Combinatorial Security Testing Course

Dimitris E. Simos, SBA Research, Austria

Mathematics for Testing, Reliability and Information Security (MaTRIS) Research Group

Rick Kuhn, NIST, USA

Information Technology Laboratory, Computer Security Division

Yu Lei, University of Texas at Arlington, USA

Department of Computer Science and Engineering

Raghu Kacker, NIST, USA

Information Technology Laboratory, Applied and Computational Mathematics Division

April 10, 2018

Hot Topics in the Science of Security: Symposium and Bootcamp (HoTSoS) 2018, Raleigh, NC, USA

Outline of the Tutorial

Introduction

Outline of the Tutorial

Outline of the Tutorial

Outline of the Tutorial

Outline of the Tutorial

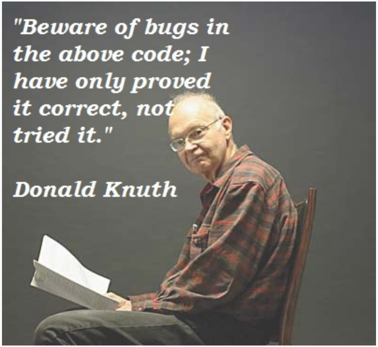
Outline of the Tutorial

Introduction

Introduction

Software (Security) Testing

Should we Care about Software Testing?

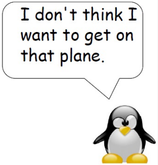


- Proving correctness seems to be not quite enough
- **Testing** is required: both on the sides of verification and validation!
  - “The process of analyzing a software system to detect the differences between existing and expected conditions (that is, bugs)” [IEEE]

Should we Really Care for Software Testing?

Finding 90% of flaws is pretty good, right?

"Relax, our engineers found 90 percent of the flaws."



Can we devise testing methods that show the presence of all flaws?

(assuming certain conditions are met)

Why Software Security Testing?

The Heartbleed Bug (2014)

- Allowed anyone on the Internet to read the memory of the systems protected by OpenSSL software (e.g. e-banking applications)
- "Catastrophic" is the right word. On the scale of 1 to 10, this is an 11 (Schneier, 2014)

How to search for a yet unknown vulnerability - that can be exploited?

Motivation for Combinatorial Methods

Key Observations

- Great need to ensure an attack-free environment for implementations of software systems
- Software testing may consume up to half of the overall software development cost
  - **Combinatorial explosion:** Exhaustive search of input space
  - Added level of complexity for security testing (modelling vulnerabilities)
- How can we estimate the residual risk that remains after testing and guarantee aspects of test quality (e.g. test coverage, locating faults)?

In this Talk

Formulate problems of software security testing as combinatorial problems and then use efficient algorithms/solvers/tools to tackle them

Introduction

Combinatorial Methods

A Large Example for Testing

- Suppose we have a system with on-off switches
- 34 switches = $2^{34} = 1.7 \times 10^{10}$ possible settings

- How do we **test** this system?

Example of a Covering Array for Software Testing

System Under Test (SUT) with 3 Boolean Input Parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)

Table 1: 2-way test set (left) covering all pairs of parameters (right)

Example of a Covering Array for Software Testing

System Under Test (SUT) with 3 Boolean Input Parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)

Table 1: 2-way test set (left) covering all pairs of parameters (right)

Example of a Covering Array for Software Testing

System Under Test (SUT) with 3 Boolean Input Parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)
0	1	1	(0, 1)	(1, 1)	(0, 1)

Table 1: 2-way test set (left) covering all pairs of parameters (right)

Example of a Covering Array for Software Testing

System Under Test (SUT) with 3 Boolean Input Parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)
0	1	1	(0, 1)	(1, 1)	(0, 1)
1	0	1	(1, 0)	(0, 1)	(1, 1)

Table 1: 2-way test set (left) covering all pairs of parameters (right)

Example of a Covering Array for Software Testing

System Under Test (SUT) with 3 Boolean Input Parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)
0	1	1	(0, 1)	(1, 1)	(0, 1)
1	0	1	(1, 0)	(0, 1)	(1, 1)
1	1	0	(1, 1)	(1, 0)	(1, 0)

Table 1: 2-way test set (left) covering all pairs of parameters (right)

Example of a Covering Array for Software Testing

System Under Test (SUT) with 3 Boolean Input Parameters a, b, c

- Could be function, application, configuration file, etc.
- Exhaustive test set: $2^3 = 8$ tests
- 2-way covering array (test set): 4 tests

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)
0	1	1	(0, 1)	(1, 1)	(0, 1)
1	0	1	(1, 0)	(0, 1)	(1, 1)
1	1	0	(1, 1)	(1, 0)	(1, 0)

Table 1: 2-way test set (left) covering all pairs of parameters (right)

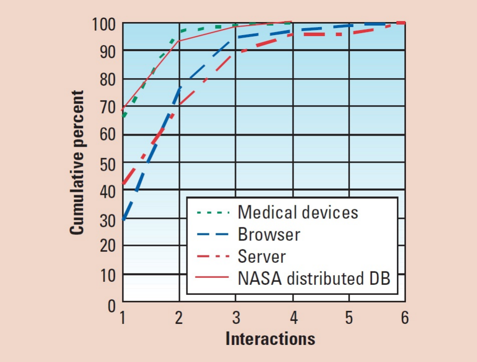
Covering Arrays $CA(N; t, k, v)$ of Strength t

- Cover all t-way combinations of k input parameters at least once
- Input parameters have v total values each
- Such a mathematical object has N total rows (tests)

How is this Knowledge Useful?

