NISTIR 7920

Report on the Metrics and Standards for Software Testing (MaSST) Workshop 2012

Paul E. Black
Elizabeth Fong

http://dx.doi.org/10.6028/NIST.IR.7920

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

NISTIR 7920

Report on the Metrics and Standards for Software Testing (MaSST) Workshop 2012

Paul E. Black
Elizabeth Fong
Software and Systems Division
Information Technology Laboratory

http://dx.doi.org/10.6028/NIST.IR.7920

April 2013

U.S. Department of Commerce
Rebecca Blank, Acting Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Under Secretary of Commerce for Standards and Technology and Director

Abstract

The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project conducted a workshop on Metrics and Standards for Software Testing (MaSST) on June 20, 2012. This workshop was co-located with the IEEE Sixth International Conference on Software Security and Reliability (SERE) 2012 at the National Institute of Standards and Technology, Gaithersburg, Maryland. The main goals of MaSST were to bring together researchers and practitioners to (1) understand the state of the art and state of practice in software testing, (2) define work needed for improved methods and tools for software testing, and (3) list any important problems needing to be solved.

This report contains observations and recommendations based upon the workshop. This report also includes position statements submitted to the workshop and presentation slides. Presentations addressed software testing standards; best practices in testing; testing techniques such as fuzzing, model-based, static and dynamic verification; and vulnerability reporting, etc.

Keywords:

Measurement; metrics; software assurance; software testing; standards.

Disclaimer:

Acknowledgement:

The authors wish to thank Ms. Terri Hunter for formatting this manuscript.

Table of Contents

1. Overview

The goals of this workshop are to bring together a select group of researchers and practitioners to (1) understand the state of the art and state of practice in software testing and (2) define work needed for improved methods and tools for software testing.

Topics of interest include, but are not limited to, the following areas:

- Advanced measurement techniques for properties of software related to testing,
- Theoretically justified means of comparing different coverage metrics,
- The sources of assurance in testing or static analysis,
- Languages to express testable policy or software requirements,
- Standards (in both etalon and norme senses) needed for software testing,
- Gaps and future directions of software analysis using static or dynamic analysis tools,
- Research topics to advance the state of the art in software testing,
- Technology transfer approaches so that current practice benefits from the state of the art, and
- Software testing metrics and measurements.

1.1 Organization

The workshop was co-chaired by Paul E. Black and Elizabeth Fong, National Institute of Standards and Technology. The program committee consisted of the following:

- Paul Ammann, George Mason University
- Taz Daughtrey, DACS
- Mary Ann Davidson, Oracle
- Helen Gill, NSF
- Mark Harman, University of College London
- Cem Kaner, Florida Institute of Technology
- Satoshi Masuda, IBM Japan

- Thomas Ostrand
- Alexander Pretschner, Karlsruhe Institute of Technology
- Gregg Rothermel, University of Nebraska, Lincoln
- Laurie Williams, North Carolina State University

1.2 Agenda and Schedule

The workshop was held at the National Institute of Standards and Technology, Gaithersburg, Maryland, on June 20, 2012. The agenda for this one-day workshop was as follows:

1030	MaSST Keynote: Standards for testing? Stuart Reid, Testing Solutions Group
1100	Paradigm in Verification of Access Control, JeeHyun Hwang, North Carolina State University
1200	Lunch & SERE talk
1400	Why Fuzzing (Still) Works, Allen D. Householder, CERT
1430	Viewpoint-based Testing Architecture Design, Yasuharu Nishi, University of electro-Communications, Japan
1500	Break
1530	Discussion session: Gaps in and Roadmap for Measures and Standards in Software Testing, led by Taz Daughtrey, DACS
1600	Software Testing of Business Applications, Vijay Sampath, Tata Consultancy Services, India

2. Observations and Recommendations

Workshop participants came from government, academic, and industry organizations from several countries.

The general observations below highlight significant trends associated with the activities, standards, methodologies, techniques identified by the workshop participants.

2.1 Software Testing vs Software Assurance

The mainstream definition of software testing is the "process of exercising software to verify that it satisfies specified requirement and to detect errors." [BS7925-1] As such, software testing is one way of performing both software verification and software validation and of achieving confidence in properties of the software.

Static analysis is complementary to testing and involves examining the software instead of executing it. Static analysis includes techniques and approaches ranging from manual design and code reviews to fully automated source code scanners. Static analysis requires access to the software, which may be difficult in proprietary cases or remote or embedded systems. However at least in theory static analysis may consider the effects of *all* possible inputs subject to imprecision from model abstractions and assumptions. For instance, testing could not find a "back door" in code through which the user gains full rights if the user enters a specific string, such as "JoshuaCaleb." However, testing may exercise an entire system end to end. Some tools combine testing and static analysis, gaining the best of both approaches.

The term "software assurance" has been adopted to include both static analysis and testing: any technique to improve the assured security, safety, reliability, quality, etc. of software. The definition of software assurance conveys the thought that development, assessment, and operation processes must provide a reasonable level of justifiable confidence that the software will function correctly and predictably in a manner consistent with its documented requirements [SOAR2007]. Hereafter we use "software assurance" to be more inclusive.

The MaSST workshop adopted the "software assurance" approach in which presentations and discussions were focused on reliability, safety, dependability, and security. In fact, the name of any follow-on workshop should be changed from "Software Testing" to "Software Assurance."

2.2 Software Testing Standards

The word "standard" has two distinct meanings. "Standard" may mean an artifact used as a reference in measurement, for example, the standard kilogram in Paris. The meaning of standard-as-a-thing may be referred to as an "etalon," from a French word for standard. Another meaning is an agreed-upon method or format, such as the C 99 standard, the HTML 5 standard, or ISO 29119. Another French word, "norme," may be used for the standard-as-a-document meaning.

We have two kinds of standards (normes) in software assurance: normes for the software development process and software engineering and normes for software itself. Process normes touch upon steps in the process, roles, techniques, phases, and measurements of productivity and progress. Product normes deal with the properties of software, such as measuring its size, estimating number of remaining bugs, assessing security or safety attributes, validating functionality, and verifying correctness. Product normes may be based on etalons (standard-as-a-thing). For instance, the norme for estimating the number of bugs may be based on using several reference programs with bug thoroughly identified as etalons.

In reviewing the landscape of software testing and assurance standards, the following software testing and assurance standards were discussed:

The MaSST workshop's keynote speaker, Stuart Reid, considered the usefulness of software testing standards such as ISO/IEC 29119. He also briefly covered what should be included in a standard and what is mostly likely to prevent a standard from being adopted. ISO/IEC 29119 comprises four parts:

Part 1: Concepts and vocabulary

Part 2: Test Process

Part 3: Test Documentation

Part 4: Testing Techniques

Part 2 will cover a generic testing process model that can be used within any software development and testing lifecycle. This process will be a layered process covering: organizational test process, test management processes, and fundamental test processes.

Part 3 will cover test documentation across the entire software testing lifecycle. This will include templates across all layers of the 29119 software testing process model, for example: organization test policy process, organizational test strategy process, project test management process, and fundamental test process. IEEE has given ISO permission to use the well-known IEEE 829 test documentation standard as a basis for this part of the standard.

Part 4 will cover software testing techniques across all types of testing, including static (e.g., reviews, inspections, walkthroughs), functional (e.g., black-box, white-box), non-functional (e.g., performance, security, usability) and experience-based (e.g., error guessing, exploratory). The choice of testing techniques is based on the list of application-specific risks. The British Computer Society has given ISO permission to use the BS-7925-1/2 component testing standards as a basis for this part of the standard.

It will replace a number of existing IEEE and BSI standards for software testing:

IEEE 829 Test Documentation

IEEE 1008 Unit Testing

BS 7925-1 Vocabulary of Terms in Software Testing

BS7925-2 Software Component Testing Standard

A future part 5 of the 29119 structure is planned. This part is "Process Assessment" which will be based on ISO/IEC 33063.

Although not discussed at the workshop, other work in software standards is going on. The IEEE software and systems engineering standards committee (S2ESC), chaired by Paul Croll, has many working groups related to Software testing, including:

- No. 730 – Standard for software quality assurance plans, chair: Sue Carroll
- No. 1008 – Standard for software unit testing, chair: Jim Moore
- No. 1012 – Standard for system and software verification and validation, chair: Roger Fujii
- No. 15026 – System and software assurance, chair: Paul Croll

IEEE P1671, Standard for Automatic Test Markup Language (ATML), is for exchanging automatic test equipment and test information via XML. ATML defines a standard exchange medium for sharing information between components of automatic test systems. This information includes test data, resource data, diagnostic data and historic data.

In the software assurance area, there are some standardization activities in the quality and metrics area. The ISO/IEC 9126 standard addresses the quality model, external metrics, internal metrics, and quality in use metrics. The external metrics and internal metrics define six broad, independent categories of quality characteristics as follows: Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability.

The ISO/IEC 15026 system and software engineering committee revised the standard to incorporate the concept of "assurance case." Because the assurance case is considered a life cycle artifact, the revised 15026 also specified how it should be defined, maintained, and revised throughout the system/software life cycle. By 2006, IEEE initiated project P15026 to take over work on the 15026 revision. The latest ISO/IEC draft of the revised standard defines the following life cycle processes, expectations and outcomes: plan assurance activities; establish and maintain the assurance case; and monitor assurance activities and products.

There are many standardization activities in the application-specific areas. There are some standardization activities in the process improvement area. Due to the breadth of the testing discipline covered by the different types of testing standards, this is not discussed here.

2.3 Measures for Software Testing

Can one measure software testing? As a process, yes, but as a product, the answer is more complex. It depends upon which attributes or product characteristics one needs.

In the MaSST workshop, Taz Daughtrey led a discussion session on "gaps in and roadmap for measures and standards in software testing." In attempting to quantify scales of measure for values of software testing, he compared testing to Kirkpatrick's four levels of evaluation of training. This measure can be compared to levels of software testing result.

> Level 1 – Yes/No; The equivalent in testing is did you run the tests you planned?
>
> Level 2 – You pass the quiz or an immediate feedback. The testing equivalent is what are the results?
>
> Level 3 – Your performance improved on the job. The testing equivalent is applying test results to improve the code.
>
> Level 4 – Your work generated return of investment in dollar value. For testing, how much did we improve the code for an amount spent on testing?

Most testers are between levels 1 and 2. Cost-effective education and certification focuses on improving the practice of the bottom 90% of testers. People at the bottom need clear guidance on what to do. Test managers are so busy they do not have time to carefully analyze new techniques or approaches.

There are many papers published in the area of measurement in software engineering and testing as a process. There are few concrete metrics and measurement to precisely and objectively measure the effectiveness or the value of software testing on a software product. In fact, there is no agreement, let alone standards, regarding exactly what can and should be measured as a meaningful indicator that software is "totally" tested or secure (or not vulnerable). Developers need to measure not just return on investment, but properties like correctness and meeting customers' expectations. It would be useful to know how much testing is worth the risk of, say, $1 million? Do we have a model of predictive software quality? For instance if event A (some level of testing) occurs, what is the chance of event B (some level of quality)? If we know the life-cycle cost of software, we may be able to justify levels of testing. With input from Computer Emergency Readiness Team (CERT) and its equivalent in other countries, the software community can ask, what testing would have caught each problem?

2.4 Recommendations

In the area of standards for software testing, there are several working groups; however, the standards for software testing are mainly process oriented. There was agreement that there are big gaps in standards, but no agreement on what the most urgently needed standards are.

We believe that widespread software engineer certification with a continuing education component would *require* software engineers to keep learning.

In the software testing metrics and measurements area, we believe that very little progress was made toward answering the hard question, how much testing is enough? Future research in this area is very much needed. As Reid said, “The only way to do research is to compare effectiveness [of a technique] against a standard.”

We believe that future areas of research, possibly focused more narrowly on a workshop, might be the following:

- Precise definitions of testing techniques [BBT2012];
- Measurement theory which is based on insights from other disciplines such as psychology and physics;
- Laws of software quality and usability;
- Requirements for the next standards needed for software testing;
- Gaps and future directions of software testing using tools;
- Measurement techniques for properties of software;
- The place and future of automated testing [DGG2009].

2.5 References

[BS7925-1] British Standard BS7925-1 Software Testing vocabulary, 1998.

[SOAR2007] Goertzel, K.M., Winograd, T, McKinley, H. L., Oh, L, Colon, M,. McGibbon, T, Fedchak, E, Vienneau, R., “Software Security Assurance: A State-of-the-Art Report (SOAR)”, July 31, 2007.

[BBT2012] Cem Kaner, Black Box Software Testing course site, accessed 27 November 2012.

[DGG2009] Elfriede Dustin, Thom Garrett, and Bernie Gauf , “Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality”, Addison-Wesley, 2009.

Keynote

‘Standards for Testing?’

for Metrics and Standards for Software Testing (MaSST) 2012, NIST, Gaithersburg

This presentation shall consider the topic of software testing standards, briefly covering the following questions:

- Are software testing standards, such as ISO 29119, required by the discipline?
- How do standards authors justify the inclusion of content into standards? – and how should this change?
- What is most likely to stop ISO 29119 being adopted by the testing industry?

The presenter has been the convener of WG26, the working group developing the new ISO/IEC standard, since its inception in 2007 and will provide personal insights into the development of testing standards, having been involved in this area since 1990.

Standards for Testing?

Speaker - Stuart Reid

- 29 Years in Software Eng/IT
- Working on Standards since 1990
- Founder of ISTQB
- Convener - ISO WG26 - Software Testing

What is most likely to stop ISO 29119 being adopted by the testing industry?

- Fear of change
- 'Not invented here'
- Competition
- Lack of required use
- Anti-standardization
- No evidence of efficacy
- Cost of use
- Complexity
- Lack of responsiveness

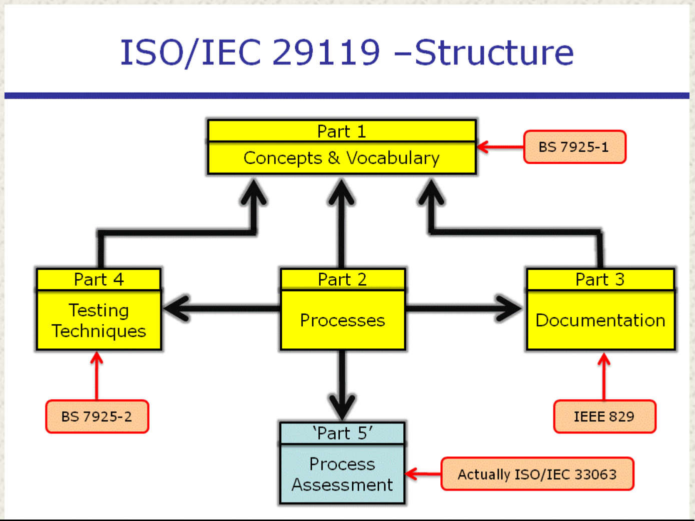
How do standards authors justify the inclusion of content into standards? - and how should this change?

- Input from existing standards
- Input from published texts
- ISO/IEEE Review processes
- Evidence-based inclusion?
- Feedback from use

"There are two things you should never see made ...

...hot dogs and standards"

ISO/IEC 29119


ISO/IEC 29119 –Structure

Are software testing standards, such as ISO 29119, required by the discipline?

- Demand for existing 'standards'
- Gap in the market
- A Baseline for the Testing Discipline

Reprinted from - 2012 IEEE Sixth International Conference on Software Security and Reliability Companion

Paradigm in Verification of Access Control

(*Position Paper*)

JeeHyun Hwang[1], Vincent Hu[2], Tao Xie[1]
[1]Department of Computer Science, North Carolina State University, Raleigh, USA
[2]Computer Security Division, National Institute of Standards and Technology, Gaithersburg, USA
jhwang4@ncsu.edu, vincent.hu@nist.gov, xie@csc.ncsu.edu

Access control (AC) is one of the most fundamental and widely used requirements for privacy and security. Given a subject's access request on a resource in a system, AC determines whether this request is permitted or denied based on AC policies (ACPs). In a system, an ACP is implemented at various places with different purposes. For example, operating systems adopt AC to regulate which users or groups are permitted to read/write/execute files or folders.

The main objective of AC is to protect resources against unauthorized user access. Faults in AC may result in critical consequences such as unauthorized user access on sensitive resources. However, it is a challenging task to implement and maintain AC correctly for two main reasons. First, AC can be complex, especially, when an ACP includes a large number of resources in a sophisticated structure for various groups and users. Second, policy authors may make mistakes when specifying or combining ACPs.

This position paper introduces our approach to ensure the correctness of AC using verification. More specifically, given a model of an ACP, our approach detects inconsistencies between models, specifications, and expected behaviors of AC. Such inconsistencies represent faults (in the ACP), which we target at detecting before ACP deployment. At a high level, ACPs are policy specifications, which encapsulate the expected AC behaviors from policy authors. An ACP model is a representation of ACP behaviors in a formal language.

An ACP consists of a set of rules, which regulate which subject can take a specific action on a specific resource under which condition. In the context of ACPs, input and output are a request (e.g., can user *A* access resource *B*?) and a response (e.g., Permit), respectively. Policy authors may write properties, which can be verified against a given AC model. Properties are different from rules because users create properties based on business practice or user experience. For example, properties can be known security vulnerabilities or a user's security and privacy concerns of interest in AC. We use safety and liveness properties where safety and liveness are characteristics of a given property, denoted by *p*.

Safety property. Safety denotes that *p* is satisfied against an AC model. In other words, there exist no rules in the AC model to violate *p*. Therefore, verification of safety properties is to ensure that "something bad" (i.e., faults) does not happen. For example, a conference program committee member should not review her own submitted paper.

Liveness property. Liveness denotes that an AC model does "something good" (i.e., desired system behaviors). Therefore, verification of a liveness property is to ensure that a "good thing" does happen eventually. One example is deadlock free. Deadlock denotes that a system does not make progress forever since a system waits for an action forever due to more than two competing actions, each of which waits for the other to finish.

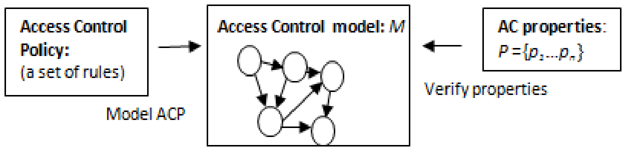

Access Control Policy: (a set of rules)
Model ACP
Access Control model: M
AC properties: P={p1...pn}
Verify properties

Figure 1. Overview of our approach

Figure 1 illustrates our approach. More specifically, we translate an ACP to its corresponding AC model, which is represented as a finite state machine. In this paper, our approach is applied to mandatory access control (MAC) policies, which regulate user and process access to resources.

Our verification uses black-box and white-box checking techniques. For black-box checking, policy authors specify either properties *P*. Given an AC model *q*, if there is no violation, we ensure that *q* is correct according to *P*. Otherwise, *q* is not correct and should be fixed to satisfy property *p'* ∊ *P* that causes violations. In such cases, we use white-box checking to modify *q* to satisfy *P*. For example, we create another *p''* (called a confined property [1]) modified from *p'* where *p''* is a subset of *p* that is responsible for violations. *p''* can be converted to a rule. We add this rule in *q* where *p'* is satisfied after this addition based on the confined property.

