
<tr>
<td>bd</td>
<td>00, 01, 10, 11</td>
<td>1.0</td>
</tr>
<tr>
<td>cd</td>
<td>00, 01, 10, 11</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Rearranging the table

```plaintext
bd | 00, 01, 10, 11
cd | 00, 01, 10, 11
ab | 00, 01, 10
ac | 00, 01, 10
ad | 00, 01, 11
bc | 00, 11
```
Graphing Coverage Measurement

Bottom line:
All combinations covered to at least 50%
What else does this chart show?

Tested combinations => code works for these

Untested combinations (look for problems here)
Spacecraft software example
82 variables, 7,489 tests, conventional test design
(not covering arrays)
# Additional coverage metrics

## Relative Coverage Gain per Test

<table>
<thead>
<tr>
<th>Class</th>
<th>Race</th>
<th>Weapon</th>
<th>Shield</th>
<th>Armor</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Thief</td>
<td>Sword</td>
<td>0</td>
<td>Light</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Mage</td>
<td>Sword</td>
<td>1</td>
<td>Heavy</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Warrior</td>
<td>Sword</td>
<td>0</td>
<td>Heavy</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Thief</td>
<td>Sword</td>
<td>1</td>
<td>Light</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Mage</td>
<td>Sword</td>
<td>0</td>
<td>Light</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Warrior</td>
<td>Sword</td>
<td>1</td>
<td>Heavy</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>Thief</td>
<td>Sword</td>
<td>0</td>
<td>Heavy</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>Mage</td>
<td>Sword</td>
<td>1</td>
<td>Light</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>Warrior</td>
<td>Sword</td>
<td>0</td>
<td>Light</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>Thief</td>
<td>Sword</td>
<td>1</td>
<td>Heavy</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>Mage</td>
<td>Sword</td>
<td>0</td>
<td>Light</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>Warrior</td>
<td>Sword</td>
<td>1</td>
<td>Light</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>Thief</td>
<td>Wabbajack</td>
<td>1</td>
<td>Heavy</td>
<td>8</td>
</tr>
</tbody>
</table>
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?

- **Input model** 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?
  - **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
Using model checking to produce tests

The system can never get in this state!

Yes it can, and here’s how …

- 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
## Tests generated

<table>
<thead>
<tr>
<th>$t$</th>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-way:</td>
<td>156</td>
</tr>
<tr>
<td>3-way:</td>
<td>461</td>
</tr>
<tr>
<td>4-way:</td>
<td>1,450</td>
</tr>
<tr>
<td>5-way:</td>
<td>4,309</td>
</tr>
<tr>
<td>6-way:</td>
<td>11,094</td>
</tr>
</tbody>
</table>

![Bar chart showing the number of tests generated for different $t$ values.](chart.png)
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
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
Overview

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:
Sample of XSS and SQLi vulnerabilities found

W3C

Tidy your HTML

An error (2/0 error: 403 Access to url 'http://example.com' was blocked by the W3C Tidy (DOM)

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](https://example.com/tidy.py).

script $Revision: 1.22 $ of $Date: 2013-10-21 12:13:39 $ by [Dan Connolly](mailto:dc@w3.org)

Further developed and maintained by [Dominique Hazael-Ma�iaux](mailto:dhazael@w3.org)
Security Protocol Testing

- Input universe
  - Well-formed inputs
    - Invalid inputs
  - Security protocol implementation
    - Abnormal behaviors
      - Behaviors
        - May expose vulnerabilities
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

<table>
<thead>
<tr>
<th>version</th>
<th>hash</th>
<th>key</th>
<th>signature</th>
<th>active</th>
<th>critical</th>
<th>is_authority</th>
<th>pathlen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>md5</td>
<td>dsa</td>
<td>self</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>sha1</td>
<td>rsa</td>
<td>unrelated</td>
<td>false</td>
<td>dummy</td>
<td>dummy</td>
<td>dummy</td>
</tr>
<tr>
<td>0</td>
<td>sha256</td>
<td>dsa</td>
<td>parent</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>md5</td>
<td>rsa</td>
<td>unrelated</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>sha1</td>
<td>rsa</td>
<td>parent</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>sha256</td>
<td>dsa</td>
<td>self</td>
<td>false</td>
<td>dummy</td>
<td>dummy</td>
<td>dummy</td>
</tr>
<tr>
<td>2</td>
<td>md5</td>
<td>rsa</td>
<td>parent</td>
<td>false</td>
<td>dummy</td>
<td>dummy</td>
<td>dummy</td>
</tr>
<tr>
<td>2</td>
<td>sha1</td>
<td>dsa</td>
<td>self</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>sha256</td>
<td>rsa</td>
<td>unrelated</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>md5</td>
<td>dsa</td>
<td>unrelated</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>sha1</td>
<td>dsa</td>
<td>parent</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>sha256</td>
<td>rsa</td>
<td>self</td>
<td>false</td>
<td>dummy</td>
<td>dummy</td>
<td>dummy</td>
</tr>
</tbody>
</table>
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