- Recall the system with on-off switches
- 34 switches = $2^{34} = 1.7 \times 10^{10}$ possible settings
- **Assumption: What if we knew no failure involves more than 3 switch settings interacting?**
    - If only 3-way combinations, need a CA with only 33 tests
    - If only 4-way combinations, need a CA with only 85 tests

Empirical Evidence: Fault Coverage vs. Interactions



- The maximum degree of interaction observed so far in actual real-world faults is **relatively** small (six)
    - 2-way interaction: **age** $>$ 100 and **zip-code** = 5001, DB push fails
- Most failures are induced by single factor faults or by the **joint combinatorial effect** (interaction) of two factors, with progressively fewer failures induced by interactions between three or more factors

Combinatorial Testing (CT)

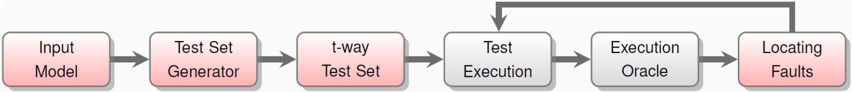
What is Combinatorial Testing?
Combinatorial Strategy for Higher Interaction Testing ($t \geq 2$)

Where it can be Applied?
To system configurations, input data or both

Key Facts:

- CT utilizes 100% coverage of t-way combinations of k input data or system configuration parameters
- Coverage is provided by **mathematical objects** (covering arrays), that are later transformed to software artifacts
- t-way tests that cover **all** such few parameter (factor) interactions can be very effective and provide strong assurance

Research Challenges for Combinatorial Testing



Simplified testing process (CT-dependent parts in red) for given **SUT**

1. Modelling of the test space (configuration space and/or input space) including **specification** of test factors & settings and constraints
2. Efficient generation of t-way test suites, including constraints
3. Determination of the **expected** behavior of the SUT for each test and checking whether the **actual** behavior agrees with the expected one
4. Identification of the **failure-inducing** test value combinations from pass/fail results of CT

Research Challenges for Security Testing

Traditional Software Testing
Generate possible inputs, check if SUT fails

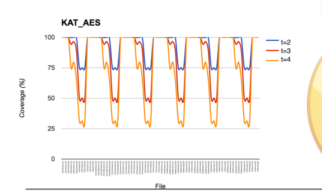
Security Testing: Scope
Generate **malicious** inputs, check if SUT deviated from security regulations (e.g. a payload is executed)

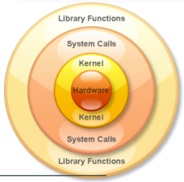
Security Testing: Research Challenges
Security testing always faces the challenge of finding an interaction with the system **not previously tested** that reveals a *new* vulnerability

Combinatorial Security Testing (CST)[1]

- **Large-scale software testing for security**
  - Complex web applications
  - Linux kernels
  - Protocol testing & crypto alg. validation
  - Hardware Trojan horse detection
- **Automated** testing frameworks / Joint Programme with US NIST

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





[1] Simos et al., **Combinatorial Methods in Security Testing**, IEEE Computer, 2016

Web Security Interaction Testing

Web Security Interaction Testing

Attack Models for Web Applications

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

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

Difference from Classical CT: Modelling Attack Vectors

- Attacker injects client-side script in parameter msg:
  `http://www.foo.com/error.php?msg=`<script>alert(1)</script>
- **Input parameter modelling for XSS** attack vectors:
  AV $:= \left(\textit{parameter}_1, \textit{parameter}_2, \ldots, \textit{parameter}_k\right)$

Combinatorial Design of XSS Attack Vectors[2]

Design of an Input Model for XSS

- Input parameters in the model ⇒ parts of the URL
- Parameter value selection: Equivalence and category partitioning
- Constraints derived from expert knowledge (e.g. `JSO(1) => JSE(1)`)

```
JSO(15)::= <script> | <img | ...
WS1(3)::= tab | space | ...
INT(14)::= "'; | ">> | ...
WS2(3)::= tab | space | ...
EVH(3)::= onLoad( | onError( | ...
WS3(3)::= tab | space | ...
PAY(23)::= alert('XSS') | ONLOAD=alert('XSS') | ...
WS4(3)::= tab | space | ...
PAS(11)::= ') | '> | ...
WS5(3)::= tab | space | ...
JSE(9)::= </script> | > | ...
```

[2] Garn, Kapsalis, Simos and Winkler, JAMAICA/ISSTA 2014

Attack Model for XSS with Constraints[3]

Constraints for XSS Input Model

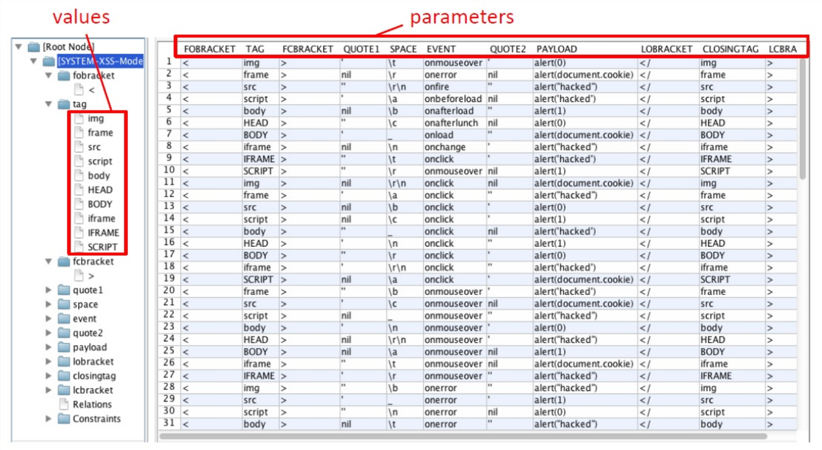
- Enforce combinations that are likely to evade filters
- Exclude combinations that result in inexecutable code
- **Added Value:** Higher quality test sets; further reduction of input space

```
(JSO=1) => (JSE=1)
(JSO=4) => (JSE=2 || JSE=4)
(JSO=5) => (JSE=5 || JSE=6 || JSE=7 || JSE=8)
(EVH=3) => (PAY=12 || PAY=13 || PAY=17)
(INT=2) => (PAS=10 || PAS=11)
(WS1=WS2 && WS2=WS3 && WS3=WS4 && WS4=WS5)
```