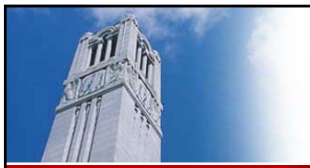
We use NuSMV (http://nusmv.irst.itc.it/), a symbolic model checker to model an ACP. NuSMV supports both BDD-based and SAT-based model-checking approaches, and various analyses including Linear Temporal Logic (TTL) and Computation Tree Logic (CTL) model checking for safety and liveness properties, and counterexample generation. Manually writing properties is tedious and error-prone. To address this issue, our approach generates test requests that can be used as properties for testing AC implementations. An AC implementation evaluates test requests and produces responses, which testers need to inspect to determine whether the responses are correct. We have implemented a prototype [1, 2] for the approach.

Acknowledgment. This work is supported in part by a NIST grant.

References

[1] V. Hu, R. Kuhn, T. Xie, and J. Hwang. Model checking for verification of mandatory access control models and properties, in IJSEKE, Volume 21, Issue 1, Pages 103-127, 2011.

[2] J. Hwang, T. Xie, V. Hu, and M. Altunay, ACPT: A tool for modeling and verifying access control policies. In Proc. POLICY, Demo, Pages 40-43, 2010.



NC STATE UNIVERSITY Computer Science

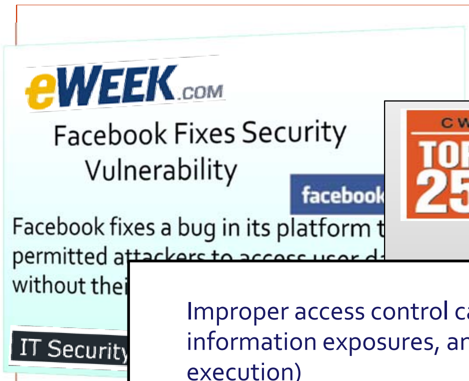
Paradigm in Verification of Access Control

JeeHyun Hwang[1], Tao Xie[1], and Vincent Hu[2]

North Carolina State University[1]
National Institute of Standards and Technology[2]
(MaSST 2012 position paper)

1

Access Control Vulnerabilities



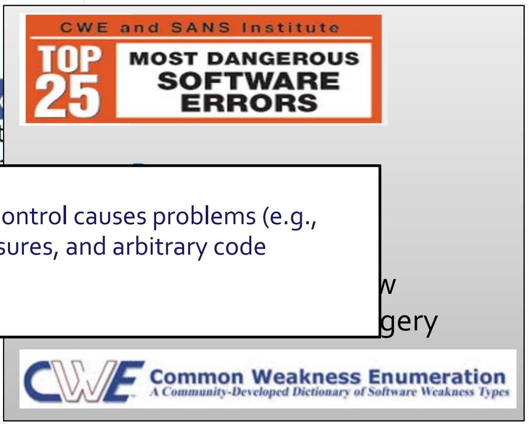
eWEEK.com

Facebook Fixes Security Vulnerability

facebook

Facebook fixes a bug in its platform t
permitted attackers to access user d
without thei

IT Security



CWE and SANS Institute TOP 25 MOST DANGEROUS SOFTWARE ERRORS

w
gery

CWE Common Weakness Enumeration
A Community-Developed Dictionary of Software Weakness Types

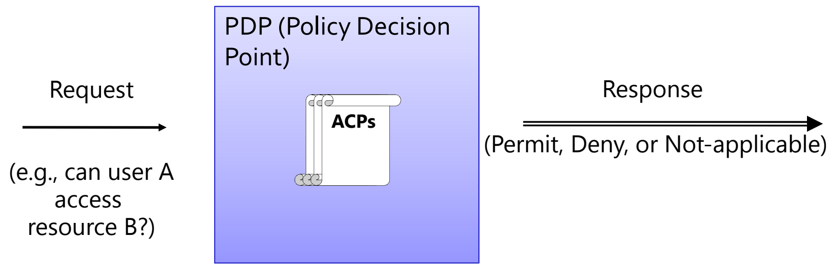
Improper access control causes problems (e.g., information exposures, and arbitrary code execution)

Access Control

- Access control is one of the most widely used privacy and security mechanisms
  - protect critical IT infrastructures such as healthcare, military, intelligence systems
  - prevent security vulnerabilities by controlling access to resources
- Access control is often governed by security policies called Access Control Policies (ACP)
  - include rules that specify which principals such as users or processes have access to which resources

Access Control Mechanism

- Access control mechanisms control which subjects (such as users or processes) have access to which resources
  - A request is evaluated by the Policy Decision Point (PDP) security function based on ACPs


Request
(e.g., can user A access resource B?)
PDP (Policy Decision Point)
ACPs
Response
(Permit, Deny, or Not-applicable)

XACML ACP example

OASIS standard

XML-based language to specif
ACPs

1. The Federal employee is permitted to access the confidential document
2. The State employee is denied to access the confidential document

```
<Rule Effect=“permit“ RuleId="rule-1">
  <Target>
   <Subjects>
    <Subject>federal employee</Subject>
   </Subjects>
   <Actions>
    <Action>access </Action>
   </Actions>
   <Resources>
    <Resource>confidential document </Resource>
   </Resources>
  </Target>
 </Rule>
<Rule Effect=“deny“ RuleId="rule-2">
  <Target>
   <Subjects>
    <Subject>state employee</Subject>
   </Subjects>
   <Actions>
    <Action>access </Action>
   </Actions>
   <Resources>
    <Resource>confidential document </Resource>
   </Resources>
</Target>
 </Rule>
```

Advancing open standards for the information society

Motivation

- Ensure correctness of ACPs
  - ACPs specification may not encapsulate security requirements
    - manual verification of ACPs against user-defined properties is tedious and error-prone
  - ACPs are becoming more complex and manage a large amount of information
    - manual verification of request/response is time-consuming and incomplete

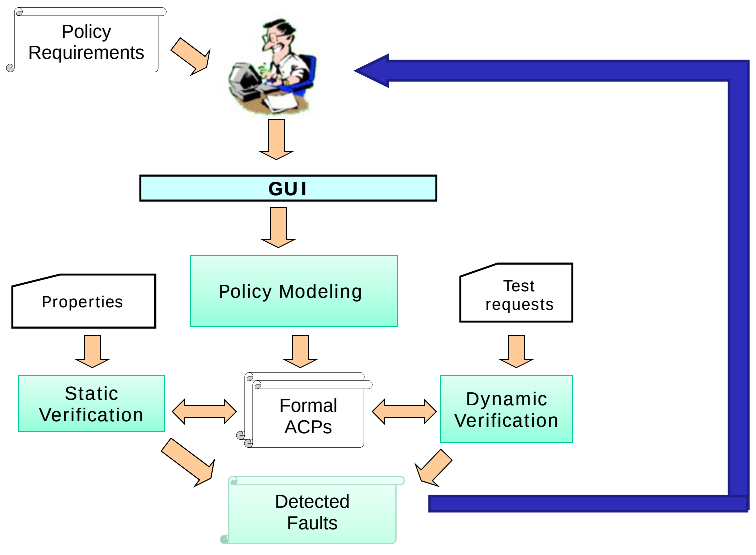
Motivation – con't

- Our approach includes:
  - Static verification: check whether properties are satisfied by a policy
    - Confidence on policy correctness is dependent on the quality of specified properties
    - Check for semantic correctness of the ACP models
  - Dynamic verification: evaluate requests and check whether their evaluated decisions are correct
    - Consider test effort and their effectiveness together
    - Complement static verification
    - Check for syntactic correctness of the ACP implementations

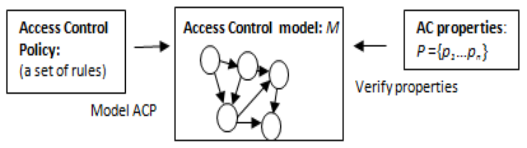
Agenda

- Our approach
  - ACP verification
    - Static verification
      - Safety properties
      - Liveness properties
      - Verification using NuSMV
    - Dynamic verification
      - Combinatorial Array Method by ACTS
- Prototype
- Comparison
- Conclusion

Overview of Our Proposed Approach



Static Verification


Access Control Policy: (a set of rules)
Model ACP
Access Control model: M
AC properties: $P=\{p_1 \ldots p_n\}$
Verify properties

- Static verification requires ACPs and properties for checking
  - Verify a formal ACP against given properties
  - Verification result is "pass" (i.e., satisfied) or "fail" (I.e., unsatisfied)
- Properties
  - Properties are written based on business practice or user experience
    - e.g.) known security vulnerabilities, security and privacy concerns of interest in ACP

Safety Properties

- Safety denotes that a property *p* is satisfied against a formal ACP
  - To ensure the safety, a formal ACPs do not include rules that violate *p*
  - verification of safety properties is to ensure that "something bad" (i.e., faults) does not happen.
    - e.g.) a conference program committee member should not review her own submitted paper
  - *p* is checked on finite executions of a formal ACP

Liveness Properties

- Liveness denotes that a formal ACP does "something good" (i.e., desired ACP behaviors)
  - verification of a liveness property is to ensure that a "good thing" does happen eventually
  - e.g.) whenever any process requests to access its critical section, the request will eventually be permitted
  - Liveness properties are checked on infinite executions of a formal ACP

Our Static Verification Approach

- adopt NuSMV Model Checker, which is Symbolic model checker based on binary decision diagram (BDD) and Boolean satisfiability (SAT)
  - An ACP is represented as Finite State Machine
  - Properties are expressed in temporal logic formula such as LTL (Linear Temporal Logic) or CTL (Computational Temporal Logic)
  - Can control search space by bound search strategy

NuSMV Representation

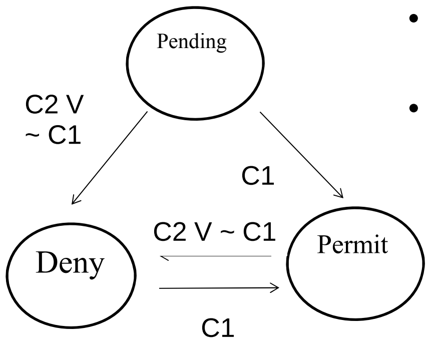
```
var decision: {Pending, Permit, Deny}
...
init(decision) := Pending;
next (decision) :=
     case
     subject = federal employ
     & Resource = confidentia
      & Action = access:  Permit;
     subject = state employee
     & Resource = confidential doc
      & Action = access: Deny ;
     ...
     1 : Deny;
esac;
...
```

Declare variables

Initialize variables

Specify ACP behavior

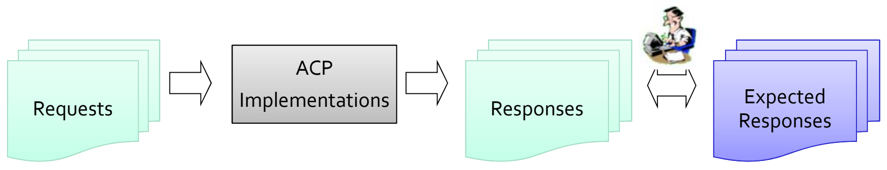
Static Verification: Property Checking



- Search states where a user-specified property is violated
- Properties
  - Safety property example: In case of C1, state always reach "Permit" state
  - Liveness property example: a path exists to reach state "Permit" state

- Conditions
  - C1: Subject = federal employee & Resource = confidential doc & Action = access
  - C2: Subject = state employee & Resource = confidential doc & Action = access

Our Approach: - Dynamic Verification (Testing)



- Automatically generate test inputs (requests) for testing of ACP implementations
  - White box testing
    - based on policy structural coverage (e.g., rule)
  - Black box testing
    - based on combinatorial coverage
      Cover n-wise (e.g., pair-wise) combinations of attribute values

Rule Coverage

Rationale: when the policy rule containing a fault is not checked (i.e., "covered") with a property, the fault is not exposed

- A rule r is covered by a property p when the access decision d of p depends on r of the ACP model
  - "CM(r, p) = True" means that a property p depends on r to reach the access decision

<u>We write properties to cover each rule at lease once</u>

Impact Analysis

In order to determine whether a rule is covered by p, we use impact analysis by mutating chosen rule's decision (e.g., permit -> deny)

- Given r's decision-mutated rule ~r
  - CM(r, p) = True & CM(~r, p) = True: r is not covered by p
  - CM(r, p) = True & CM(~r, p) = False: r is covered by p

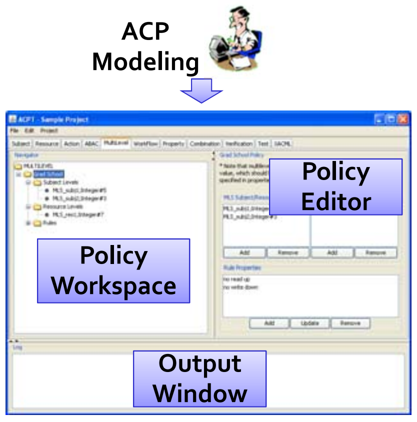
Our prototype

- Our prototype is developed as a part of Access Control Policy Tool (ACPT) research in collaboration with NIST
  - model ACPs via GUI
  - verify ACPs via dynamic / static verification
  - Beta-tested in agencies/labs and companies

NIST National Institute of Standards and Technology U.S. Department of Commerce · DISA Defense Information Systems Agency Department of Defense · Johns Hopkins University · NOAA · inventure enterprises · SAIC From Science to Solutions · Fermilab

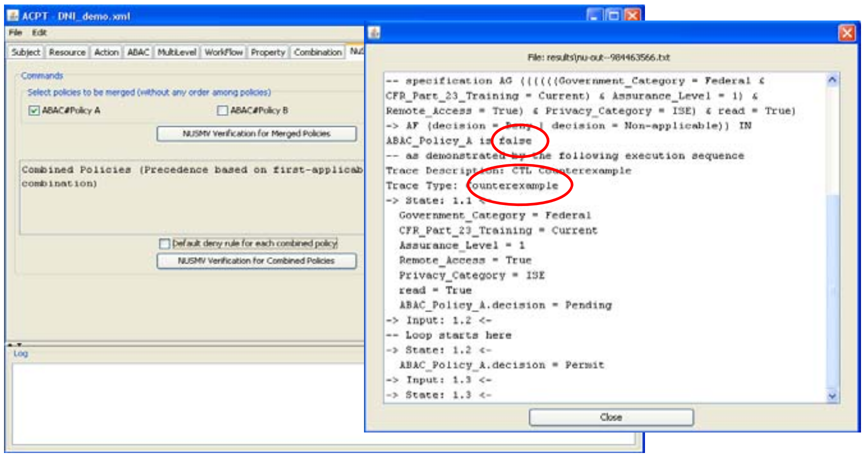
ACP Modeling via GUI

- Assist users to compose ACPs (such as Role Based, Multi-Level, workflow policy models) via GUI
  - Help specifying ACPs and properties through model templates
  - Support various ACP combining algorithms (e.g., first applicable or permit-overrides)



Static Verification

Verify the property against Policy A, the result return false with counterexample.

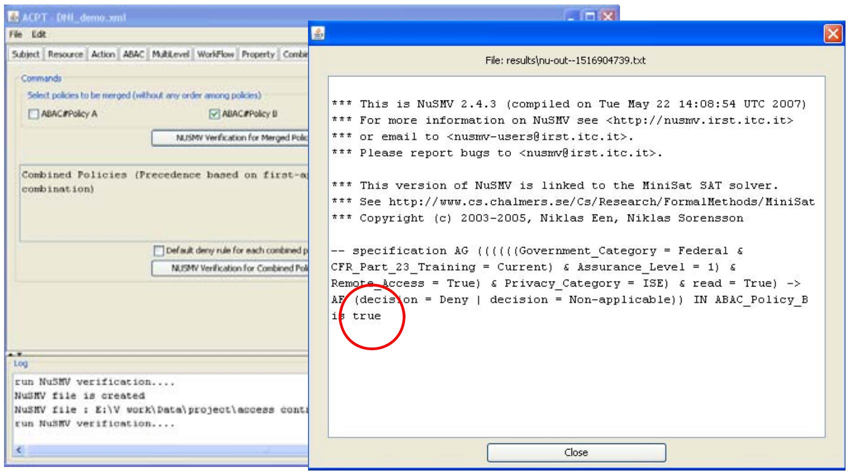

ACPT - DNI_demo.xml
File Edit
Subject Resource Action ABAC MultiLevel WorkFlow Property Combination
Commands
Select policies to be merged (without any order among policies)
ABAC#Policy A
ABAC#Policy B
NUSMV Verification for Merged Policies
Default deny rule for each combined policy
NUSMV Verification for Combined Policies
Log
File: results\nu-out--984463566.txt
Close

```
Combined Policies (Precedence based on first-applicab
combination)
```

```
-- specification AG ((((((Government_Category = Federal &
CFR_Part_23_Training = Current) & Assurance_Level = 1) &
Remote_Access = True) & Privacy_Category = ISE) & read = True)
-> AF (decision = Deny | decision = Non-applicable)) IN
ABAC_Policy_A is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  Government_Category = Federal
  CFR_Part_23_Training = Current
  Assurance_Level = 1
  Remote_Access = True
  Privacy_Category = ISE
  read = True
  ABAC_Policy_A.decision = Pending
-> Input: 1.2 <-
-- Loop starts here
-> State: 1.2 <-
  ABAC_Policy_A.decision = Permit
-> Input: 1.3 <-
-> State: 1.3 <-
```

Static Verification (cont.)

Verify the property against Policy B, the result return true.