![Bar charts showing errors observed for TLS implementations]

<table>
<thead>
<tr>
<th>Error</th>
<th>BouncyCastle</th>
<th>wolfSSL</th>
<th>GnuTLS</th>
<th>NSS</th>
<th>OpenJDK</th>
<th>OpenSSL</th>
<th>mbed</th>
</tr>
</thead>
<tbody>
<tr>
<td>untrusted</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>expired or not yet valid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>parse-error</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>crash</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>use of insecure algorithm</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>invalid signature</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>unknown critical extension</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>extension in non-v3 cert</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>use of weak key</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>name constraint violation</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>key usage not allowed</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
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

Complete test sequence set: $S$ with $|S| = 1956$

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\}
# Recommendations on TLS cipher suites

<table>
<thead>
<tr>
<th>Organization</th>
<th>Cipher Suite Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td>IETF</td>
<td>The registry contained in early 2016 more than 300 named cipher suites. There are 28 cryptographic algorithms for the authenticated key exchange, 25 for the encryption part and five for the MAC.</td>
</tr>
<tr>
<td>Mozilla</td>
<td>22 TLS cipher suites for hardened configurations of server-side implementations.</td>
</tr>
<tr>
<td>BSI</td>
<td>Suggest the use of TLS v1.2 with 16 cipher suites.</td>
</tr>
<tr>
<td>ENISA</td>
<td>Commissioned study suggests to use version 1.2 of the protocol and a set of 24 recommended cipher suites.</td>
</tr>
<tr>
<td>NSA</td>
<td>RFC 6460 defines a TLS v1.2 profile that is fully compliant with Suite B comprised of two cipher suites.</td>
</tr>
</tbody>
</table>