Example constraints for the parameters of the XSS attack model

[3] Bozic, Garn, Simos and Wotawa (QRS 2015, IWCT 2015)

A Sample of XSS Attack Vectors



	FOBRACKET	TAG	FCBRACKET	QUOTE1	SPACE	EVENT	QUOTE2	PAYLOAD	LOBRACKET	CLOSINGTAG	LCBRA
1	<	img	>	'	\t	onmouseover	'	alert(0)	</	img	>
2	<	frame	>	nil	\r	onerror	nil	alert(document.cookie)	</	frame	>
3	<	src	>	"	\r\n	onfire	"	alert("hacked")	</	src	>
4	<	script	>	'	\a	onbeforeload	nil	alert('hacked')	</	script	>
5	<	body	>	nil	\b	onafterload	"	alert(1)	</	body	>
6	<	HEAD	>	"	\c	onafterlunch	nil	alert(0)	</	HEAD	>
7	<	BODY	>	'	_	onload	"	alert(document.cookie)	</	BODY	>
8	<	iframe	>	nil	\n	onchange	'	alert("hacked")	</	iframe	>
9	<	IFRAME	>	"	\t	onclick	'	alert('hacked')	</	IFRAME	>
10	<	SCRIPT	>	"	\r	onmouseover	nil	alert(1)	</	SCRIPT	>
11	<	img	>	nil	\r\n	onclick	nil	alert(document.cookie)	</	img	>
12	<	frame	>	'	\a	onclick	"	alert("hacked")	</	frame	>
13	<	src	>	nil	\b	onclick	'	alert(0)	</	src	>
14	<	script	>	nil	\c	onclick	'	alert(1)	</	script	>
15	<	body	>	"	_	onclick	nil	alert('hacked')	</	body	>
16	<	HEAD	>	'	\n	onclick	"	alert(1)	</	HEAD	>
17	<	BODY	>	"	\r	onclick	'	alert(0)	</	BODY	>
18	<	iframe	>	'	\r\n	onclick	"	alert('hacked')	</	iframe	>
19	<	SCRIPT	>	nil	\a	onclick	'	alert(document.cookie)	</	SCRIPT	>
20	<	frame	>	"	\b	onmouseover	'	alert('hacked')	</	frame	>
21	<	src	>	'	\c	onmouseover	nil	alert(document.cookie)	</	src	>
22	<	script	>	nil	_	onmouseover	"	alert("hacked")	</	script	>
23	<	body	>	'	\n	onmouseover	'	alert(0)	</	body	>
24	<	HEAD	>	nil	\r\n	onmouseover	'	alert("hacked")	</	HEAD	>
25	<	BODY	>	nil	\a	onmouseover	nil	alert(1)	</	BODY	>
26	<	iframe	>	"	\t	onmouseover	nil	alert(document.cookie)	</	iframe	>
27	<	IFRAME	>	'	\r	onmouseover	"	alert("hacked")	</	IFRAME	>
28	<	img	>	"	\b	onerror	"	alert("hacked")	</	img	>
29	<	src	>	'	_	onerror	'	alert(1)	</	src	>
30	<	script	>	"	\n	onerror	nil	alert(0)	</	script	>
31	<	body	>	nil	\t	onerror	"	alert("hacked")	</	body	>

Figure 1: XSS vectors in ACTS combinatorial test generation tool (Courtesy of US NIST and Univ. of Texas at Arlington)

How to Design a Security Testing Framework for XSS

Modelling Phase

- Discretize the input space // Designer, tester
- Devise attack model(s) // black-box testing

Test Generation Phase

- Generate CAs from a CT generation tool // automated
- Translate abstract tests to XSS attack vectors // parsers

Test Execution Phase

Extraction **urls**:=CRAWLER(webpage) // parameters to exploit

Injection XSSINJECTOR(**urls**, $\mathrm{attack\ vectors}$) // execution tool

Oracle Check whether an attack vector is executed on webpage

Case Studies and Experimental Evaluation

Case Studies

- OWASP Broken Web Application Project (training applications)
  - **SUTs:** Mutillidae, DVWA, WebGoat, BodgeIt, Gruyere, Bitweaver
  - **Features:** Multiple input fields; several difficulty levels
- Compare CT generated vectors with fuzzers (e.g. OWASP XSS Filter Evasion Cheat Sheet, HTML5 Security Cheat Sheet)
- Compare attack models and test generation algorithms
- Compare penetration testing tools (e.g. BURP Suite, OWASP Xenotix XSS Exploit Framework)

Experimental Results from using CT in Web Security Testing

- Filters can be evaded with low t-way **interaction**
- Largest repository of XSS attack vectors (ahead of commercial and open-source related tools)

Multiple XSS Vulnerabilities in Koha Library

Security Tests for Koha Library

- **SUT:** open source Integrated Library System (used by Museum of Natural History in Vienna, UNESCO, Spanish Ministry of Culture)
- **Results:** unauthenticated SQL Injection, Local File Inclusions, XSS
- **References:** CVE-2015-4633, CVE-2015-4632, CVE-2015-4631