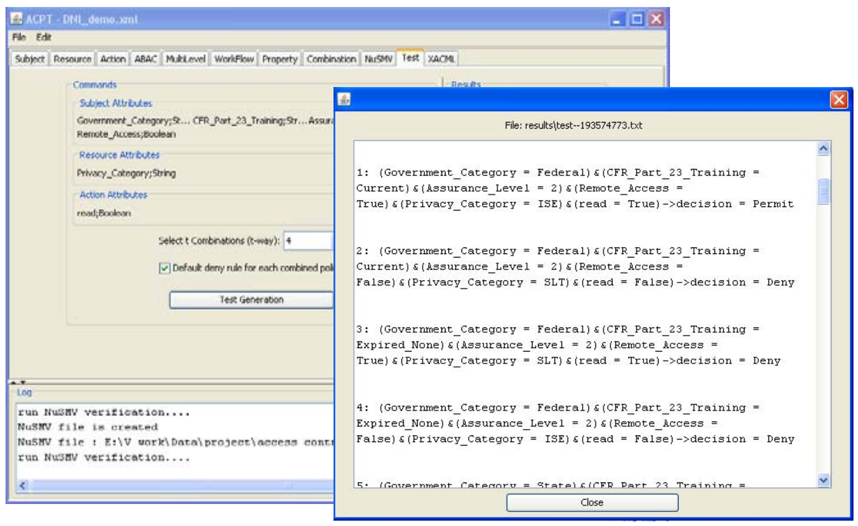

ACPT - DNI_demo.xml
File Edit
Subject Resource Action ABAC MultiLevel WorkFlow Property Combi
Commands
Select policies to be merged (without any order among policies)
ABAC#Policy A
ABAC#Policy B
NUSMV Verification for Merged Polic
Default deny rule for each combined p
NUSMV Verification for Combined Pol
Log
File: results\nu-out--1516904739.txt
Close

```
Combined Policies (Precedence based on first-a
combination)
```

```
run NuSMV verification....
NuSMV file is created
NuSMV file : E:\V work\Data\project\access cont
run NuSMV verification....
```

```
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification AG ((((((Government_Category = Federal &
CFR_Part_23_Training = Current) & Assurance_Level = 1) &
Remote_Access = True) & Privacy_Category = ISE) & read = True) ->
AF (decision = Deny | decision = Non-applicable)) IN ABAC_Policy_B
is true
```

Test Input Generation and Evaluation


ACPT - DNI_demo.xml
File Edit
Subject Resource Action ABAC MultiLevel WorkFlow Property Combination NuSMV Test XACML
Commands
Results
Test Generation
Log
Close

Subject Attributes
Government_Category;St... CFR_Part_23_Training;Str... Assura
Remote_Access;Boolean

Resource Attributes
Privacy_Category;String

Action Attributes
read;Boolean

Select t Combinations (t-way): 4

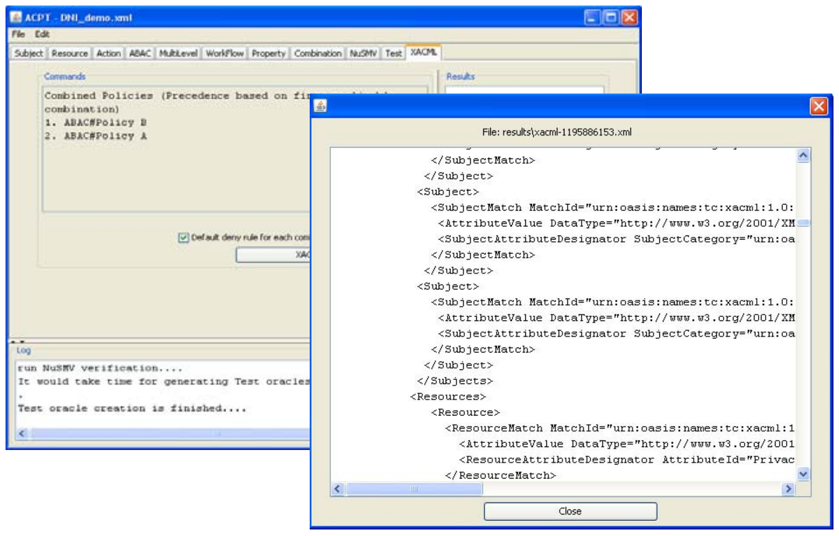
[✓] Default deny rule for each combined pol

```
run NuSMV verification....
NuSMV file is created
NuSMV file : E:\V work\Data\project\access cont
run NuSMV verification....
```

File: results\test--193574773.txt

```
1: (Government_Category = Federal)&(CFR_Part_23_Training =
Current)&(Assurance_Level = 2)&(Remote_Access =
True)&(Privacy_Category = ISE)&(read = True)->decision = Permit


2: (Government_Category = Federal)&(CFR_Part_23_Training =
Current)&(Assurance_Level = 2)&(Remote_Access =
False)&(Privacy_Category = SLT)&(read = False)->decision = Deny


3: (Government_Category = Federal)&(CFR_Part_23_Training =
Expired_None)&(Assurance_Level = 2)&(Remote_Access =
True)&(Privacy_Category = SLT)&(read = True)->decision = Deny


4: (Government_Category = Federal)&(CFR_Part_23_Training =
Expired_None)&(Assurance_Level = 2)&(Remote_Access =
False)&(Privacy_Category = ISE)&(read = False)->decision = Deny


5: (Government_Category = State)&(CFR_Part_23_Training =
```

XACML ACP Generation


ACPT - DNI_demo.xml
File Edit
Subject Resource Action ABAC MultiLevel WorkFlow Property Combination NuSMV Test XACML
Commands
Results
XAC
Log
Close

```
Combined Policies (Precedence based on fi
combination)
1. ABAC#Policy B
2. ABAC#Policy A
```

[✓] Default deny rule for each com

```
run NuSMV verification....
It would take time for generating Test oracles
.
Test oracle creation is finished....
```

File: results\xacml-1195886153.xml

```
        </SubjectMatch>
       </Subject>
      <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
         <AttributeValue DataType="http://www.w3.org/2001/XM
         <SubjectAttributeDesignator SubjectCategory="urn:oa
        </SubjectMatch>
       </Subject>
      <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
         <AttributeValue DataType="http://www.w3.org/2001/XM
         <SubjectAttributeDesignator SubjectCategory="urn:oa
        </SubjectMatch>
       </Subject>
      </Subjects>
     <Resources>
        <Resource>
          <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1
            <AttributeValue DataType="http://www.w3.org/2001
            <ResourceAttributeDesignator AttributeId="Privac
          </ResourceMatch>
```

Comparison

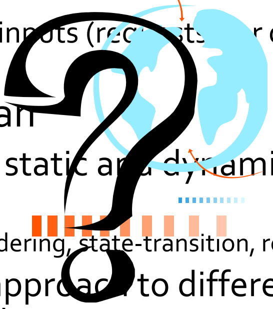
Comparison with commercial or research tools

Product	Modeling via GUI	Static Verification	Dynamic Verification
Our approach	✓	✓	✓
Existing model checking approaches [1,2,3]		Partial ✓	
IBM Security Policy Manager V7.0 [4]	✓	Partial ✓	
Cisco Policy Manager [5]	✓		

1. N. Zhang et al., Evaluating access control policies through model checking. In Proc. 8th ISC, 2005.
2. S. Kikuchi, et al., Policy verification and validation framework based on model checking approach, in Proc. ICAC, 2007.
3. A. Schaad, V. Lotz, and K. Sohr, A model-checking approach to analysing organisational controls in a loan origination process, in Proc SACMAT, 2006.
4. IBM Policy Manager V7.0: http://www.redbooks.ibm.com/redpapers/pdfs/redp4512.pdf
5. Cisco Policy Manager: http://www.cisco.com/en/US/products/ps9530/index.html

Conclusion

- ACP verification to ensure correctness of ACPs
  - Formally verify ACPs against user-specified properties
  - Generate test inputs (requests) for dynamic verification
- Our future plan
  - Improve our static and dynamic verification
    - Condition, ordering, state-transition, role hierarchy, …
  - Extend our approach to different application domains
    - Healthcare, Law statutes, Military, …



ACPT NIST

NIST

Reprinted from - 2012 IEEE Sixth International Conference on Software Security and Reliability Companion

Viewpoint-based Test Architecture Design

Yasuharu NISHI
Department of Informatics, Graduate School of Informatics and Engineering
The University of Electro-Communications, Tokyo
Tokyo, Japan
e-mail: Yasuharu.Nishi@uec.ac.jp

***Abstract*— Software test recently becomes large-scale and complicated artifact as software itself. Research and practices has to be boosted such as test architecture. In this paper first we mention TDLC: Test Development Life Cycle, which includes test requirement design phase and test architecture design phase instead of test planning from engineering view. Second we discuss concepts of test architecture and propose NGT: Notation for Generic Testing, which is a set of concepts or notation for design of software test architecture. Viewpoint is discussed as a key concept of test architecture representing a group of test cases and test objective. And this paper gives an example of test architecture model. Finally this paper shows possibility that viewpoint diagram will be a platform of test architecture design technology such as test design patterns, test architecture style, variability analysis of product line engineering and so on.**

Keywords- test architecture; test development life cycle; test requirement analysis; test suite; viewpoint; UTP; NGT;

I. INTRODUCTION

Software test recently becomes large-scale and complicated artifact as software itself. There can be a test project with over one million test cases or with over ten test levels. Technology of large-scale and complicated software test has just begun advance and has to be boosted.

"Software architecture" technology arose in 1990s for development of large-scale and complicated software based on abstraction, separation of concerns, modeling, patterns and so on. "Software test architecture" technology has just arising in our age, and we have to boost research and practices on software test architecture technology more and more. This paper shows perspective of research and practices on software test architecture.

Architecture of software system has two kinds of scope: system architecture and software architecture. System architecture is for software, platform, peripherals, environment, network et al. Software architecture is only for software inside, which mainly consists of modules (groups of statements) such as classes.

Test architecture also has two kinds of scope: test system architecture and test suite architecture. Test system architecture is for test system, system/software to be tested (SUT), platform where SUT is executed, generator of test cases et al. Test suite architecture is for test suite inside, which mainly consists of groups of test cases such as test levels and test types.

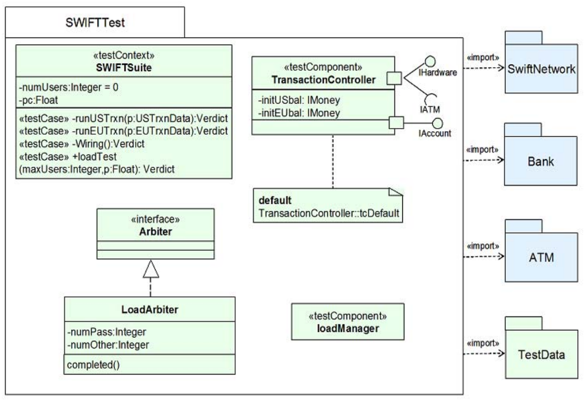


Figure 1. A Test system architecture example on UTP[1]

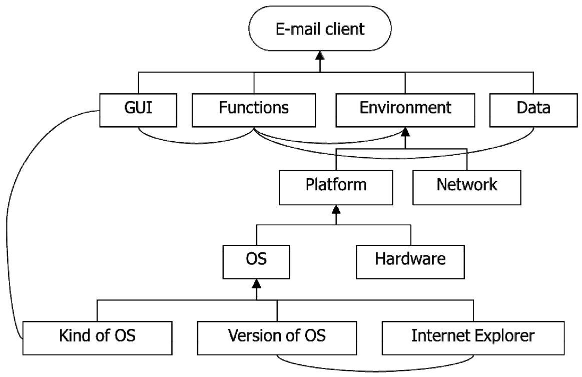


Figure 2. A Test suite architecture example on NGT

II. TEST SYSTEM ARCHITECTURE AND TEST SUITE ARCHITECTURE

There are several research and practices on test system architecture. UML Test Profile[1] is standardized as a notation based on UML for test system architecture. But research and practices of test suite architecture stays just experiences and heuristics. In this paper hereinafter the word "test architecture" means test suite architecture. Fig.1 shows an example of test system architecture according to UTP, UML Test Profile. Fig.2 shows an example of test suite architecture according to NGT, Notation of Generic Testing discussed in chapter V.

978-0-7695-4743-5/12 $26.00 © 2012 IEEE
DOI 10.1109/SERE-C.2012.15

III. TEST PLANNING AND TEST ARCHITECTURE DESIGN

Test process is recognized roughly by tradition as below: Test planning, test design and test execution. Traditional test design means a phase to derive test cases by test techniques such as control path testing. Traditional test planning means a phase which includes planning test project and drawing big picture of test cases, that is, which includes both tasks of management side and engineering side.

In software development project planning phase includes only tasks of management side and software architecture design phase fills a role of drawing big picture of software, that is, just engineering side. A lot of companies have both positions of project manager and software architect. In software testing tasks of management side and engineering side are traditionally mixed as test planning, test strategy or test approach, because software testing is tight and careful task for budget and effort. Fig.3 shows Heuristic Test Strategy Model by James Bach [2]. Mixture and severe constraint lead test researchers and practitioners to one-sided view. A lot of companies have a position of "test manager" but only a few companies have a position of "test architect".

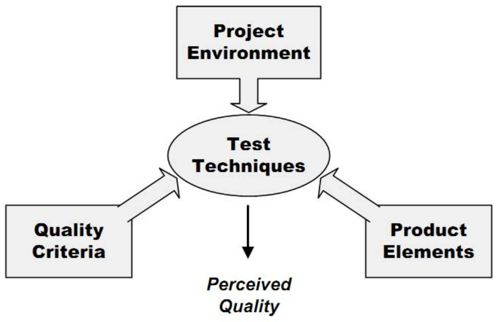


Figure 3. Heuristic Test Strategy Model [2]

To boost research and practices on software test architecture technology, we have to distinguish management side and engineering side. It is necessary to re-define test process only from engineering side named TDLC, Test Development Life Cycle. Fig. 4 shows TDLC, which consists of four phases: test requirement analysis, test architecture design, test detail design and test implementation. TDLC is just to develop test cases or test script. Whole test process needs test execution phase, test result recording phase and several test management tasks.

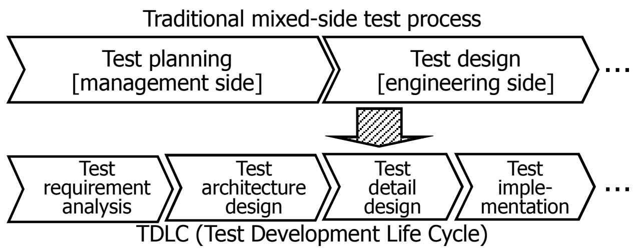

Traditional mixed-side test process
Test planning [management side]
Test design [engineering side]
…
Test requirement analysis
Test architecture design
Test detail design
Test imple-mentation
…
TDLC (Test Development Life Cycle)

Figure 4. TDLC (Test Development Life Cycle)

IV. CONCEPTS FOR TEST ARCHITECTURE

As there is still no agreement on the precise definition of the term "software architecture", the precise definition of test architecture is impossible for the present. For example IEEE std. 1471[3] defines "architecture" as "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution". To follow IEEE's definition, we have to clarify what are components and relationships as well as statements in software testing.

It is natural for statements to correspond to test cases or test scripts. This correspondence leads components to be group of test cases such as test types or test levels, which are essentially hierarchical. It should be noted that classes, which are components in OO paradigm, has two angles. The one is group of statements (and data) as an extension of structured programming as a way of OOP. The other one is constituent of the world as a way of OOA. Test types or test levels may be from the former angle. We should deeply discuss which angle is suitable for test architecture just following test requirement analysis and how seamless test requirement analysis model and test architecture model should be.

Relationships are more difficult than statements and components. There may be at least two types of relationships. The one is for combinatorial testing. If a load test type should be tested combinatorially with a configuration test type, they have some relationship. The other one is sequential dependency. As an integration test level should be tested after a unit test level, they have some relationship. We should find various types of relationships.

In addition some principles for software design can be applicable such as abstraction, separation of concerns, modularity. Quality characteristics of test suite can indicate and assist good test design such as maintainability of test suite or test cases. Notation or formulation can make engineers easy to store reusable test assets, test design patterns and test architecture styles. Product line engineering of test suite can arise separately from test design just for software product line.

V. NGT: NOTATION FOR TEST ARCHITECTURE DESIGN

For design of test architecture, notation or a set of concepts is necessary. It should consist of concepts of a group of test cases, hierarchical structure, relationship for combinatorial testing, relationship for sequential dependency. It would be better if it can harmonize the principles, abstraction, separation of concerns, modularity, quality characteristics.

We propose notation or a set of concepts named NGT, Notation for Generic Testing. NGT consists of three concepts which are viewpoint, hierarchical relationship and interactive relationship. Viewpoint is a concept of a group of test cases. Hierarchical relationship is used for hierarchical structure of viewpoints. Hierarchical relationship means abstraction (is-a), composition (has-a), cause-effect and object-attribute. Interactive relationship means necessity for combinatorial testing.

Lowest boxes (most detailed viewpoints) usually mean coverage items of groups of test cases or test detail design. Test detail design is a phase to extract test cases by test design technique such as equivalence partitioning, control flow testing and state transition testing. When control flow testing is used, “control flow” is most detailed viewpoints.

In Fig.2 viewpoint diagram of NGT, boxes represent viewpoints. Directional lines represent hierarchical relationships and unidirectional curved lines represent interactive relationships. This diagram is named as “Viewpoint diagram”.

Though viewpoint diagram looks similar to classification tree[4], viewpoint diagram is more suitable for drawing big picture of software testing than classification tree. Viewpoint concept doesn’t include only equivalence partition but coverage item. Viewpoint diagram can represent combinatorial relationships in the same diagram as viewpoints at higher abstraction level, i.e. coverage item level, although classification tree can do so in different diagrams at lower abstraction level, i.e. parameter level.

Viewpoint diagram has also two angles. Like an angle of OOP, which is lower abstraction level, viewpoint means a group of test cases. Like an angle of OOA, which is higher abstraction level, viewpoint means test objective. In test requirement analysis phase test objectives are listed and refined. Fig.5 shows an example of viewpoint diagram for testing of some mission critical system in test requirement analysis phase. Viewpoints are listed enough but combinatorial relationships are too many and too complicated to test. In test architecture design phase the viewpoints diagram should be well-organized using modeling technique.

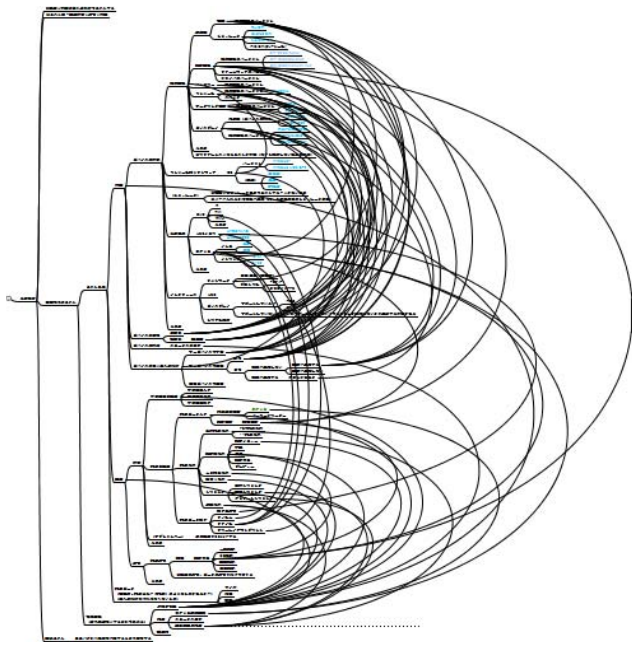


Figure 5. An example of viewpoint diagram in test requirement analysis phase

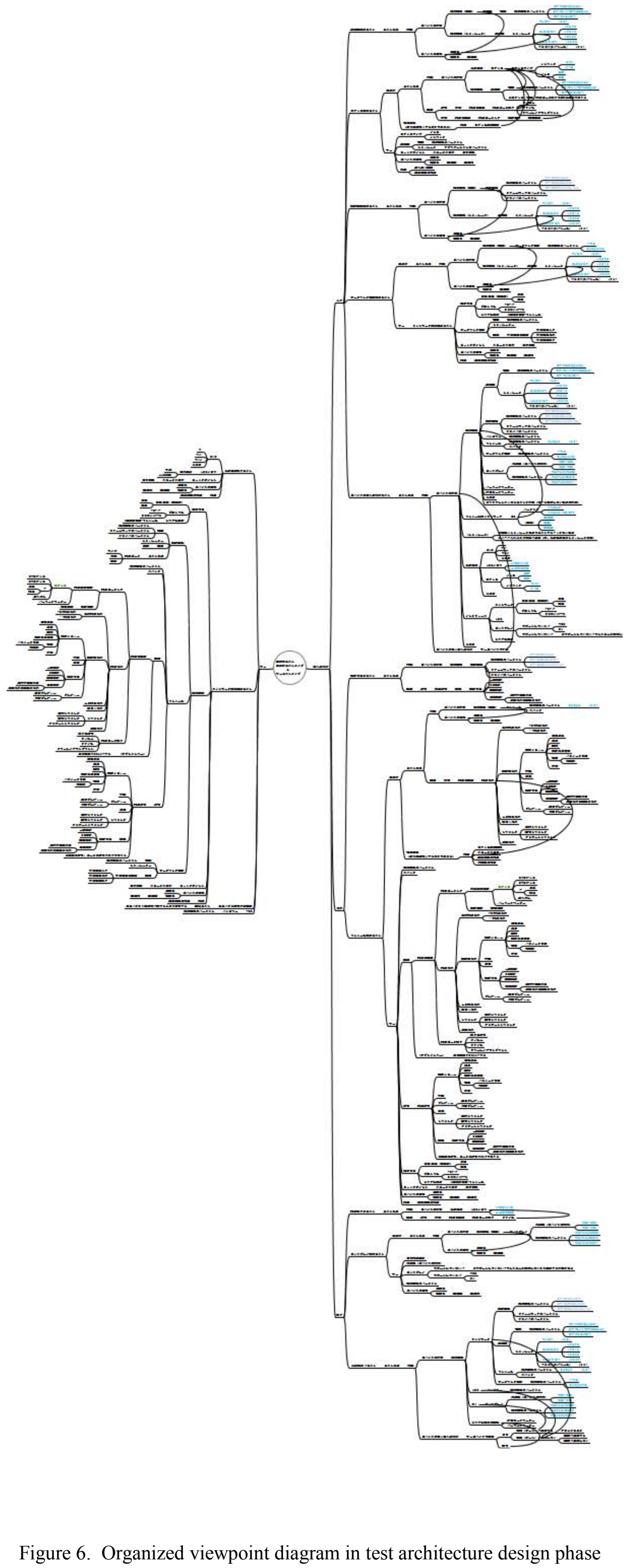


Figure 6. Organized viewpoint diagram in test architecture design phase

Fig.6 shows an organized viewpoint diagram in test architecture design phase. Fig 5 includes less viewpoints and more interaction, which means combinatorial relationship. Fig.6 includes more viewpoints and less interaction. Fig.6 is more complicated visually and conducts far more test cases because test cases conducted by interaction are proportional to multiplication among test cases conducted by each viewpoint. Fig 5 is larger visually but conducts less test cases because it has less interaction.