*Notes:*
- IETF: Internet Engineering Task Force
- Mozilla: Securing Mozilla
- BSI: Bundesamt für Sicherheit in der Informationstechnik
- ENISA: European Network and Information Security Agency
- NSA: National Security Agency
- NIST: National Institute of Standards and Technology
- SBA: Spanish Research Ministry
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: [<fffffffff818baf55>] prb_fill_curr_block.isra.62+0 x15/0xc0
[30.605525] PGD 1e0c067 PUD 1e0d067 PMD ffd4067 PTE 8010000007a60065
```

Excerpt of a Kernel crash produced with KERIS
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
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 $k \ll 128$ 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

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t$</th>
<th>Lesperance et al. (2015)</th>
<th>CWV</th>
<th>ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>2</td>
<td>$2^7$</td>
<td>129</td>
<td>11</td>
</tr>
<tr>
<td>128</td>
<td>3</td>
<td>-</td>
<td>256</td>
<td>37</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>$2^{13}$</td>
<td>8,256</td>
<td>112</td>
</tr>
<tr>
<td>128</td>
<td>5</td>
<td>-</td>
<td>16,256</td>
<td>252</td>
</tr>
<tr>
<td>128</td>
<td>6</td>
<td>-</td>
<td>349,504</td>
<td>720</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>-</td>
<td>682,752</td>
<td>2,462</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>$2^{23}$</td>
<td>11,009,376</td>
<td>17,544</td>
</tr>
</tbody>
</table>

**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 \((k)\)

<table>
<thead>
<tr>
<th>(t)</th>
<th>Suite size</th>
<th>Number of activations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(k = 2)</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>37</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>252</td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
<td>307</td>
</tr>
<tr>
<td>7</td>
<td>2462</td>
<td>615</td>
</tr>
<tr>
<td>8</td>
<td>17544</td>
<td>4246</td>
</tr>
</tbody>
</table>

Our Evaluation Results at a Glance

- There are about 366 \textit{trillion} possible combinations for the Trojan activation;
- The whole space is \textit{covered} with less than 18 \textit{thousands} vectors
- .. and these vectors \textit{activate} the Trojan \textit{hundreds} of times
SHA-3 Hash Algorithm Code

• Fully exhaustive testing of 86 implementations
• Found bugs in 41 of these
• 1510 tests based on covering arrays found all errors detected by 1,048,571 exhaustive tests
• Initial project for new direction – designing application domain-specific tests; additional work on post-quantum crypto testing ongoing
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:

\[ \text{ensures balance} \equiv \text{old(balance)} - \text{amount} \]
\[ \&\& \ \text{\result} \equiv \text{balance}; \]

- 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 time of day and day of the week

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 >= MON && d <= 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    A2

time: (0,100,539), (540,1020), (1021,1439)
   B1    B2    B3

priv: boolean

---
aud: boolean

---

day (enum) : A1, A2

time (enum): B1, B2, B3
All of these should be equal

\[ f(0, \begin{bmatrix} 1 \\ 7 \\ 539 \end{bmatrix}, 0, 0) \]

\[ f(0, \begin{bmatrix} 1 \\ 7 \\ 539 \end{bmatrix}, 0, 0) \]

\[ f(0, \begin{bmatrix} 1 \\ 7 \\ 539 \end{bmatrix}, 0, 0) \]

\[ f(0, \begin{bmatrix} 1 \\ 7 \\ 539 \end{bmatrix}, 0, 0) \]
These should also be equal

Now we’re using class A2

\[
\begin{align*}
B1 & \quad \begin{bmatrix} 0 \\ 100 \\ 539 \end{bmatrix}, 0, 0) & A2 & \quad \begin{bmatrix} 0 \\ 100 \\ 539 \end{bmatrix}, 0, 0) \\
2 \quad \begin{bmatrix} 0 \\ 100 \\ 539 \end{bmatrix}, 0, 0) & \quad f(0, 2, 6, 0, 0) & 2 \quad \begin{bmatrix} 0 \\ 100 \\ 539 \end{bmatrix}, 0, 0) & \quad f(0, 2, 6, 0, 0)
\end{align*}
\]
Covering array

Primary array:
0, A2, B1, 1, 1
1, A1, B1, 0, 0
0, A1, B2, 1, 0
1, A2, B2, 0, 1
0, A1, B3, 0, 1
1, A2, B3, 1, 0

emp: boolean
day: (1,7), (2,6)
time: (0,539), (540,1020), (1021, 1439)

A1   A2
B1   B2   B3

priv: boolean
aud: boolean

Class A2 = (2,6)
Class B1 = (0,539)
Run the tests

Correct code output:
3333
0000
0000
1111
0000
2222

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

Faulty code output:
3333
0000
0000
0000
2222
3311
What’s happening here?

We simply detect inconsistency between partitions.
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:

<table>
<thead>
<tr>
<th>Primary x secondary</th>
<th>#tests</th>
<th>total</th>
<th>faults detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-way x 3-way</td>
<td>285x8</td>
<td>2280</td>
<td>6</td>
</tr>
<tr>
<td>4-way x 3-way</td>
<td>970x8</td>
<td>7760</td>
<td>22</td>
</tr>
</tbody>
</table>

- 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?
Test designer evolves to:

All these aspects correspond to the explicit requirements a password must satisfy.
Finished tree -> test parameters

These aspects correspond to the explicit requirements a password must satisfy.

These aspects arise from the experience of the tester or from the use of test catalogues.
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
Learning and Applying Combinatorial Testing

Web sites:

• csrg.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
-- 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 $\phi$ - All: $\phi$ holds on all paths starting from the current state.

E $\phi$ - Exists: $\phi$ holds on some paths starting from the current state.