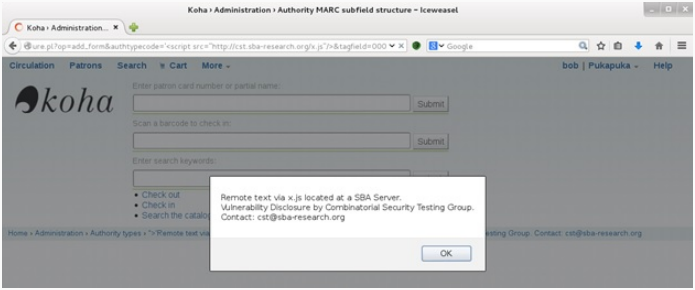


Figure 2: One of the vulnerabilities found by XSSINJECTOR (Prototype tool for automated mounting of XSS attacks)

W3C Vulnerability

Scan of the Whole W3C Website

- **www:** 122 URLs, **Services:** 1 URL, **Validator:** 56 URLs
- **Acknowledgements**: Ted Guild and Rigo Wenning (W3C Team)

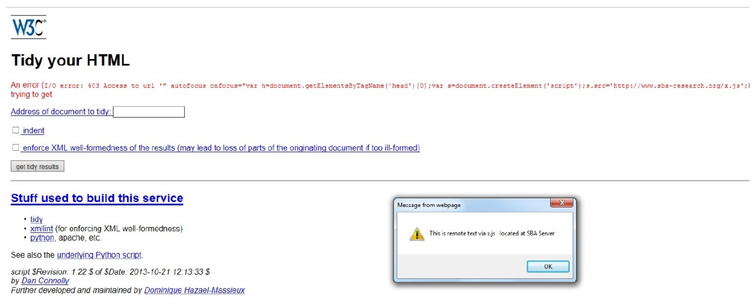


Figure 3: Vulnerability found in tidy service using XSSInjector (Prototype tool for automated mounting of XSS attacks)

Web Security Interaction Testing

Root Cause of Security Vulnerabilities

Analyzing XSS Vulnerabilities using Fault Localization

Goal

- Identify one or more combinations of input parameter values that would definitely trigger an XSS vulnerability
- Different from traditional fault localization, which is aimed at identifying the location of a fault in the source code

XSS Inducing Combinations
If an XSS vector contains an inducing combination, then the execution of this test vector against the SUT will successfully exploit an XSS vulnerability

Why this is Important for Web Security Testing?
Provides important information about why a filter fails to sanitize a malicious vector

Methodology and Results[4]

Methodology

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

JSO	WS1	INT	WS2	EVH	WS3	PAY	WS4	PAS	WS5	JSE
"><script>	␣	’;	␣	onError=	␣	alert(1)	␣	’>	␣	\>
"><script>	␣	’>	␣	onError=	␣	alert(1)	␣	’>	␣	\>
"><script>	␣	’;	␣	onError=	␣	src="invalid"	␣	’>	␣	\>
"><script>	␣	’>	␣	onError=	␣	src="invalid"	␣	’>	␣	\>

Retrieving the Root Cause of Security Vulnerabilities

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

[4] Simos, Kleine, Ghandehari, Garn and Lei, ICTSS 2016

Security Protocol Interaction Testing

Security Protocol Interaction Testing

Combinatorial Methods for X.509 Certificate Testing

The Problem of Certificate Testing

- Standards for public key infrastructure (PKI)
- Attack vectors have the purpose to forge certificates
- **Impact:** Faults in **validation logic** can result in impersonation attacks

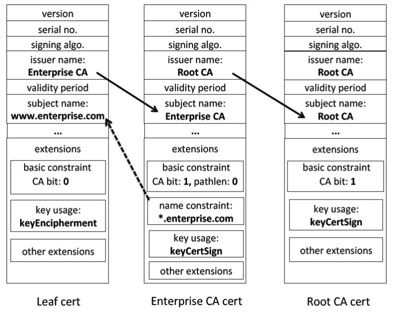


Figure 4: A sample X.509 certificate chain.

Approaches to Certificate Test Generation

Random Selection of Certificate Parts

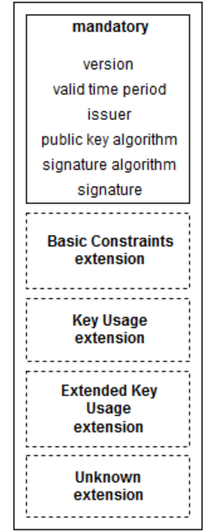
- **Frankencerts:** Random exploration of input space (Brubaker et al., IEEE S&P 2014)
- **Mucerts:** Markov chain Monte Carlo sampling (Chen et al., ESEC/FSE 2015)
- Pros: Revealed a lot of faults
- Cons: No coverage guarantees of input space

Combinatorial Approach for Certificate Test Generation

- **Coveringcerts:** Model the structure (based on RFCs); generated certificates (up to $t = 7$)[5]
- Use differential testing to check for discrepancies
  - Compared validation results of OpenSSL, GnuTLS, wolfSSL, NSS, OpenJDK, BouncyCastle, mbed