In test architecture design phase, we use some modeling techniques such as unification and re-define of viewpoint, unification and abstraction of interaction, clustering viewpoints, separation of key interaction and so on.

Each modeling technique is usually applied in test planning phase with experiences and heuristics. Viewpoint diagram can makes it easier to develop, accumulate and reuse experiences and heuristics as modeling techniques or test design patterns. In other words, viewpoint diagram will be a platform of test architecture design technology such as test design patterns, test architecture style, variability analysis of product line engineering and so on.

NGT can complement UML Test Profile because research and application of UTP mainly focus on test system architecture such as automation at present and NGT focuses on test suite architecture. NGT should harmonize UTP in future research.

VI. CONCLUSION

Software test recently becomes large-scale and complicated artifact as software itself. Research and practices has to be boosted such as test architecture. In this paper first we mentioned TDLC: Test Development Life Cycle, which includes test requirement design phase and test architecture design phase instead of test planning from engineering view. Second we discussed concepts of test architecture and propose NGT: Notation for Generic Testing, which is a set of concepts or notation for design of software test architecture. Viewpoint is discussed as a key concept of test architecture representing a group of test cases and test objective. And this paper gave an example of test architecture model. Finally this paper showed possibility that viewpoint diagram will be a platform of test architecture design technology such as test design patterns, test architecture style, variability analysis of product line engineering and so on.

REFERENCES

[1] OMG, "UML Testing Profile (UTP) Version 1.1 RTF - Beta 1," http://www.omg.org/spec/UTP/1.1/PDF/, June 2011.

[2] J. Bach, "Heuristic Test Strategy Model," http://www.satisfice.com /tools/satisfice-tsm-4p.pdf, March 2006.

[3] IEEE, "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems," IEEE Std 1471-2000, September 2000.

[4] M. Grochtmann, K. Grimm, "Classification trees for partition testing," Software Testing, Verification and Reliability, Vol. 3, Issue 2, pp. 63–82, June 1993.

Software Testing of Business Applications

Vijay Sampath
Tata Consultancy Services
Siruseri, Navalur Post, Kancheepuram District
Chennai - 603 103, India
Email: sampath.vijay@tcs.com

Vipul Shah
Tata Consultancy Services
54 B, Hadapsar Industrial Estate,
Pune - 411 013, India
Email: v.shah@tcs.com

As an IT service provider, we develop and maintain a large number of business applications for our customers. Functional and regression tests for business applications are often carried out by dedicated test teams using a black box approach. Test teams design and develop test scenarios and test cases from functional specifications and execute the test cases against the system. Automation levels are low, less than 20%, and that too for test execution only. In this position statement, we present a couple of challenges faced by the test teams. Instrumentation tools and techniques, along with measures need to be developed to address the same.

Coverage: Traditionally, one of key measures to describe the degree to which the application has been tested, is code coverage. A number of instrumentation tools, both commercial and open source, are available that provide path, decision, condition, function and other forms of code coverage measures. Several fault localization techniques have been developed that use coverage information to help developers find faults in the system. However, code coverage tools and measures are of little use to test teams that do not have access to the source code, which is primarily the case with functional test teams. In absence of tools and techniques, the test teams use manual approaches to gather coverage information. A common approach is to create a traceability matrix that links requirements to test cases and another matrix that links test scenarios to test cases. Coverage is computed in terms of requirements and scenarios covered by the test cases that are selected for execution.

As one can gather, there are several problems with the practiced approach.

- Both the requirement and scenario traceability matrices are usually maintained as spreadsheets and the coverage information is manually computed based on the test cases that have been executed. Maintainability and Scalability of such an approach is an issue for large and complex business applications. Test suites having several thousand test cases and hundreds of requirements are common.
- The above method does not address the completeness and correctness aspects. It is difficult to determine if the test cases adequately cover the requirements, if the requirements are in sync with the implementation, and if the requirements are complete.

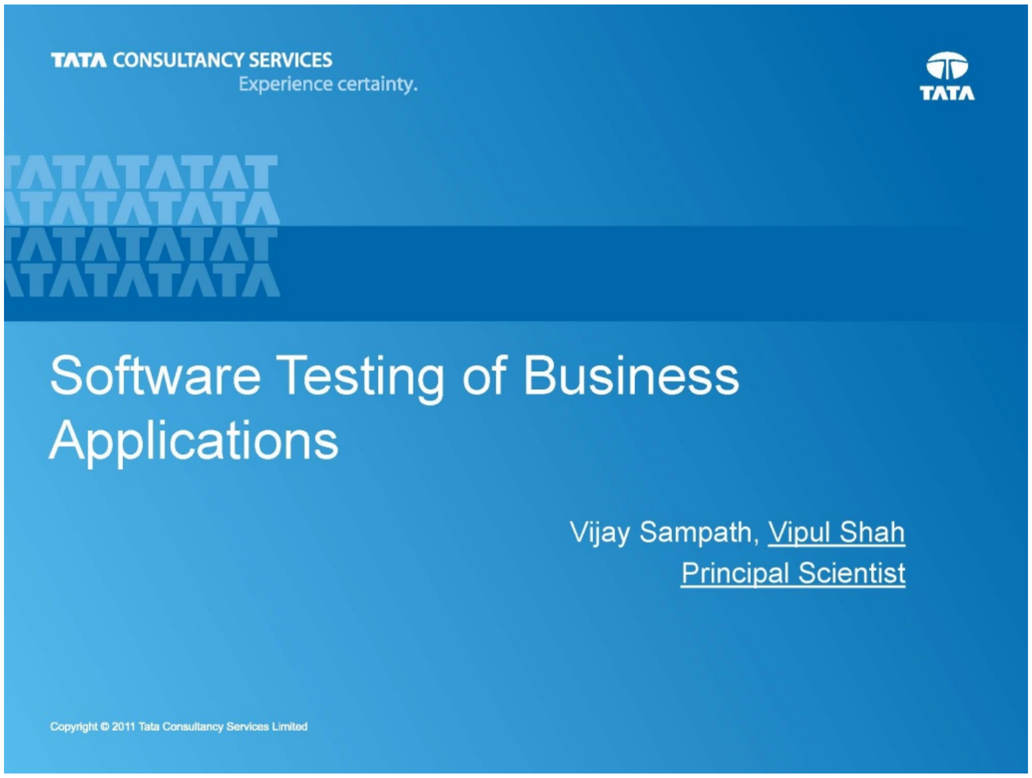
Clearly, there is a need to develop coverage measures for functional test teams that, like code coverage, provide relevant and appropriate coverage information in the absence of code. The measurement could be for elements that are well understood by test teams or in terms of input parameters and the coverage over possible range of values. Another common practice followed during regression, is to replay production logs. Coverage or rather absence of coverage information becomes critical to understand the scenarios that were tested by the production log replay.

Test Selection: While several test selection methods have been developed, it is common in the industry to use a risk based approach to test selection. A subset of test cases is identified for creation and execution, based on the risks foreseen. The test cases are prioritized to be executed in the order of minimizing the higher risks first, and then the lower order risks. When defects are identified, and the risks mitigated or eliminated, the purpose of designing and executing that test case is achieved. However, if a test case does not detect a defect, then while the confidence level in the scenario tested increases, in reality, it is a waste of time and effort to execute the test case and adds to the cost of quality.

It is a practice to classify the requirements based on their business criticality (impact on failure) and probability of failure, and quantified risk. While impact on failure is easier to enumerate, based on the business context, by the subject matter expert, the probability of failure computation is difficult as it depends on several factors, some of which are non-deterministic.

The challenge, therefore, lies in designing a risk model that lead to selection of those test cases which increase the probability of finding defects. While this may be an extremely difficult goal to achieve, what measures do we adopt that would help us quantify and measure risk dynamically? What risk model do we need to bring us closer to our goal? Also, over a period of time, high risk scenarios are likely to be tested more frequently and hence, less likely to detect defects. Can the risk model take this factor into account?

978-0-7695-4743-5/12 $26.00 © 2012 IEEE
DOI 10.1109/SERE-C.2012.48

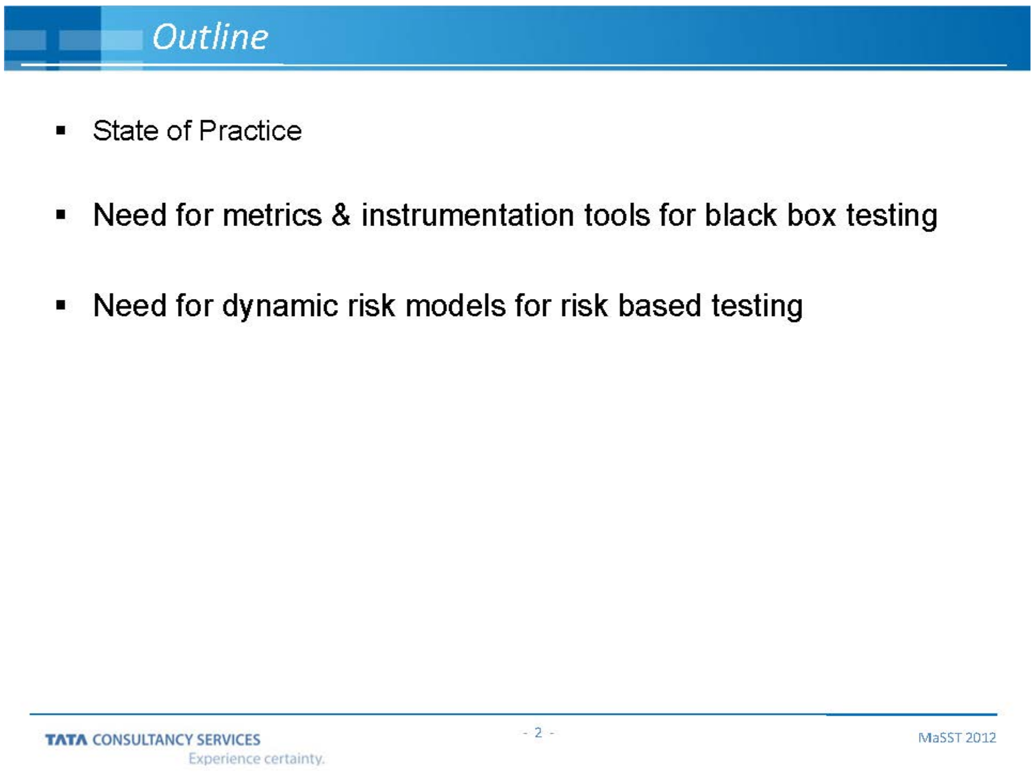


TATA CONSULTANCY SERVICES
Experience certainty.

TATA

Software Testing of Business Applications

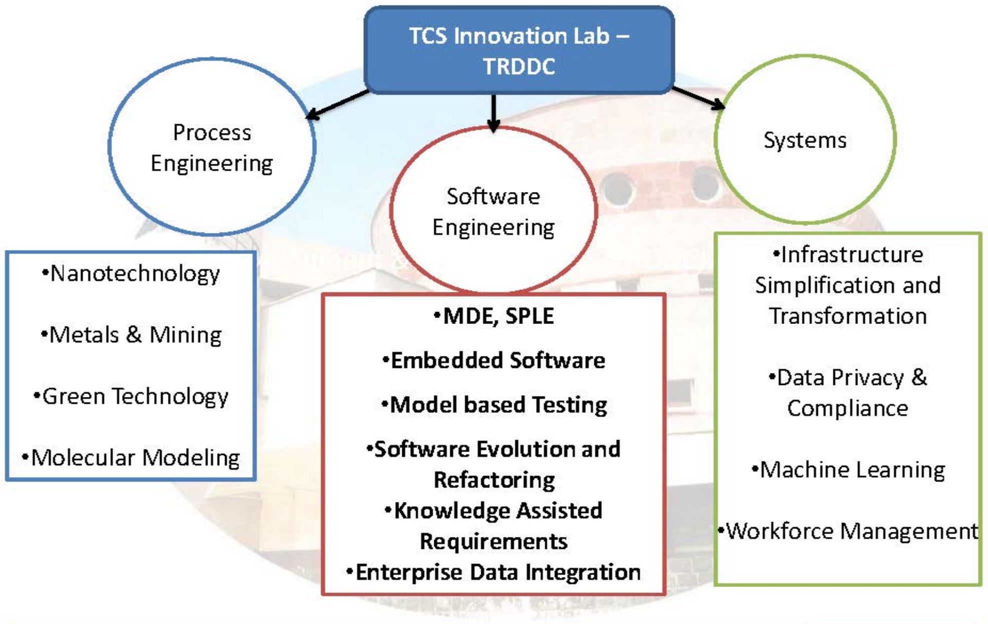
Vijay Sampath, Vipul Shah
Principal Scientist



Outline

- State of Practice
- Need for metrics & instrumentation tools for black box testing
- Need for dynamic risk models for risk based testing

Tata Research Development & Design Center (TRDDC)



- Nanotechnology
- Metals & Mining
- Green Technology
- Molecular Modeling

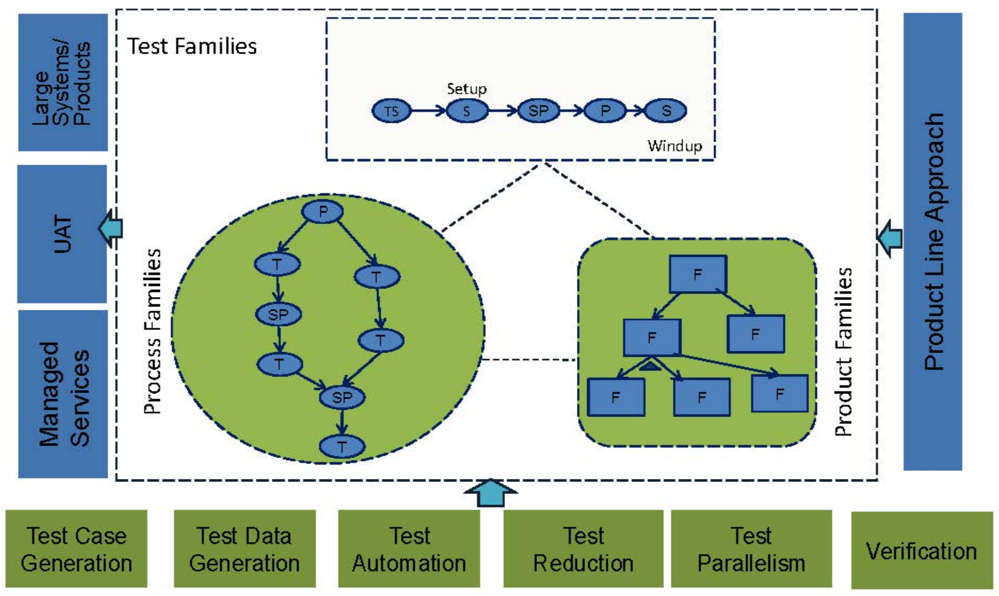
- **MDE, SPLE**
- **Embedded Software**
- **Model based Testing**
- **Software Evolution and Refactoring**
- **Knowledge Assisted Requirements**
- **Enterprise Data Integration**

- Infrastructure Simplification and Transformation
- Data Privacy & Compliance
- Machine Learning
- Workforce Management

TATA CONSULTANCY SERVICES
Experience certainty.

MaSST 2012

Software Testing R&D



TATA CONSULTANCY SERVICES
Experience certainty.

MaSST 2012

Independent Testing Services

- Independent V&V
- System Integration testing onwards
- Black box – Distributed teams, no or limited access to code
- Testing driven by requirements
- Risk based testing

- Systems - Complex architectures / multiple platforms & languages, databases, GUIs

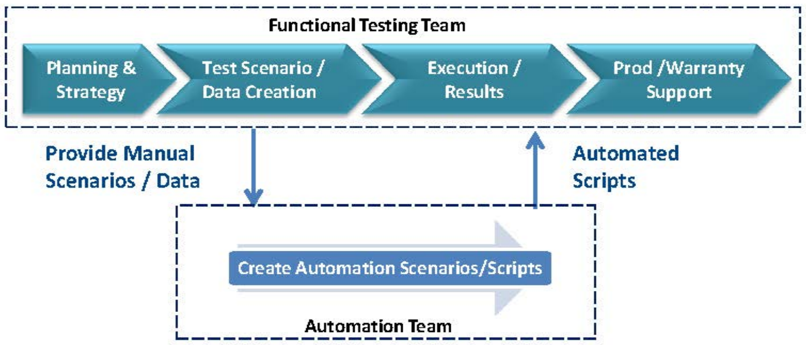
Processes & Enabling Measurements

Testing Process – Current State

- Mature Process
- Manually managed in large number of instances
- Low automation levels
- Metrics tend to be process oriented

Testing Process

- Domain SMEs from functional testing team create manual scenarios
- Hybrid Automation Framework
- Test Management & Test Automation tools
- Low automation levels



Domain SMEs usually not involved in automation scripting

Test Automation: In practice

- Automation @ 20%*
  - Dominated by single tool – 73%
- Top cited reasons for low automation
  - Change in business priorities for investments
  - No actual business case (longer break-even than expected or no ROI at all)
  - Constantly changing applications making maintenance of automation test suites costlier
- Automation seen as additional activity needing specialized resources
- Maintenance effort (Scripts/Frameworks): 50% to 90%
- Automation improves execution time by approx 50%

2011, TCS Internal survey, 900+ projects, conducted by Assurance Services Unit

Assurance - Coverage?