G $\phi$ - Globally: $\phi$ has to hold on the entire subsequent path.

F $\phi$ - Finally: $\phi$ eventually has to hold

X $\phi$ - Next: $\phi$ 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 \( \text{AG}(P \rightarrow AX(R)) \)
• “for all paths, in every state, if \( P \) then in the next state, \( R \) holds”

• For k-way variable combinations, \( v_1 \& v_2 \& \ldots \& v_k \)

• \( v_i \) abbreviates “\( \text{var}1 = \text{val}1 \)”

• Now combine this constraint with assertion to produce counterexamples. Some possibilities:

1. \( \text{AG}(v_1 \& v_2 \& \ldots \& v_k \& P \rightarrow AX!(R)) \)

2. \( \text{AG}(v_1 \& v_2 \& \ldots \& v_k \rightarrow AX!(1)) \)

3. \( \text{AG}(v_1 \& v_2 \& \ldots \& v_k \rightarrow AX!(R)) \)
What happens with these assertions?

1. $\text{AG}(v_1 \land v_2 \land \ldots \land v_k \land P \rightarrow AX \neg(R))$
   P may have a negation of one of the $v_i$, so we get $0 \rightarrow AX \neg(R)$
   always true, so no counterexample, no test.
   This is too restrictive!

2. $\text{AG}(v_1 \land v_2 \land \ldots \land v_k \rightarrow AX \neg(1))$
   The model checker makes non-deterministic choices for variables not in $v_1..v_k$, so all R values may not be covered by a counterexample.
   This is too loose!

3. $\text{AG}(v_1 \land v_2 \land \ldots \land v_k \rightarrow AX \neg(R))$
   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) → grant
else → 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
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:

\[
\begin{align*}
    R_1 & \rightarrow grant \\
    R_2 & \rightarrow grant \\
    \vdots \\
    R_m & \rightarrow grant \\
    \text{else} & \rightarrow deny
\end{align*}
\]
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:
  \[ G_{TEST_i} = T_i \bigwedge_{j \neq i} \sim T_j \]

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

\(!((\text{emp} \&\& \text{age} > 18 \&\& \text{fa}) \; \text{||} \; \text{(emp} \&\& \text{age} > 18 \&\& \text{emt}) \; \text{||} \; \text{(emp} \&\& \text{age} > 18 \&\& \text{med}))\)

All 3-way combinations of these variables except for positive cases

<table>
<thead>
<tr>
<th>emp</th>
<th>age</th>
<th>fa</th>
<th>emt</th>
<th>med</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
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

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

model-based: [rules] \[\begin{array}{cccccc}
0 & 1 & 0 & 0 & 1 & 1 \end{array}\] \rightarrow \text{model checker} \rightarrow \[\begin{array}{cccccc}
0 & 1 & 0 & 0 & 1 & 1 \end{array}\] \rightarrow \text{grant}

pseudo-exhaustive: [rules] \[\begin{array}{cccccc}
0 & 1 & 0 & 0 & 1 & 1 \end{array}\] \rightarrow \text{test array} \rightarrow \text{grant} \rightarrow \text{covering array} \rightarrow \text{deny}

Use model checker to determine expected result for specified conditions:

Use covering array generator to determine expected result for all $t$-way conditions:
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)

<table>
<thead>
<tr>
<th>JS0</th>
<th>WS1</th>
<th>INT</th>
<th>WS2</th>
<th>EVH</th>
<th>WS3</th>
<th>PAY</th>
<th>WS4</th>
<th>PSE</th>
<th>WS5</th>
<th>JSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&gt;&lt;script&gt;</td>
<td></td>
<td></td>
<td></td>
<td>onError=</td>
<td></td>
<td>alert(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&gt;&lt;script&gt;</td>
<td></td>
<td></td>
<td></td>
<td>onError=</td>
<td></td>
<td>alert(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&gt;&lt;script&gt;</td>
<td></td>
<td></td>
<td></td>
<td>onError=</td>
<td></td>
<td>src=&quot;invalid&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&gt;&lt;script&gt;</td>
<td></td>
<td></td>
<td></td>
<td>onError=</td>
<td></td>
<td>src=&quot;invalid&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

![Diagram](image)

### 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 | null, $kernel_address, $page_zeros, $page_0xff, $page_allocs,...
ARG_FD | fd₁, fd₂, fd₃, ..., fd₁₅
ARG_PATHNAME | pathname₁, pathname₂, pathname₃, ..., pathname₁₅
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

**M7:**
- master_secret: empty, half, default, changebyte, multiply
- finished_label: client finished
- Hash: empty, half, default, changebyte, multiply
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 settings

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

<table>
<thead>
<tr>
<th>Test</th>
<th>OS</th>
<th>Browser</th>
<th>Protocol</th>
<th>CPU</th>
<th>DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XP</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>XP</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>Sybase</td>
</tr>
<tr>
<td>3</td>
<td>XP</td>
<td>IE</td>
<td>IPv6</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>4</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>5</td>
<td>OS X</td>
<td>IE</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>6</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>Oracle</td>
</tr>
<tr>
<td>7</td>
<td>RHL</td>
<td>IE</td>
<td>IPv6</td>
<td>AMD</td>
<td>MySQL</td>
</tr>
<tr>
<td>8</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>Intel</td>
<td>Sybase</td>
</tr>
<tr>
<td>9</td>
<td>RHL</td>
<td>Firefox</td>
<td>IPv4</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
<tr>
<td>10</td>
<td>OS X</td>
<td>Firefox</td>
<td>IPv6</td>
<td>AMD</td>
<td>Oracle</td>
</tr>
</tbody>
</table>
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)