[5] Kleine and Simos, ICST 2017

Coveringcerts: Structure



Sample 2-way Test Set for (simplified) Certificates

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

Example: Test Translation

```
Version = 2
Validity_Time = valid
Issuer = Chain
Key_Type = RSA
Signature_Type = Chain
Signature_Algorithm = SHA1
Ext_BC_enabled = 1
Ext_BC_critical = 0
Ext_BC_CA = 1
Ext_BC_pathlen = 1
Ext_KU_enabled = 0
Ext_KU_critical = n/a
Ext_Extended_KU_enabled = 0
Ext_Extended_KU_critical = n/a
Ext_unknown_enabled = 0
Ext_unknown_critical = n/a
```

Listing 1: Abstract test case

```
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:
     [..]
     f8:b8
```

Listing 2: Translated certificate

Example: Test Translation - Validity Period

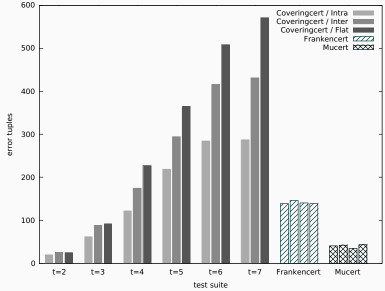
```
Version = 2
Validity_Time = valid
Issuer = Chain
Key_Type = RSA
Signature_Type = Chain
Signature_Algorithm = SHA1
Ext_BC_enabled = 1
Ext_BC_critical = 0
Ext_BC_CA = 1
Ext_BC_pathlen = 1
Ext_KU_enabled = 0
Ext_KU_critical = n/a
Ext_Extended_KU_enabled = 0
Ext_Extended_KU_critical = n/a
Ext_unknown_enabled = 0
Ext_unknown_critical = n/a
```

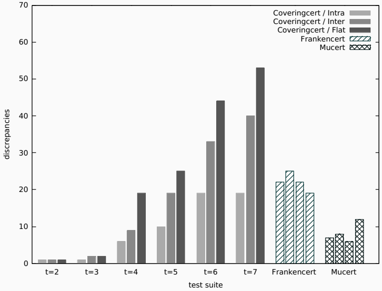
Listing 3: Abstract test case

```
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:
     [..]
     f8:b8
```

Listing 4: Translated certificate

Errors Observed for Different TLS Implementations





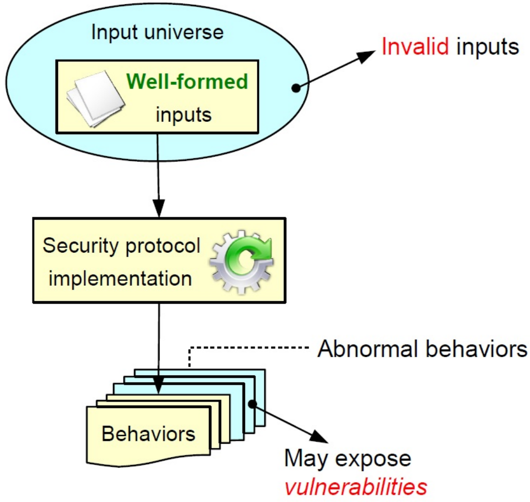
Error	BouncyCastle	wolfSSL	GnuTLS	NSS	OpenJDK	OpenSSL	mbed
untrusted	✓	✓	✓	✓	✓	✓	✓
expired or not yet valid	✓	✓	✓	✓	✓	✓	✓
parse-error	✓	✓	✓	✓	✓	✗	✓
crash	✗	✓	✗	✗	✗	✗	✗
use of insecure algorithm	✗	✗	✓	✓	✗	✗	✓
invalid signature	✗	✓	✓	✓	✗	✗	✗
unknown critical extension	✗	✗	✗	✓	✗	✓	✗
extension in non-v3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

Table 2: A check mark (✓) indicates the error was observed

Security Protocol Interaction Testing

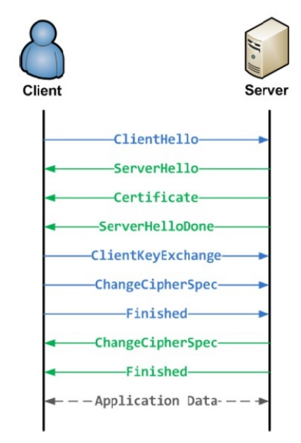
Combinatorial Testing of the TLS Security Protocol

Security Protocol Testing



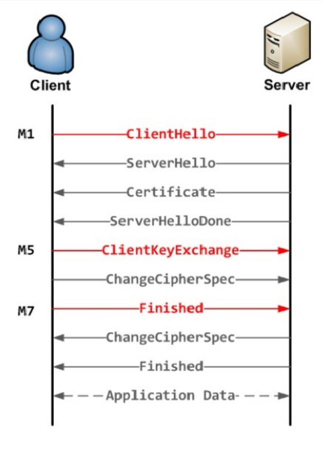
TLS Security Testing

- **TLS Handshake Protocol:** One of the most complex and vulnerable parts of TLS
- Consists of TLS events (messages)
- Every one of these events encompasses a specific set of parameters and values
- Model the interaction and execute it for testing purposes



Combinatorial Modelling of TLS

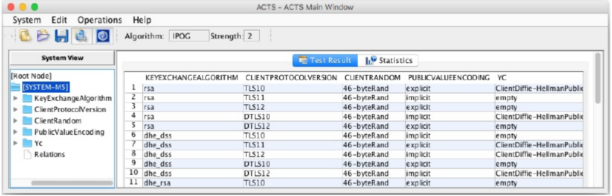
- **Input Test Space for CT:** Employ Input Parameter Modelling (IPM)
- **TLS Specification:** Select parameters and possible values for M1, M5 and M7
- Three different models are constructed which give rise to three distinctive test sets according to standard



Input Models for TLS Messages[6]