Measuring Coverage for black box testing

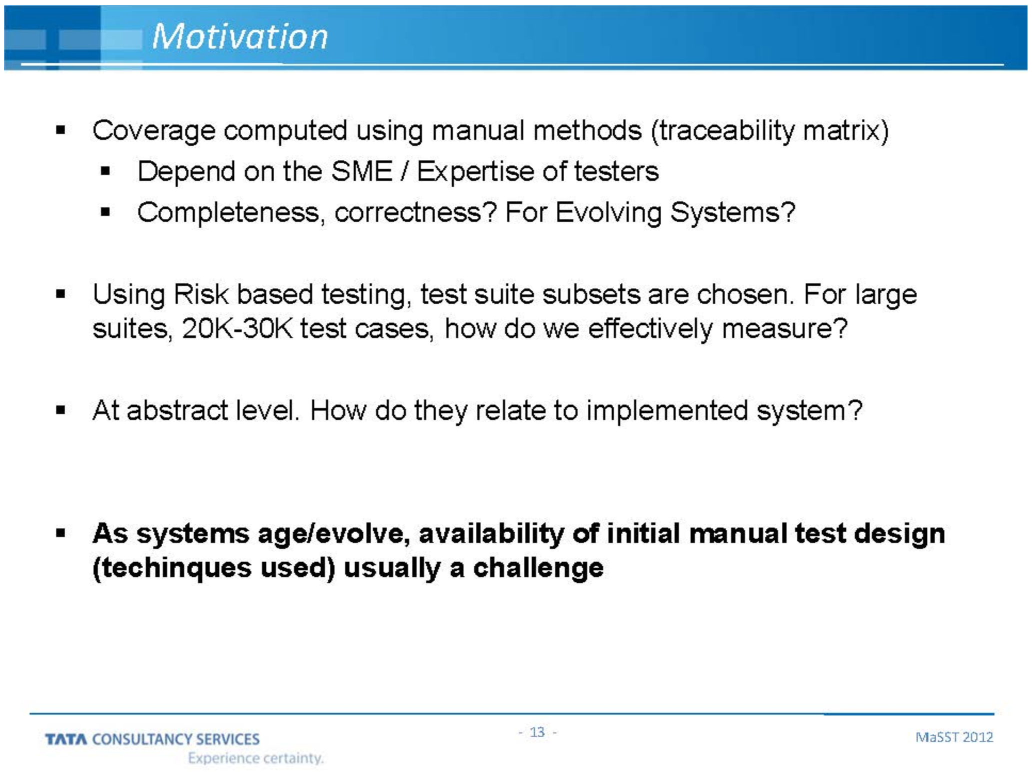
- Coverage of a software test suite is the extent of the size of the system that is assured/ tested for by a particular software test suite
- Coverage = $\dfrac{\text{Extent of system covered by the test}}{\text{Total size of the system}}$
- **Challenges:**
  - Unit of measure is difficult to arrive for a black box testing perspective
  - Some of the current measures used in the industry:
    - Function points
    - # of requirements
    - Usecase points
  - Common Practice to maintain traceability matrix

How does this higher level coverage relate to physical system?

Completeness & Correctness of Requirements to Test case mapping?

Case Study

- Large Web based application – Healthcare
- 2000+ Functional tests
- Regression – 1 week with multiple testers (average 3)
- Code instrumentation revealed 53% coverage of the system
- Through change of input parameter values coverage was increased to 67%
- No new defects found in the process!



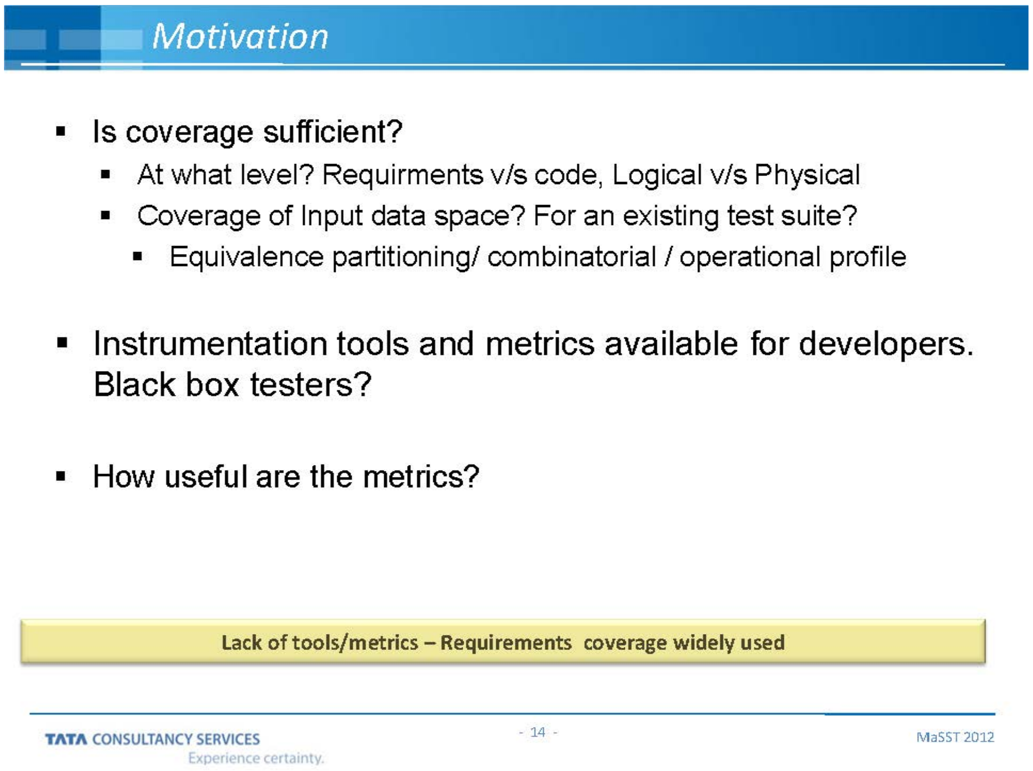
Motivation

- Coverage computed using manual methods (traceability matrix)
  - Depend on the SME / Expertise of testers
  - Completeness, correctness? For Evolving Systems?
- Using Risk based testing, test suite subsets are chosen. For large suites, 20K-30K test cases, how do we effectively measure?
- At abstract level. How do they relate to implemented system?
- **As systems age/evolve, availability of initial manual test design (techinques used) usually a challenge**

TATA CONSULTANCY SERVICES
Experience certainty.

- 13 -

MaSST 2012



Motivation

- Is coverage sufficient?
  - At what level? Requirments v/s code, Logical v/s Physical
  - Coverage of Input data space? For an existing test suite?
    - Equivalence partitioning/ combinatorial / operational profile
- Instrumentation tools and metrics available for developers. Black box testers?
- How useful are the metrics?

Lack of tools/metrics – Requirements coverage widely used

TATA CONSULTANCY SERVICES
Experience certainty.

- 14 -

MaSST 2012

TATA

Assurance - Risk models?

TATA CONSULTANCY SERVICES
Experience certainty.

State of Practice

- Requirements classified
  - Business criticality (Impact of failure)
  - Probablity of failure
  - Quantified risks

- Risk Models used for test selection
  - Mitigate risks
  - Increase probablity of finding defect
  - Over a period of time, test suites fail to uncover new defects.

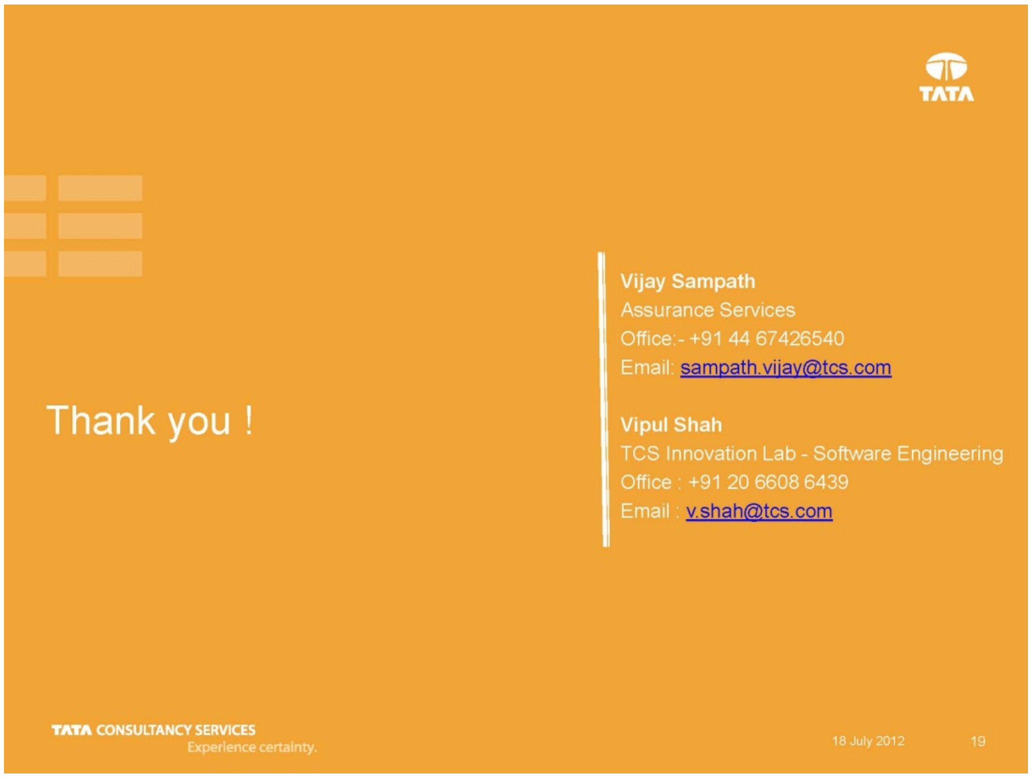
- Probablity of failure difficult to compute as failure can occur due to several independent as well as interacting causes
  - Contributing factors: System complexity, developer skills, analyst skills …

Challenges

- Evolving Systems
- Risks change, risk classification depends on the SME / Expertise of testers
- Are there simple probablistic risk models/tools to compute probablity of failure?
- What is the risk of not exetuting test cases identified by risk based testing method?
- Can we have dynamic risk models? Taking into account above?
- Risks v/s cost models and test selection accordingly?

Summarizing …

- Black box testing coverage metrics
  - High level, not dynamically from system
- Need for coverage metrics & instrumentation tools for black box testing
- Risk based testing method over a period of time reaches saturation, failing to reveal new defects
- Need to extend risk based testing



Thank you !

Vijay Sampath
Assurance Services
Office:- +91 44 67426540
Email: sampath.vijay@tcs.com

Vipul Shah
TCS Innovation Lab - Software Engineering
Office : +91 20 6608 6439
Email : v.shah@tcs.com

TATA CONSULTANCY SERVICES
Experience certainty.

18 July 2012 19

Why Fuzzing (Still) Works

Allen D. Householder,
Software Engineering Institute, CERT
adh@cert.org

Abstract:

Despite improvements in grammar-aware (i.e., “white box”) fuzz testing, static source code analysis, and other techniques, mutational (“black box”) fuzz testing remains an effective way to discover vulnerabilities in fielded software.

Our experience with developing software tools and techniques that implement a naïve approach to mutational fuzz testing demonstrates that it yields effective results with low infrastructure cost and developer startup time and effort. (Dormann, 2010) Additionally, in coordinating information about software vulnerabilities and practical attacks we have bserved that these techniques are still widely used by attackers to discover new vulnerabilities. (CERT, 2012)

We propose several possible reasons for the continued effectiveness of this approach. Perhaps most importantly, although black box fuzzing is formally specified as an activity in some software development life cycles, there is little consistency in benchmarks and metrics of efficacy. In the current state of the practice, a software developer has ittle insight into how to answer the question of “how do I know if I have fuzzed enough?” A related issue is that it is easy for developers o perform many iterations of ineffective mutation testing, resulting in low code coverage and the erroneous conclusion that their software has undergone some rigorous testing process.

Second, even "safe" development languages and environments typically require interaction with, or incorporation of, legacy software components in unmanaged native code. Black box fuzzing is particularly effective at revealing the underlying defects that arise from thisintegration.

Finally, despite being readily scalable, an industrialization of this approach has not enjoyed widespread adoption in the testing of software.

Recent advances in white box testing and static analysis techniques are promising and begin to bridge the gap towards formal software verification. However in their present state such techniques may present a challenge for resource constrained developers to implement effectively. A fast, naïve approach leverages the fact that computation is cheap (and only getting cheaper), while reasoning about complex software systems is difficult and time-intensive. Black box testing can be initiated with a minimum of expertise yet when coupled with simple stochastic techniques (Householder & Foote, 2012) can act as a stopgap until more advanced approaches can be realized.

We feel that this approach fits naturally with the practical observation that security quality assurance can be better measured in terms of conditional probability than a collection of check

boxes. Were these techniques to be incorporated into the SDL, it may be feasible to take blind mutational fuzz testing off the table as a viable means of vulnerability discovery in fielded software.

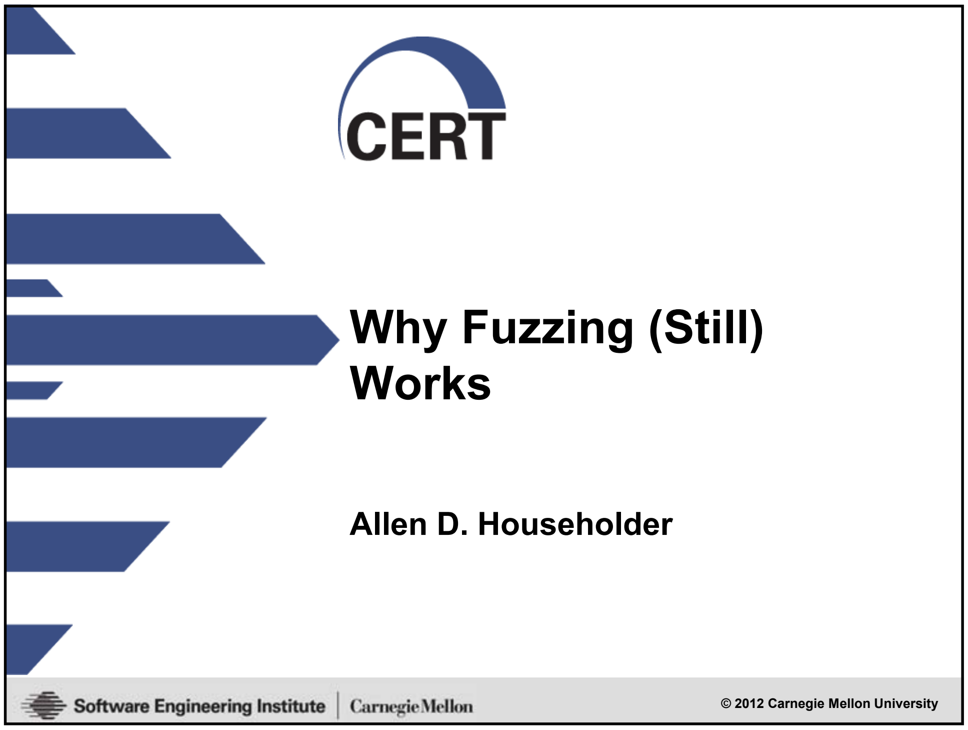
We propose to discuss the above in the context of our experiences with the CERT Basic Fuzzing Framework in a short talk for the workshop.

References :

CERT. (2012). Retrieved Feb 23, 2012, from US-CERT Vulnerability Notes: http://www.kb.cert.org/vuls

Dormann, W. (2010, May 26). CERT Basic Fuzzing Framework. Retrieved Feb 23, 2012, from http://www.cert.org/blogs/certcc/2010/05/cert_basic_fuzzing_framework.html

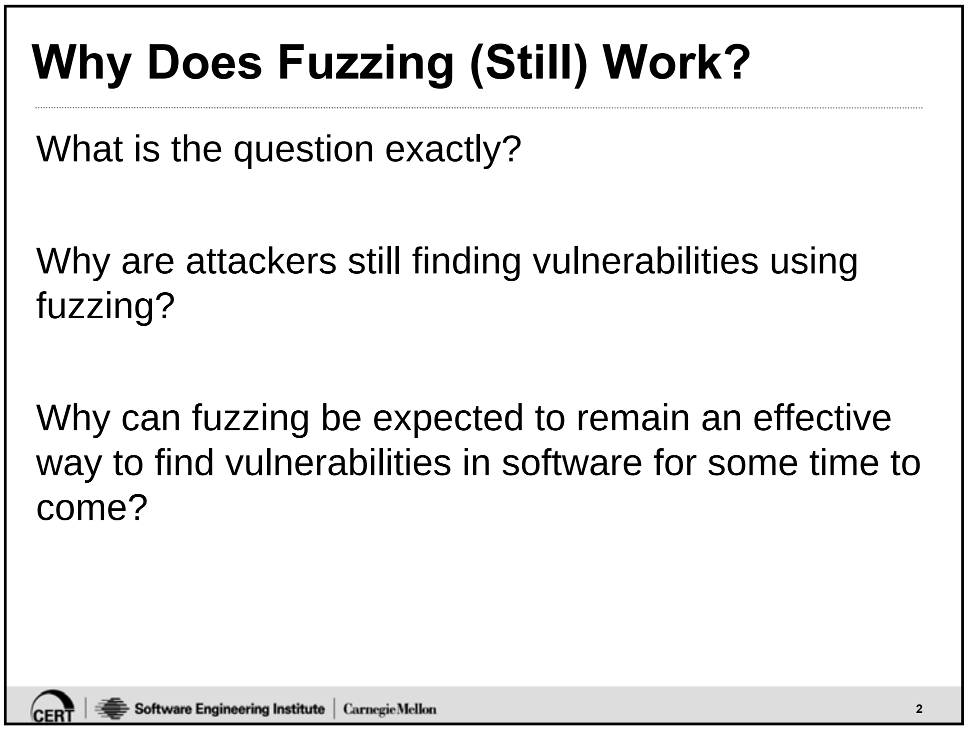
Householder, A., & Foote, J. (2012). Probability-based parameter selection for dynamic randomized-input functional testing. [Publication pending].



Why Fuzzing (Still) Works

Allen D. Householder

Software Engineering Institute | Carnegie Mellon



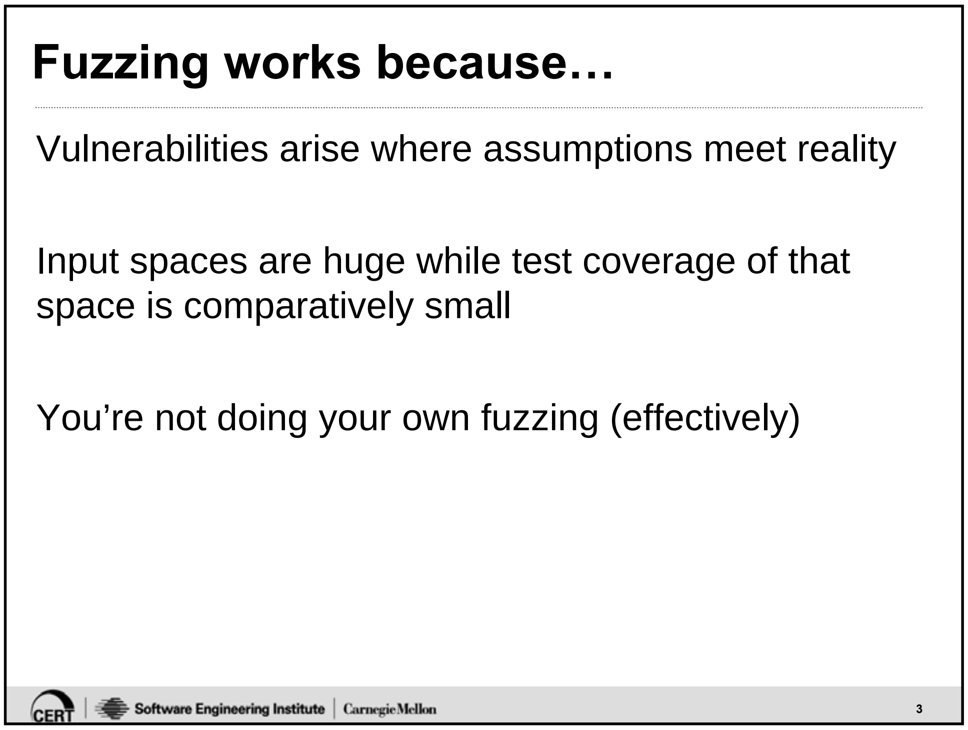
Why Does Fuzzing (Still) Work?