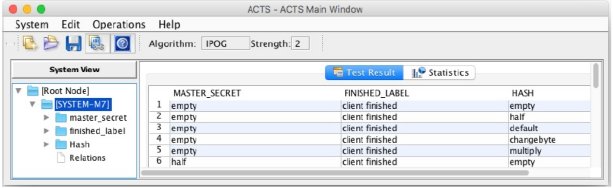
M5:

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



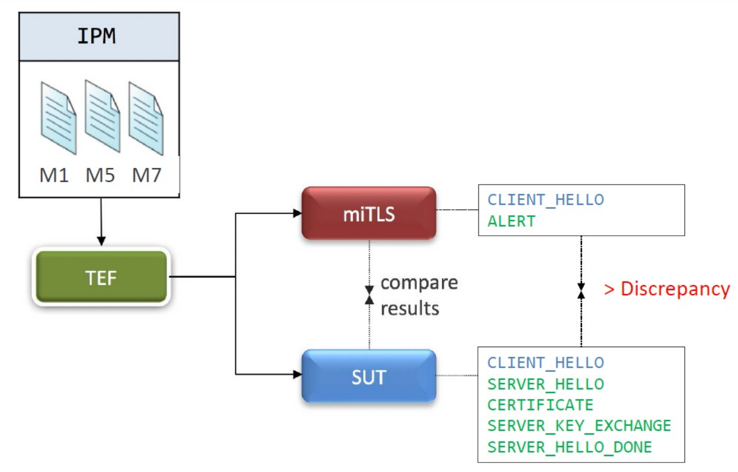
M7:

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



[6] Simos, Bozic, Duan, Garn, Kleine, Lei and Wotawa, ICTSS 2017

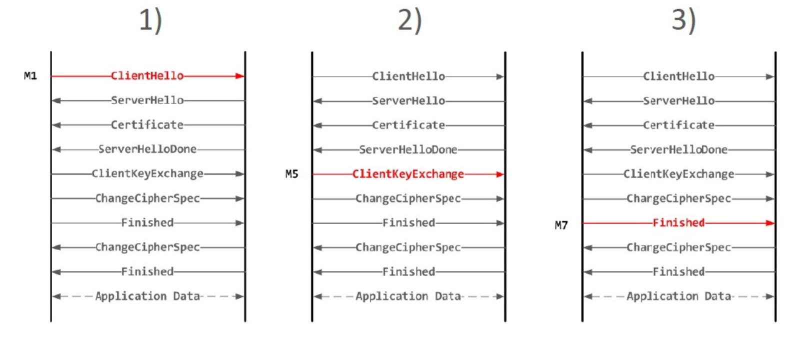
Test Execution Framework (TEF) & Oracle for TLS



Case Study

Three Scenarios for TLS Testing

- Testing each message independently
- **Question:** How does the manipulation of one single TLS event affect the **entire** handshake?



Evaluation Results

Test Case Evaluation

- Compare the resulting execution traces to the submitted input and to the results of other SUTs:
  1. Completed handshake
  2. Rejected by the server
  3. Incomplete handshake

SUT	miTLS			OpenSSL			mbed TLS		
	comp	reject	incomp	comp	reject	incomp	comp	reject	incomp
M1	0	25	0	1	24	0	1	17	7
M5	0	30	0	0	0	30	0	0	30
M7	0	25	0	1	0	24	1	0	24

Conclusion

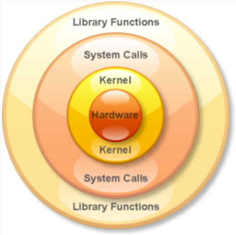
- The developed framework and oracle are **strong** enough to distinguish different behavior among TLS implementations
- Further investigation is needed to track the cause of this behavior and examine whether security leaks have occurred

Combinatorial Methods for Kernel Software

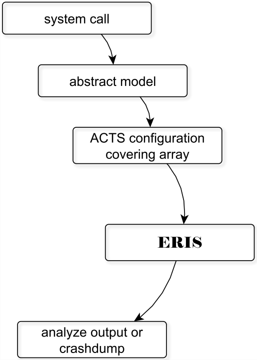
ERIS: Combinatorial Kernel Testing

Kernel Testing

- **Motivation:** Kernel is responsible for managing the hardware and running user programs
- **Challenges:** The kernel of an operating system is the central authority to enforce and control security
    - Large user base (e.g. 1.5 million Android devices activated per day, Google 2013); Critical bugs must be detected early enough!
    - Manual testing approaches (TRINITY fuzzer, Linux test project by IBM, Cisco, Fujitsu, OpenSuse, Red Hat) only
- **Goal:** Reliability and quality assurance of kernel software
- **SUTs:** System calls of every git-commit of any (variant of) Linux



Combinatorial Testing of the Linux System Call Interface[7]

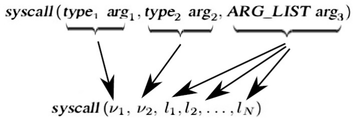


- Abstract models for system calls were manually generated
- ERIS translates them into concrete input models (e.g. ACTS configurations)

[7]Garn and Simos, IWCT 2014

Modelling APIs Function Calls

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



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	fd_1, fd_2, fd_3, ..., fd_{15}
ARG_PATHNAME	$pathname_1$, $pathname_2$, $pathname_3$, ..., $pathname_{15}$

chmod System Call: API Modelling for Input Testing

```
struct syscall syscall_chmod = {
            .name = "chmod",
            .num_args = 2,
            .arg1name = "filename",
            .arg1type = ARG_PATHNAME,
            .arg2name = "mode",
            .arg2type = ARG_MODE_T,
            .rettype = RET_ZERO_SUCCESS,
};
----------------------------------------------
[System]
Name: chmod

[Parameter]
pathname (enum): path1, path2, ... , path15
mode_t    (int): 1, 2, ... , 4096
----------------------------------------------
```