What is the question exactly?

Why are attackers still finding vulnerabilities using fuzzing?

Why can fuzzing be expected to remain an effective way to find vulnerabilities in software for some time to come?

CERT | Software Engineering Institute | Carnegie Mellon 2



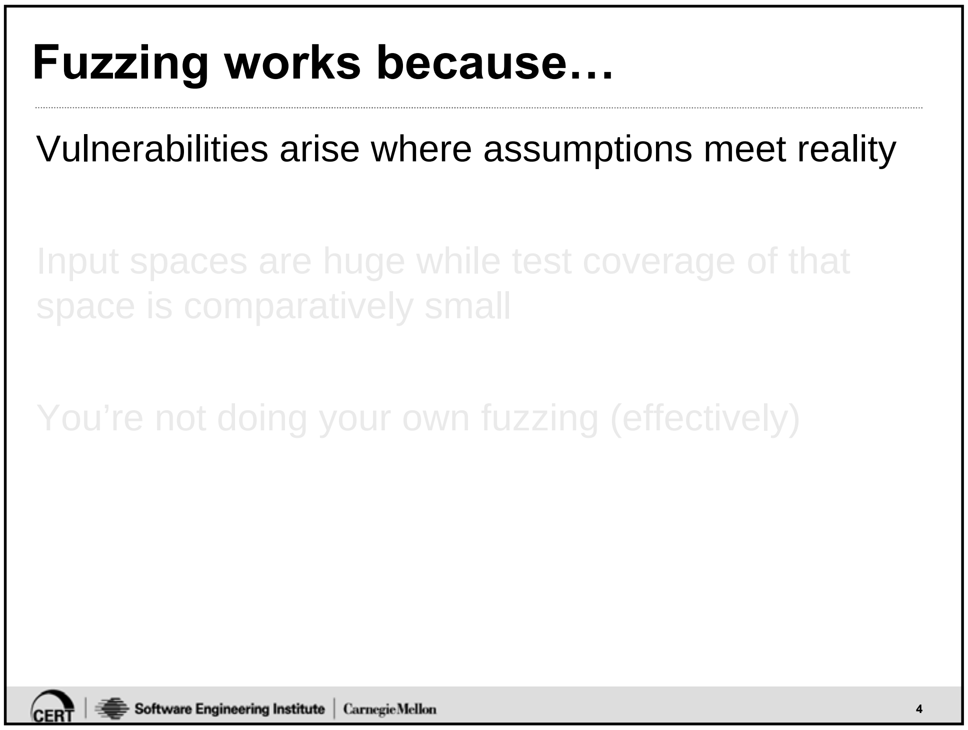
Fuzzing works because…

Vulnerabilities arise where assumptions meet reality

Input spaces are huge while test coverage of that space is comparatively small

You're not doing your own fuzzing (effectively)

Software Engineering Institute | CarnegieMellon 3



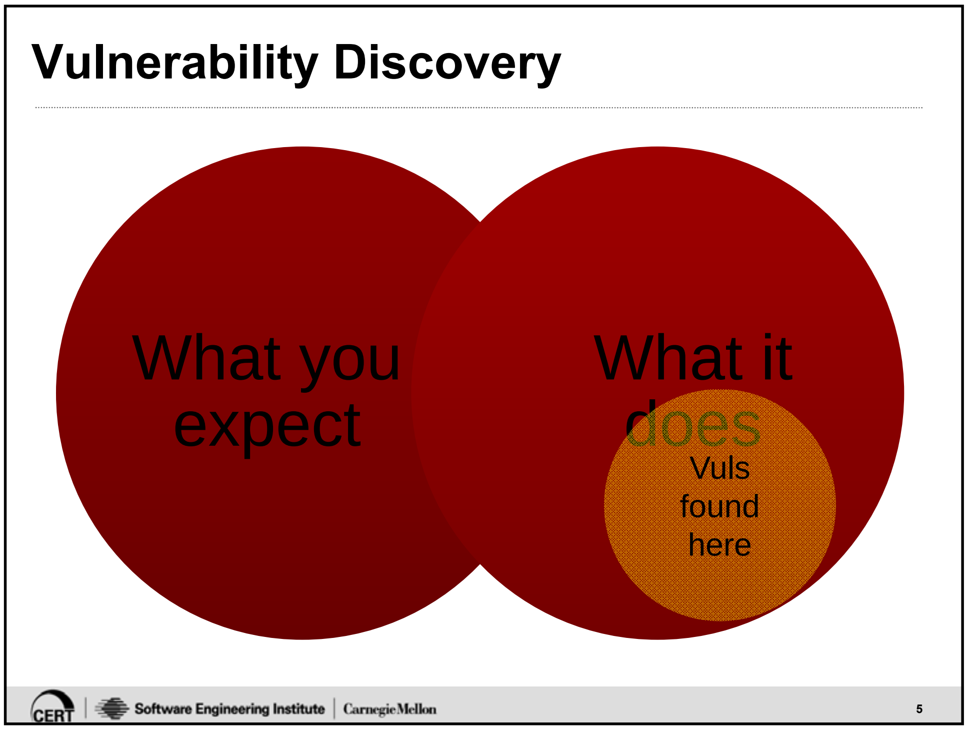
Fuzzing works because…

Vulnerabilities arise where assumptions meet reality

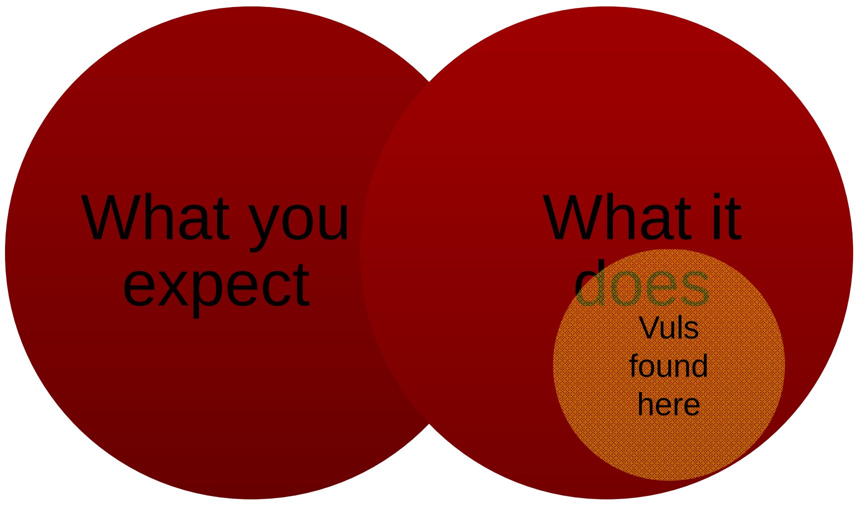
Input spaces are huge while test coverage of that space is comparatively small

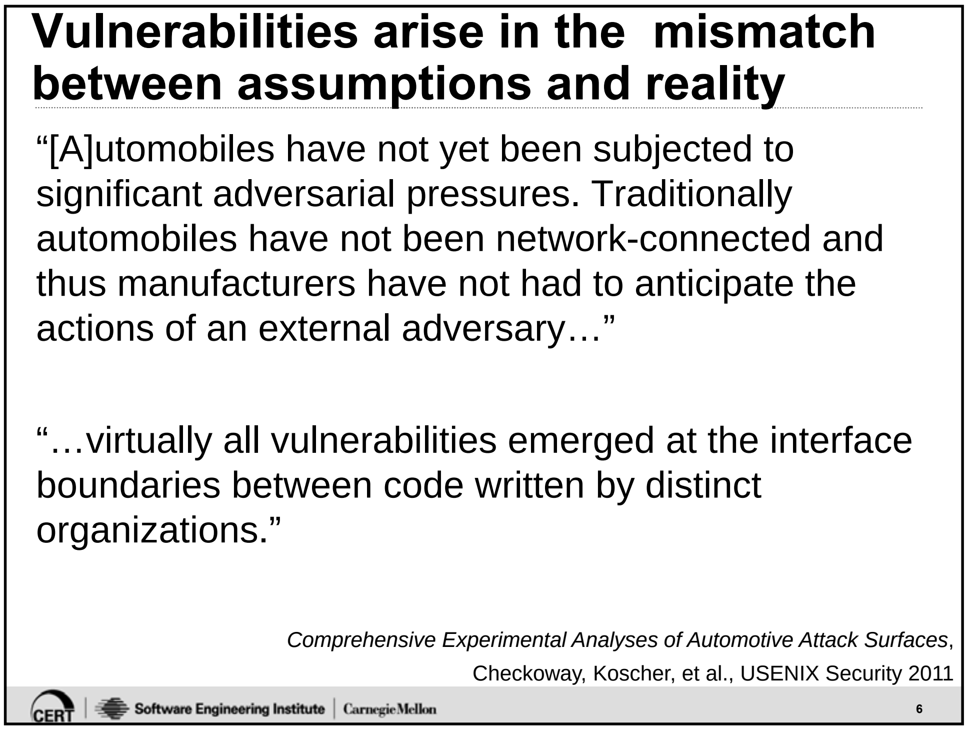
You're not doing your own fuzzing (effectively)

Software Engineering Institute | CarnegieMellon 4



Vulnerability Discovery



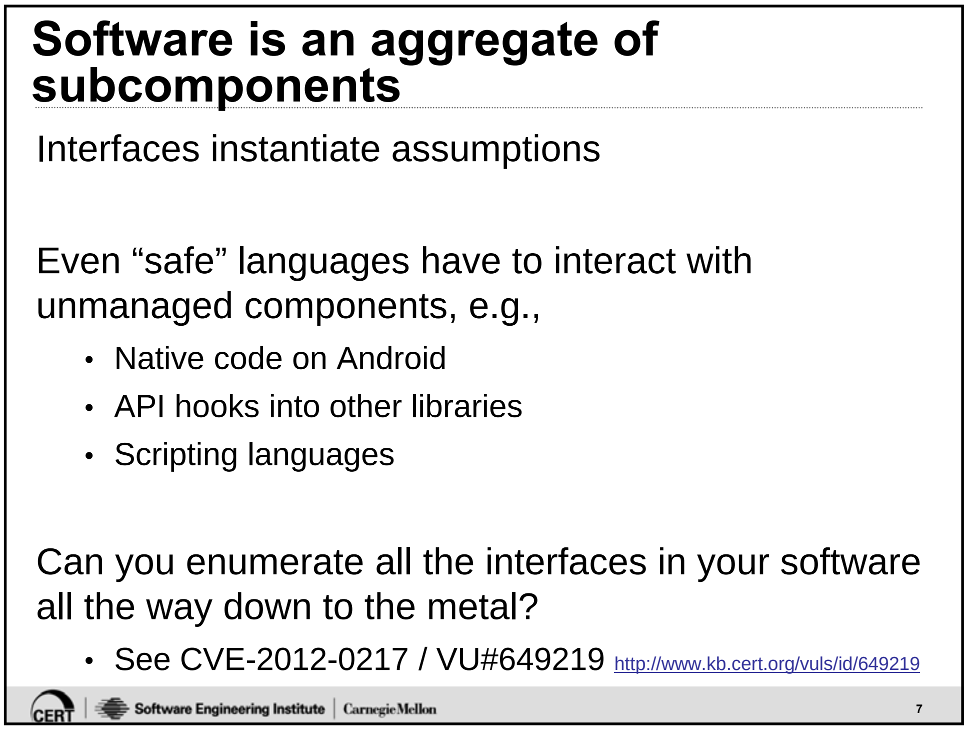


Vulnerabilities arise in the mismatch between assumptions and reality

"[A]utomobiles have not yet been subjected to significant adversarial pressures. Traditionally automobiles have not been network-connected and thus manufacturers have not had to anticipate the actions of an external adversary…"

"…virtually all vulnerabilities emerged at the interface boundaries between code written by distinct organizations."

Comprehensive Experimental Analyses of Automotive Attack Surfaces, Checkoway, Koscher, et al., USENIX Security 2011



Software is an aggregate of subcomponents

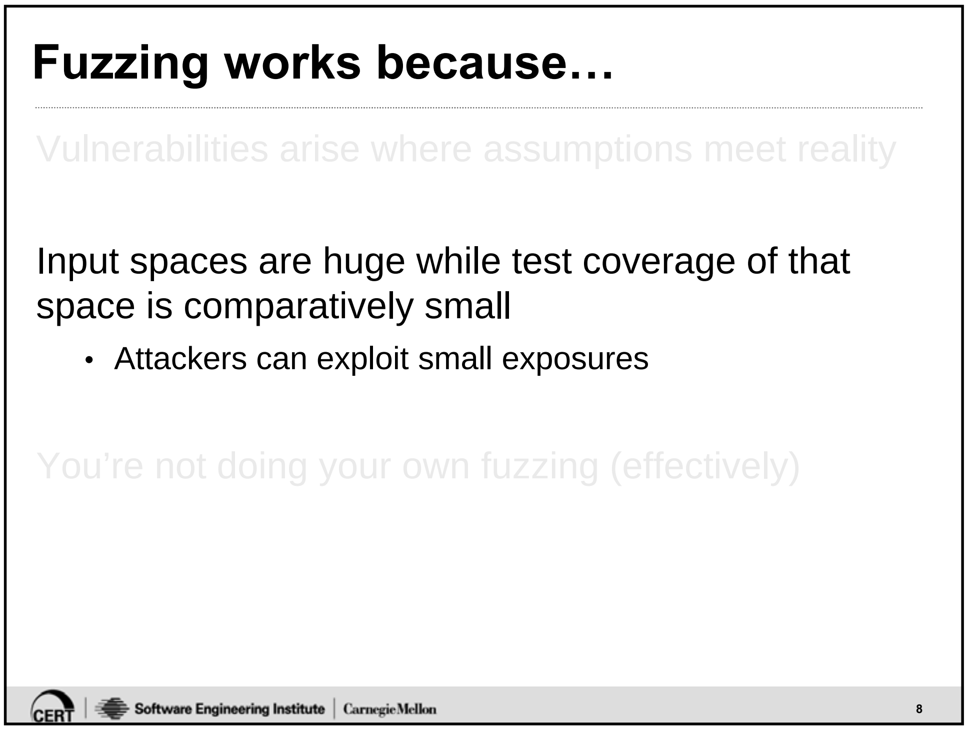
Interfaces instantiate assumptions

Even “safe” languages have to interact with unmanaged components, e.g.,

- Native code on Android
- API hooks into other libraries
- Scripting languages

Can you enumerate all the interfaces in your software all the way down to the metal?

- See CVE-2012-0217 / VU#649219 http://www.kb.cert.org/vuls/id/649219



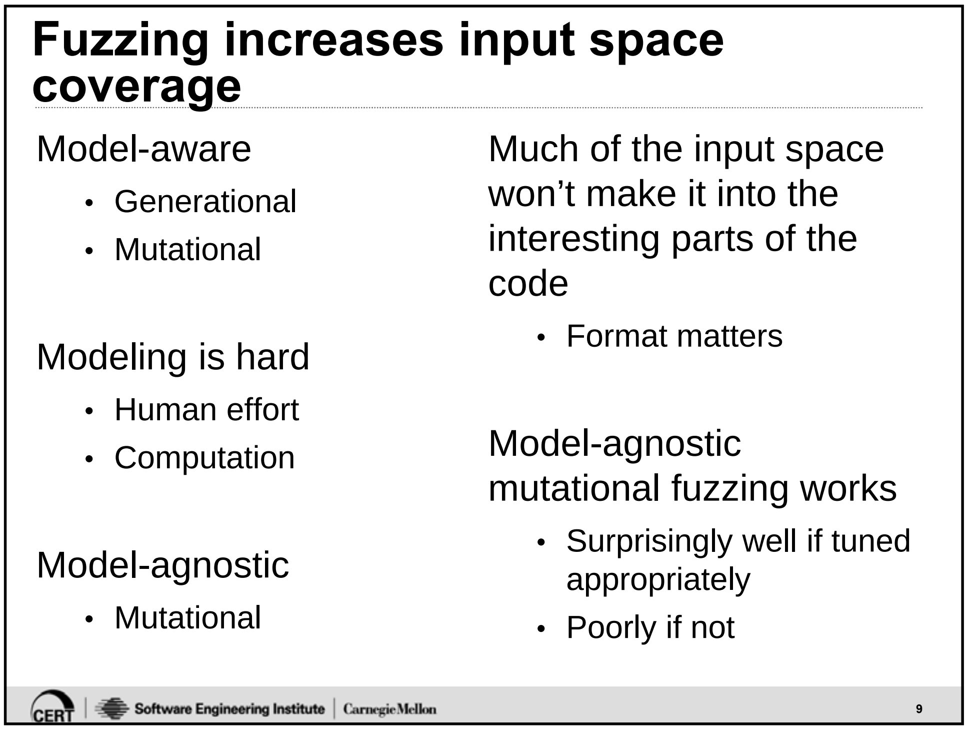
Fuzzing works because…

Vulnerabilities arise where assumptions meet reality

Input spaces are huge while test coverage of that space is comparatively small

- Attackers can exploit small exposures

You’re not doing your own fuzzing (effectively)