- 2-way CA (full search space): $15 \times 4096 = 61440$ tests

chmod System Call: Flattened API Model

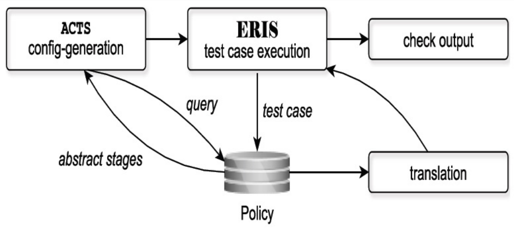
```
--------------------------------------------------
[System]
Name: chmod-flattened

[Parameter]
pathname  (enum) : path1, path2, ... , path15
S_ISUID (boolean): false, true
S_ISGID (boolean): false, true
S_ISVTX (boolean): false, true
S_IRUSR (boolean): false, true
S_IWUSR (boolean): false, true
S_IXUSR (boolean): false, true
S_IRGRP (boolean): false, true
S_IWGRP (boolean): false, true
S_IXGRP (boolean): false, true
S_IROTH (boolean): false, true
S_IWOTH (boolean): false, true
S_IXOTH (boolean): false, true
--------------------------------------------------
```

- 2-way CA (test set): 30 tests

Automated Test Execution Framework

Some Features

- **Ease of use:** Only high-level parameters needed, everything else handled by the system
- **Test generation:** Your favorite CT generation tool
- **Test-runs:** Each invocation runs in a dedicated virtual machine
- **Logging:** Extensive information is captured
- **Database:** Allows sophisticated post-processing queries



ERIS: Combinatorial Kernel Testing

Algorithm 1 Architectural Design of the Core ERIS Framework

```
function ErisCore(version, syscall, t)
Require: version, syscall                  ▷ SUT: Kernel version and system call
Require: t                                 ▷ Interaction strength of CA - test set
    Mount copy of guest image
    Copy latest version of ERIS into guest image
    Generate CA of strength t for syscall  ▷ The CA is translated to a test set
    if precompiled kernel available then
        Use precompiled kernel
    else
        Compile kernel
    end if
    Compile kernel modules
    Install kernel and modules into guest image
    Finalize guest image for testing operations
    Boot guest image using Xen hypervisor
    Execute test set for syscall in dedicated VM
    End testing cycle by shutting down the VM and perform clean-up
    Import test results into SQL database for further analysis
end function
```

Sample Query and Results

SCHEMAS

Filter objects

- eris_tracker
  - Tables
  - Views
  - Stored Procedures
  - Functions
- information_schema
- performance_schema
- test

```sql
SELECT *, kernel_syscall, config_strength, kernel_version
    FROM eris_tracker.run_004 i
    WHERE result_shutdown_clean = 1

AND EXISTS ( SELECT 1 FROM eris_tracker.run_004 o WHERE
(i.kernel_syscall = o.kernel_syscall AND i.config_strength = o.config_strength AND i.kernel_version != o.kernel
AND
(i.result_total != o.result_total OR i.result_success != o.result_success)
)

    ORDER BY kernel_syscall, config_strength, kernel_version;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

lt_shutdown_clean	result_kill	result_segfault	result_total	result_success	result_reject	expected_total	kernel_syscall	config_strength	kernel_version
	0	0	98	28	70	98	sched_getparam	2	v3.18
	0	0	98	28	70	98	sched_getparam	2	v3.19
	0	0	98	28	70	98	sched_getparam	2	v3.19-rc1
	0	0	98	28	70	98	sched_getparam	2	v3.19-rc2
	0	0	98	28	70	98	sched_getparam	2	v3.19-rc3
	0	0	98	28	70	98	sched_getparam	2	v3.19-rc4
	0	0	98	28	70	98	sched_getparam	2	v3.19-rc5
	0	0	97	28	69	98	sched_getparam	2	v3.19-rc6
	0	0	98	28	70	98	sched_getparam	2	v3.19-rc7
	0	0	196	45	151	196	settimeofday	2	v3.18
	0	0	196	45	151	196	settimeofday	2	v3.19
	0	0	196	54	142	196	settimeofday	2	v3.19-rc1
	0	0	196	45	151	196	settimeofday	2	v3.19-rc2
	0	0	196	45	151	196	settimeofday	2	v3.19-rc3
	0	0	196	45	151	196	settimeofday	2	v3.19-rc4

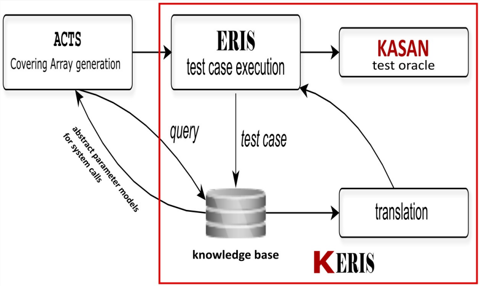
Combinatorial Methods for Kernel Software

KERIS: Combinatorial Kernel Security Testing

KERIS: KASAN Enhanced ERIS

KERIS Overview

- **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 Security Vulnerabilities with KERIS[8]

Security Vulnerability in Linux Networking Stack

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

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

Excerpt of a Kernel crash produced with KERIS

[8] Garn, Wurfl and Simos, HVC 2017

Detecting Hardware Trojan Horses

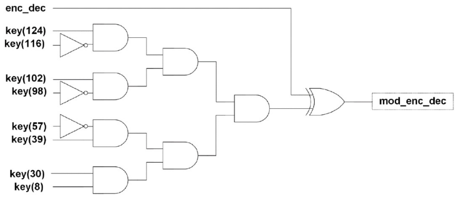
The Problem of Malicious Hardware Logic Detection

Cryptographic Trojans as Instances of Malicious Hardware

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

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)

Trojan Design Nowadays

Allegedly Reported Cases of Hardware Trojans

- **2007:** Syrian radar failed to warn of an incoming air strike (a backdoor built into the system's chips was rumored to be responsible)
- **2012:** Counterfeit semiconductor chips on the rise (commercial, military grade), rumored to be traced back to China

How Large are Today's Hardware Trojan Horses?

Recent study added fewer than 1,000 transistors to the 1.8 million already on the chip (a small backdoor circuit that gave access to privileged regions of chip memory)

- **Increased Awareness:** DARPA Report, 2011, US House of Representatives, 2012, US DoD Trusted Foundry Program 2012

Exciting (Triggering) Hardware Trojan Horses

Threat Model

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

Input Model for Symmetric Ciphers

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

The Problem of Generating a Test Set

The Problem for Testing of Hardware Trojans

- How to efficiently test all possible k-bit input vectors for Trojan activation?

The General (Combinatorial) Test Generation Problem
Let n and $k << n$ parameters of a SUT. Construct sets of test vectors of minimal size that cover **all** possible k-subspaces

- Equivalent to finding a $CA(N; t, k, v)$ with **minimum** number of rows (also called the t-way covering problem)!
- The t-way covering problem is a **hard** combinatorial optimization problem studied for centuries

Overview of Algorithmic Methods for Constructing Test Sets

Main Research Line

- Determining achievable lower bounds
- Either via algorithms or theoretical constructions

Algorithms for Covering Arrays (t-way Test Sets)

- Evolutionary algorithms (SA, TS)
- IPO strategy (extension algorithms)
- One-test-at-a-time methods (greedy, density-based)

Recent Approaches

- Algebraic models for t-way coverage
- Set-based representation of CAs
- Optimization techniques

Optimized Test Sets from CAs

- Comparison of test set sizes using the constant weight vectors (CWV) procedure (Tang and Woo, 1983) and the CA generation methods

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

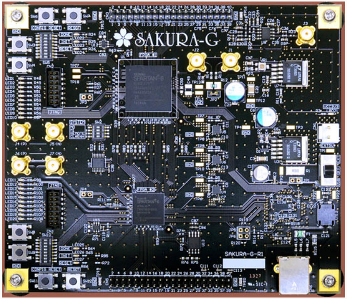
Employed CA Generation Methods:

- Simulated Annealing (SA) algorithms
- CAs from cyclotomy, constructions via Hash families

Case Study for Exciting 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)

Test Results for Detecting Hardware Trojan Horses[9]

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

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

Our Evaluation Results at a Glance

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

[9] Kitsos, Simos, Jimenez and Voyiatzis, ISSRE 2015

Similar (Malicious?) Patterns for AES Software Implementations[10]

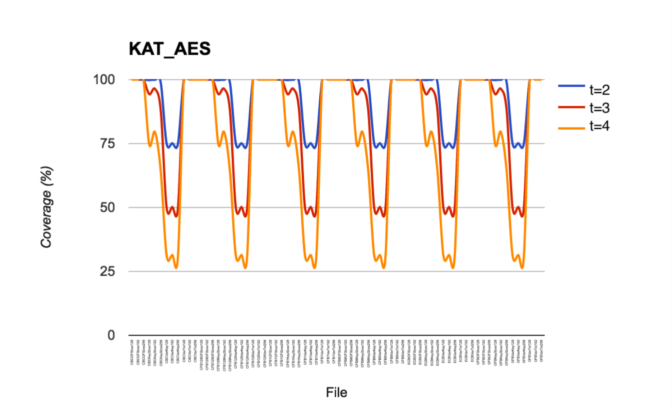
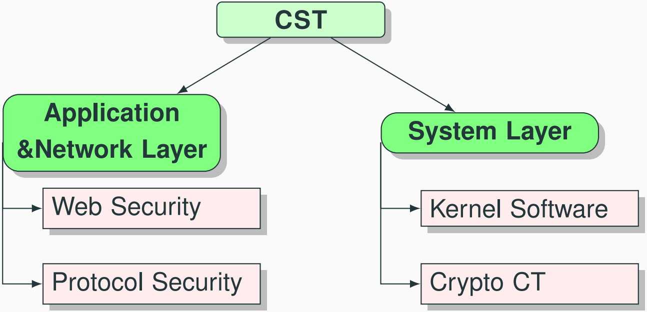


Figure 5: Distinct patterns found via combinatorial coverage measurement analysis on **AES validation test sets** (comprised of Known-Answer-Tests (KAT))

[10] Simos, Mekezis, Kuhn and Kacker, STC 2017

Summary & Future Work

CST: Current State of the Art



Major Goals and Research Challenges Ahead

Accurate **models** and thorough **combinatorial security testing** of **composed software systems** (e.g. Cyber-Physical Systems: Automotives, Avionics Systems, IoT, Critical Infrastructures)

Grand Research Challenge: Cryptographic CT (CCT)

Combinatorial methods for crypto testing (e.g. AES testing, families of cryptographic Trojans)

Grand Research Challenge: Automotive CST

Combinatorial methods for security testing of automotives (e.g. protocol communication)

Questions - Comments

Thank you for your Attention!

dsimos@sba-research.org
cst@sba-research.org