Fuzzing increases input space coverage

Model-aware

- Generational
- Mutational

Modeling is hard

- Human effort
- Computation

Model-agnostic

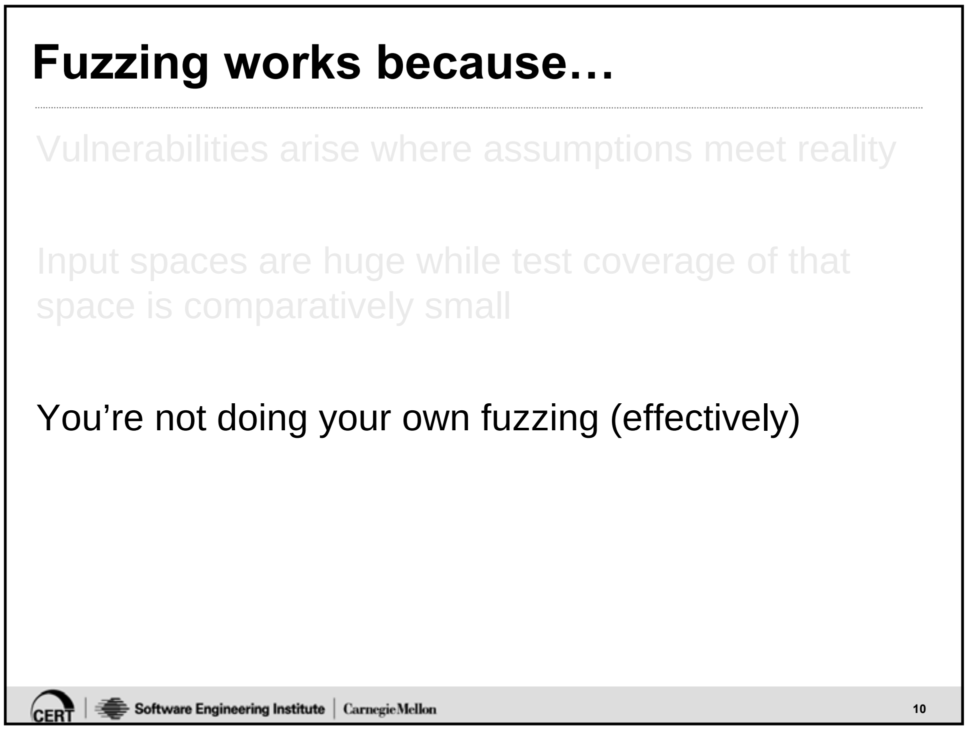
- Mutational

Much of the input space won't make it into the interesting parts of the code

- Format matters

Model-agnostic mutational fuzzing works

- Surprisingly well if tuned appropriately
- Poorly if not

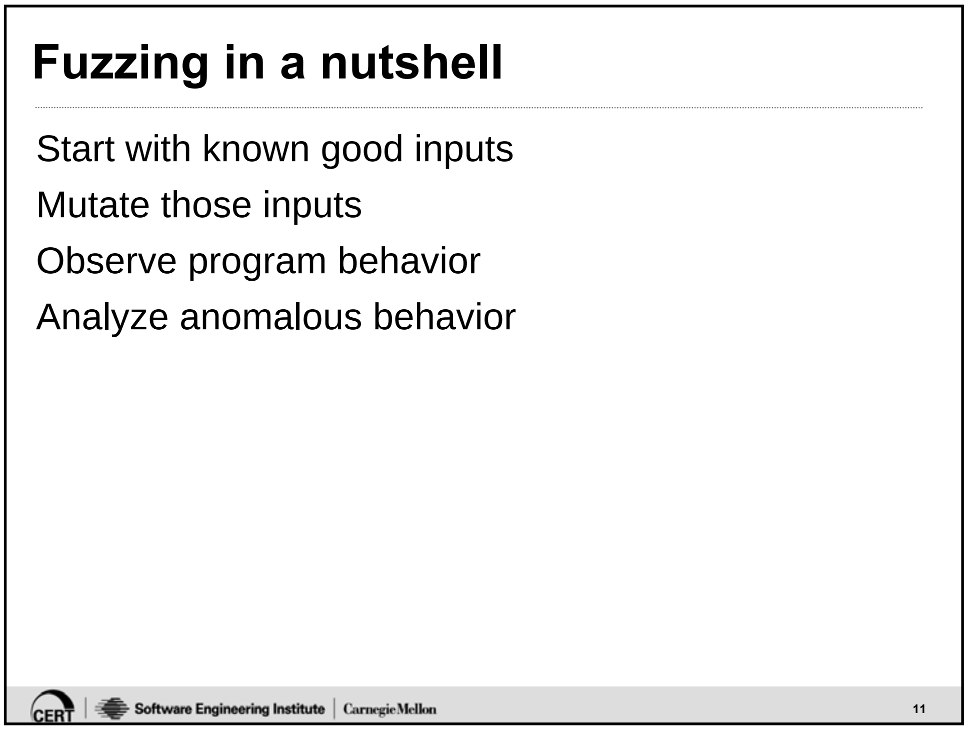


Fuzzing works because…

Vulnerabilities arise where assumptions meet reality

Input spaces are huge while test coverage of that space is comparatively small

You're not doing your own fuzzing (effectively)



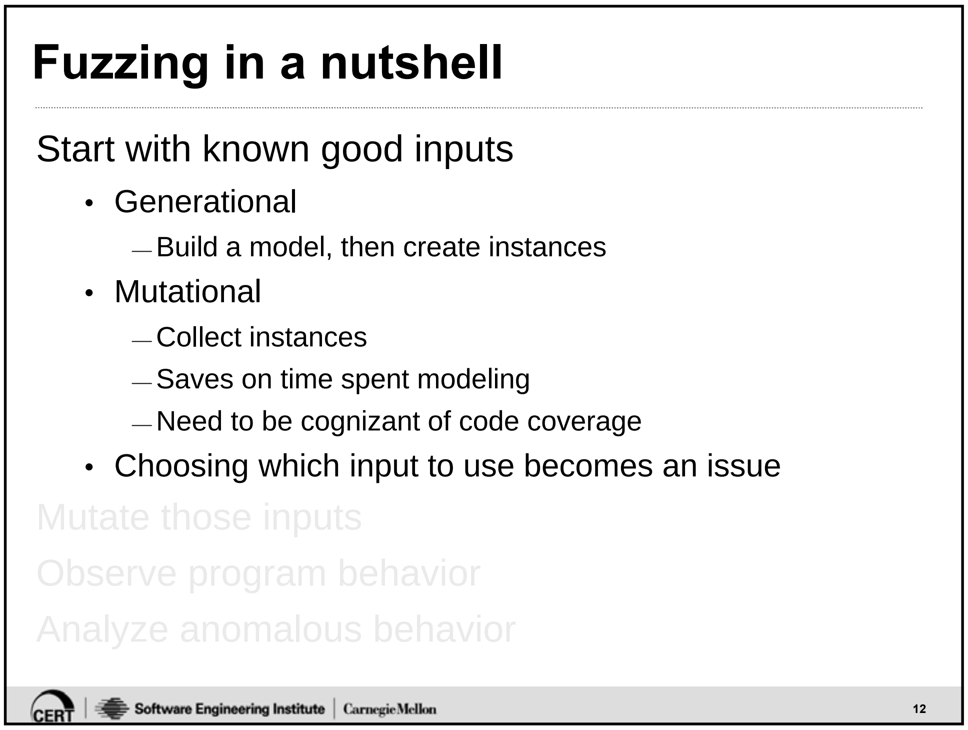
Fuzzing in a nutshell

Start with known good inputs

Mutate those inputs

Observe program behavior

Analyze anomalous behavior



Fuzzing in a nutshell

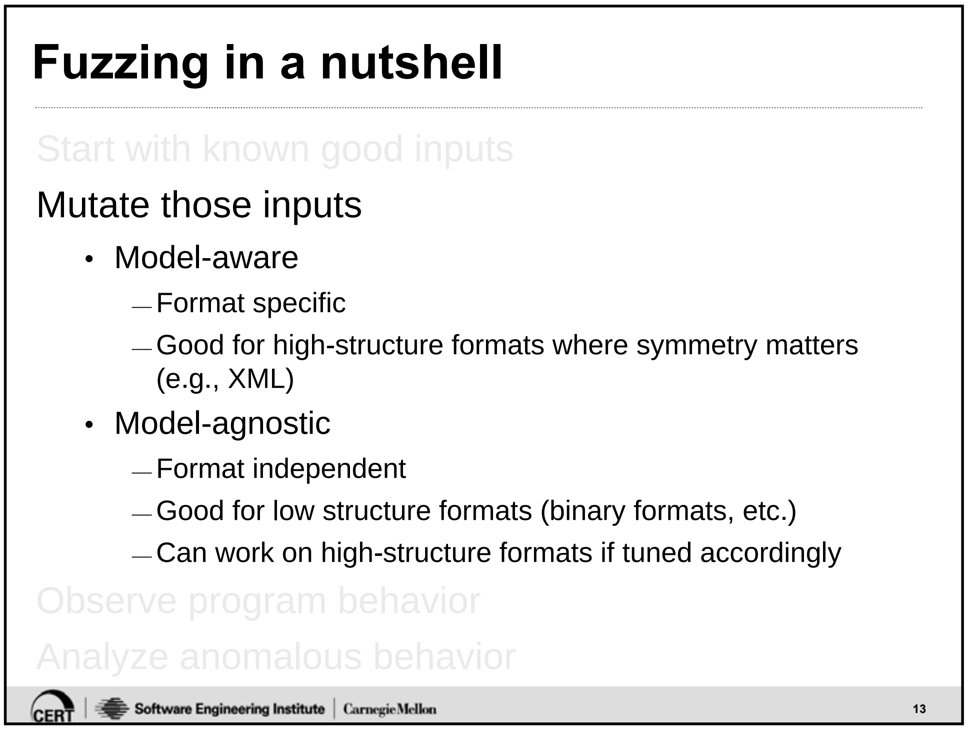
Start with known good inputs

- Generational
  - Build a model, then create instances
- Mutational
  - Collect instances
  - Saves on time spent modeling
  - Need to be cognizant of code coverage
- Choosing which input to use becomes an issue

Mutate those inputs

Observe program behavior

Analyze anomalous behavior



Fuzzing in a nutshell

Start with known good inputs

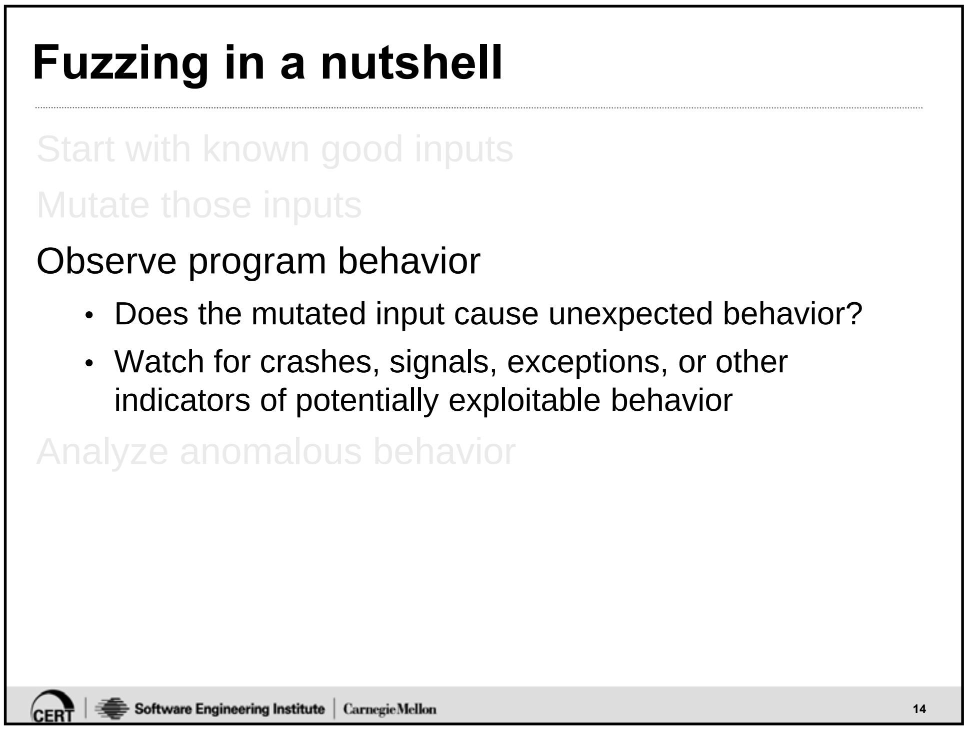
Mutate those inputs

- Model-aware
  - Format specific
  - Good for high-structure formats where symmetry matters (e.g., XML)
- Model-agnostic
  - Format independent
  - Good for low structure formats (binary formats, etc.)
  - Can work on high-structure formats if tuned accordingly

Observe program behavior

Analyze anomalous behavior

CERT | Software Engineering Institute | CarnegieMellon 13



Fuzzing in a nutshell

Start with known good inputs

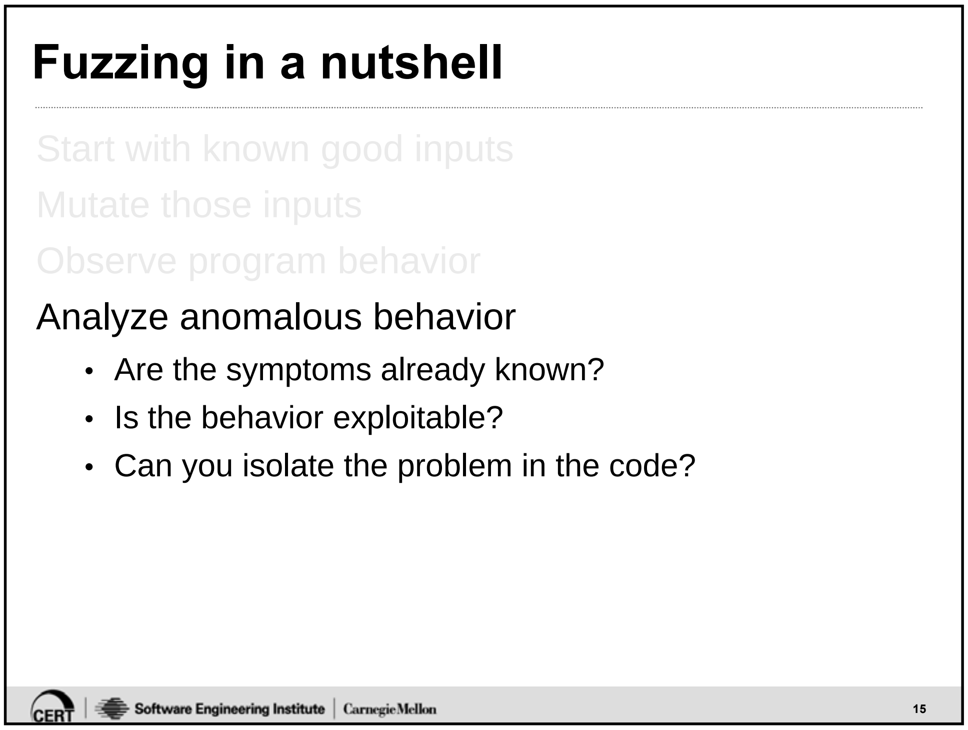
Mutate those inputs

Observe program behavior

- Does the mutated input cause unexpected behavior?
- Watch for crashes, signals, exceptions, or other indicators of potentially exploitable behavior

Analyze anomalous behavior

CERT | Software Engineering Institute | CarnegieMellon 14



Fuzzing in a nutshell

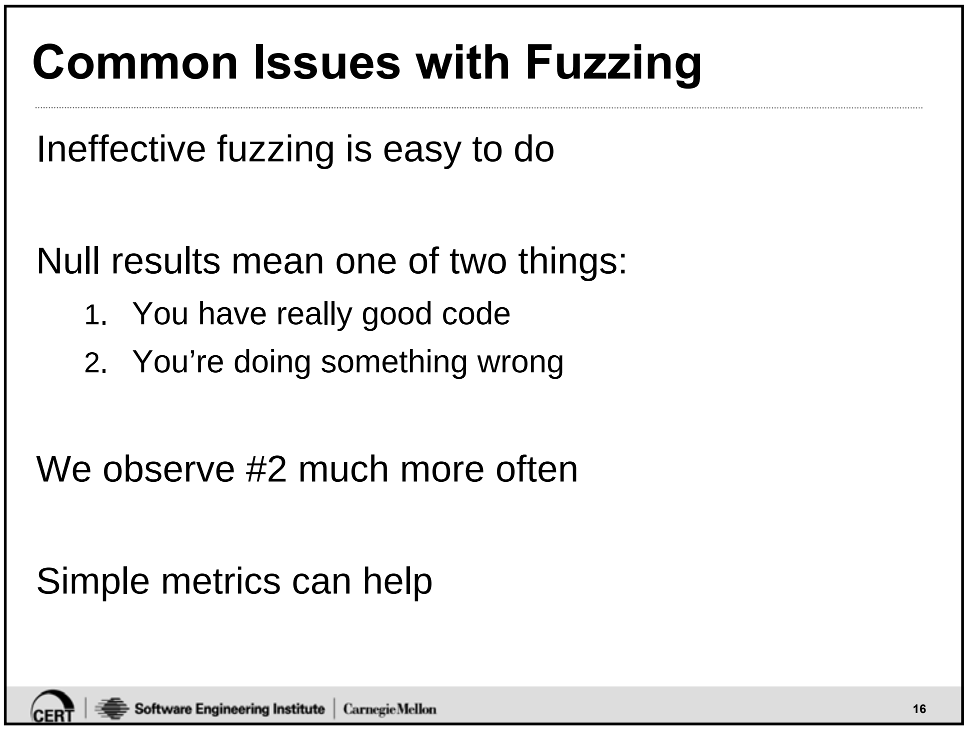
Start with known good inputs

Mutate those inputs

Observe program behavior

Analyze anomalous behavior

- Are the symptoms already known?
- Is the behavior exploitable?
- Can you isolate the problem in the code?



Common Issues with Fuzzing

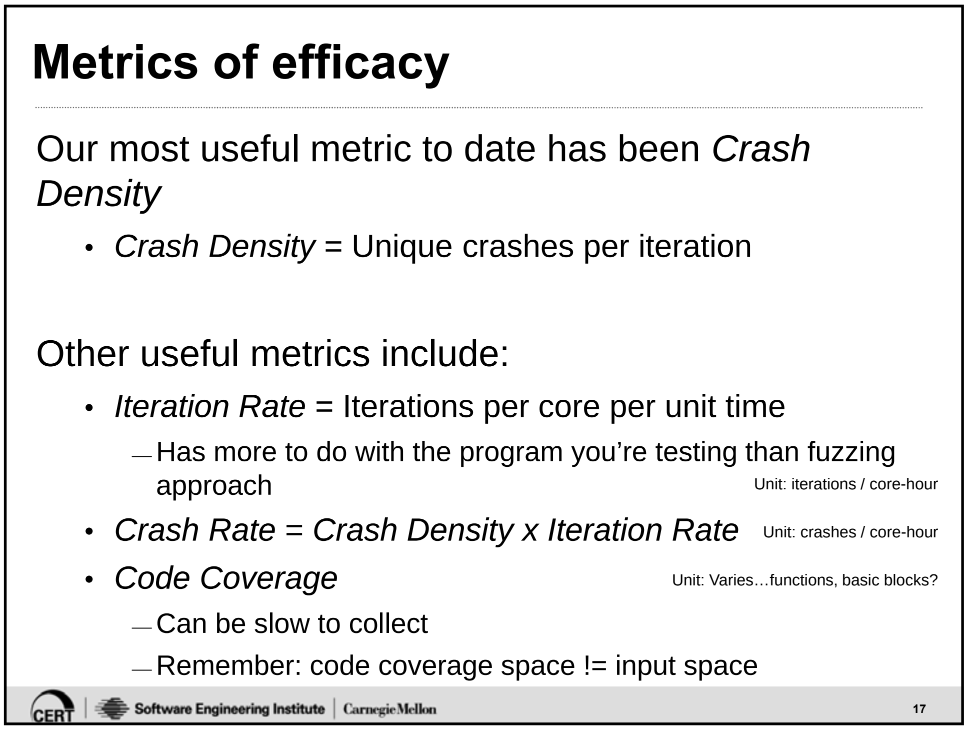
Ineffective fuzzing is easy to do

Null results mean one of two things:

1. You have really good code
2. You’re doing something wrong

We observe #2 much more often

Simple metrics can help



Metrics of efficacy

Our most useful metric to date has been *Crash Density*

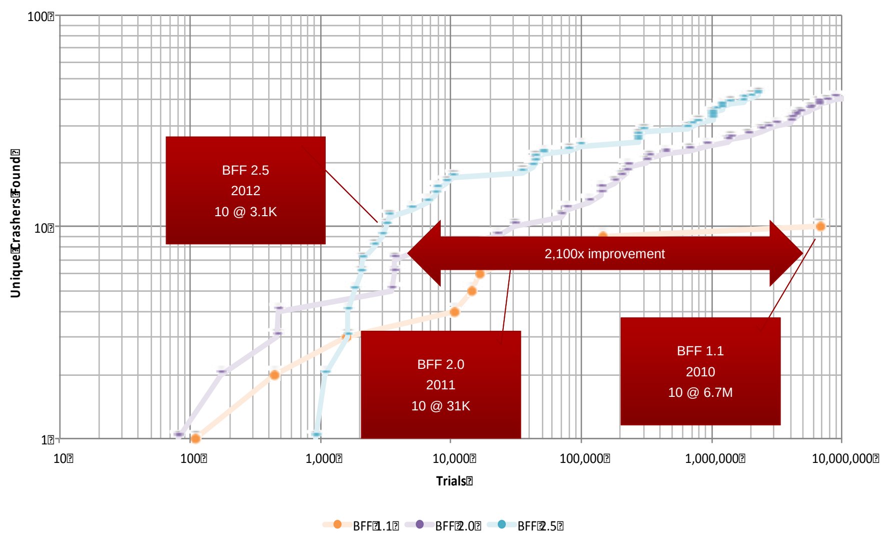
- *Crash Density* = Unique crashes per iteration

Other useful metrics include:

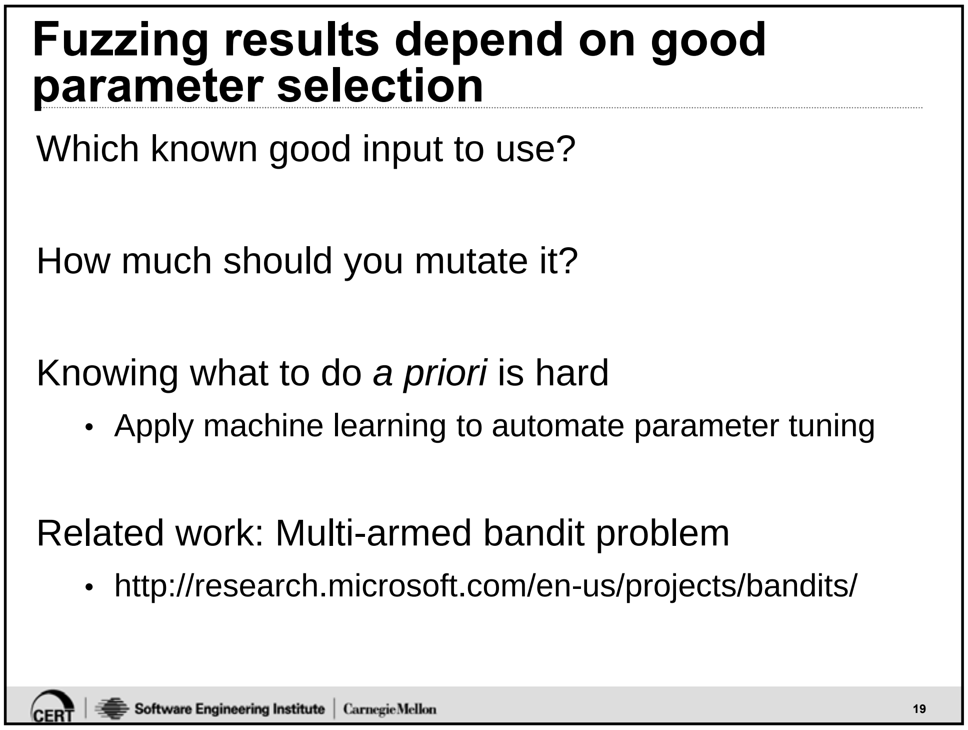
- *Iteration Rate* = Iterations per core per unit time
  - Has more to do with the program you're testing than fuzzing approach
    Unit: iterations / core-hour
- *Crash Rate = Crash Density x Iteration Rate* Unit: crashes / core-hour
- *Code Coverage* Unit: Varies…functions, basic blocks?
  - Can be slow to collect
  - Remember: code coverage space != input space

CERT | Software Engineering Institute | Carnegie Mellon 17

Crash Density Metric Applied



CERT | Software Engineering Institute | Carnegie Mellon 18



Fuzzing results depend on good parameter selection

Which known good input to use?

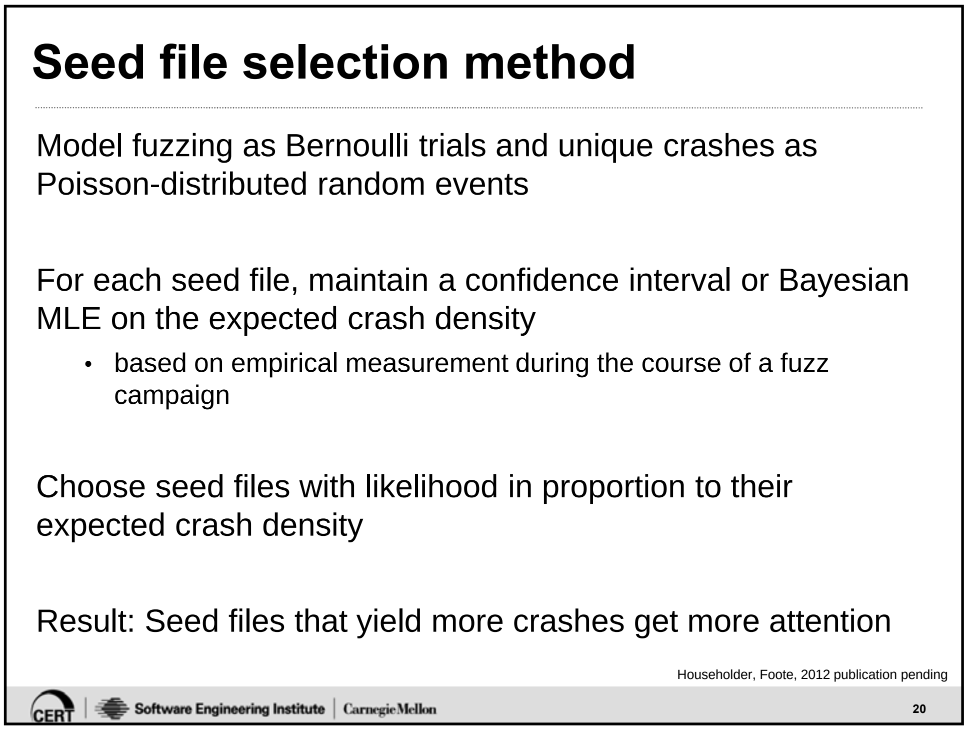
How much should you mutate it?

Knowing what to do *a priori* is hard

- Apply machine learning to automate parameter tuning

Related work: Multi-armed bandit problem

- http://research.microsoft.com/en-us/projects/bandits/



Seed file selection method

Model fuzzing as Bernoulli trials and unique crashes as Poisson-distributed random events

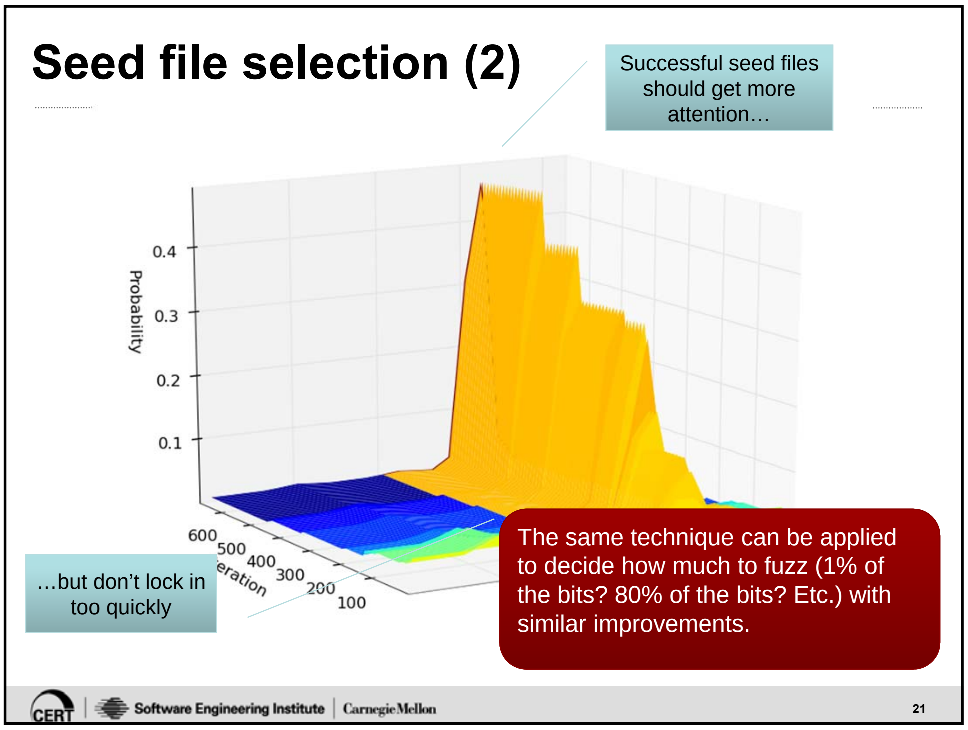
For each seed file, maintain a confidence interval or Bayesian MLE on the expected crash density

- based on empirical measurement during the course of a fuzz campaign

Choose seed files with likelihood in proportion to their expected crash density

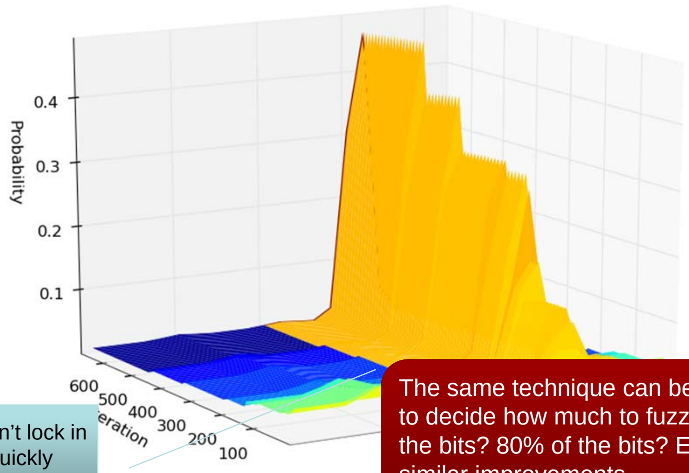
Result: Seed files that yield more crashes get more attention

Householder, Foote, 2012 publication pending



Seed file selection (2)

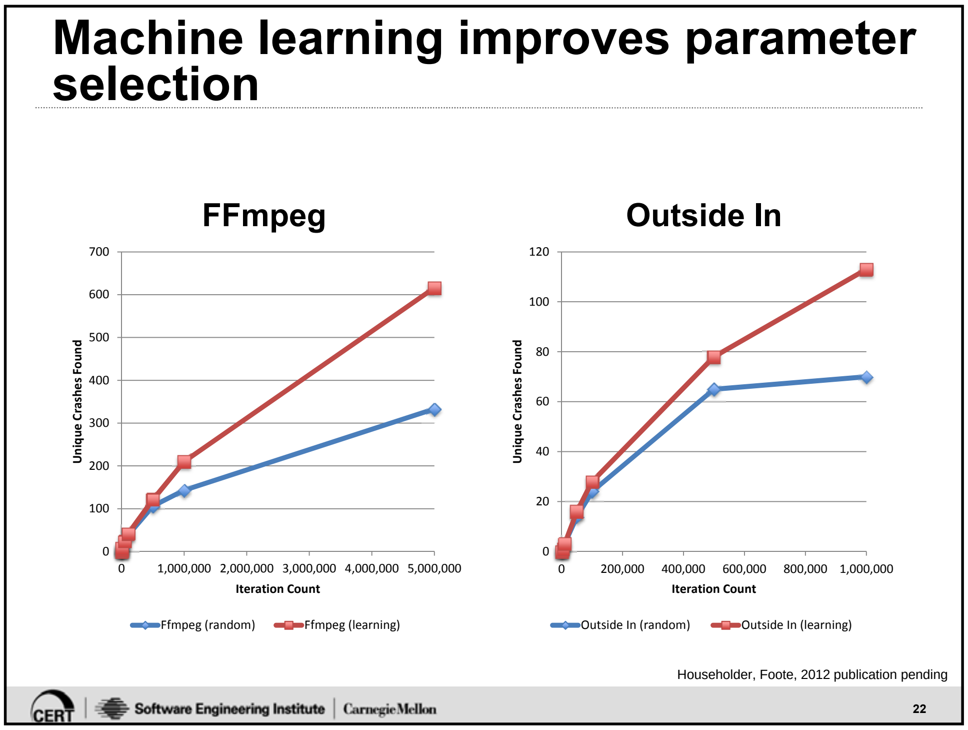
Successful seed files should get more attention…



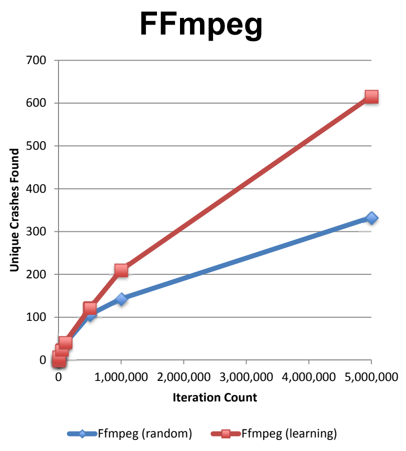
…but don't lock in too quickly

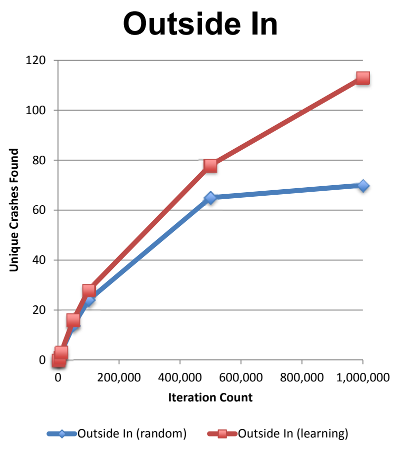
The same technique can be applied to decide how much to fuzz (1% of the bits? 80% of the bits? Etc.) with similar improvements.

21



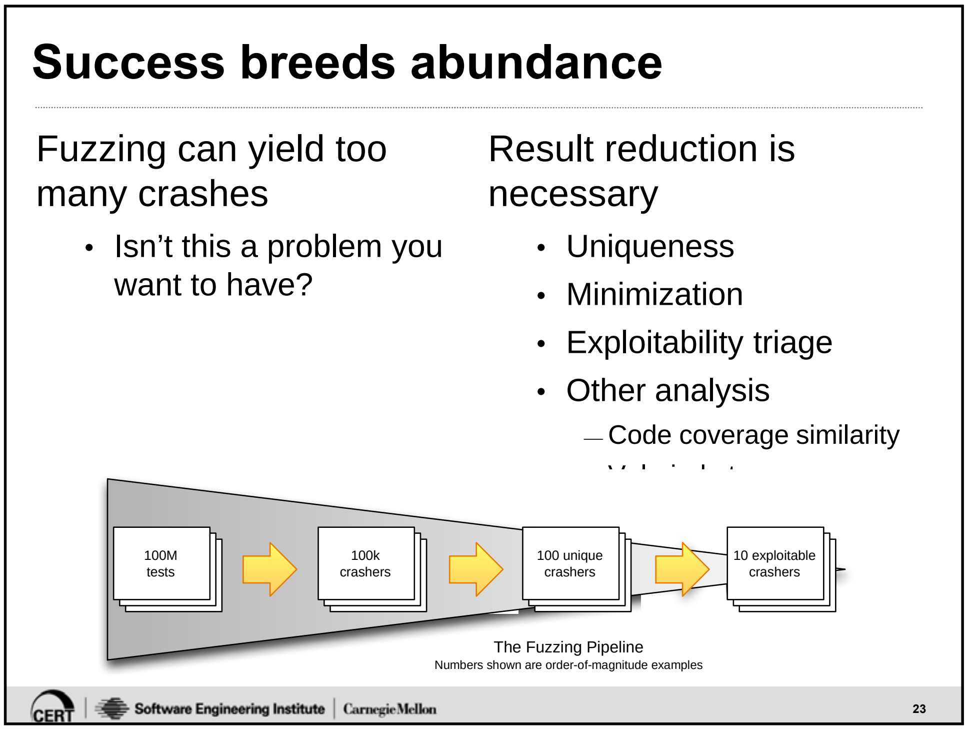
Machine learning improves parameter selection





Householder, Foote, 2012 publication pending

22



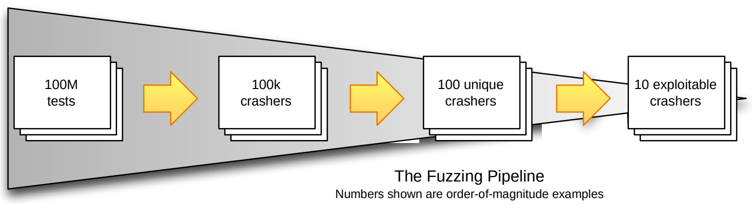
Success breeds abundance

Fuzzing can yield too many crashes

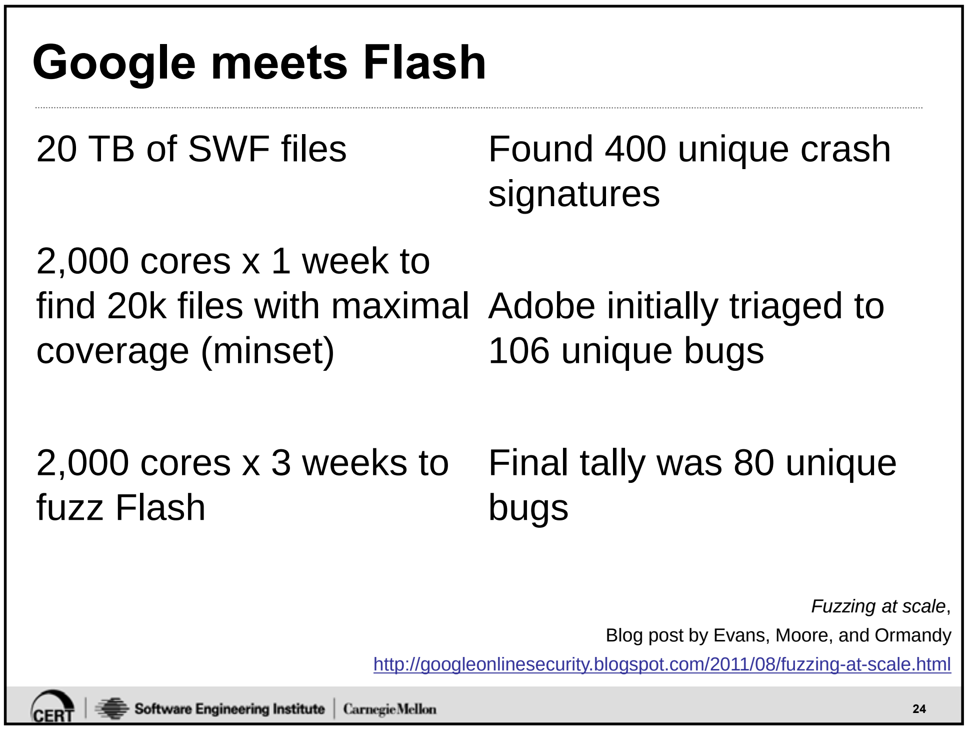
- Isn't this a problem you want to have?

Result reduction is necessary

- Uniqueness
- Minimization
- Exploitability triage
- Other analysis
  - Code coverage similarity



The Fuzzing Pipeline
Numbers shown are order-of-magnitude examples



Google meets Flash

20 TB of SWF files

Found 400 unique crash signatures

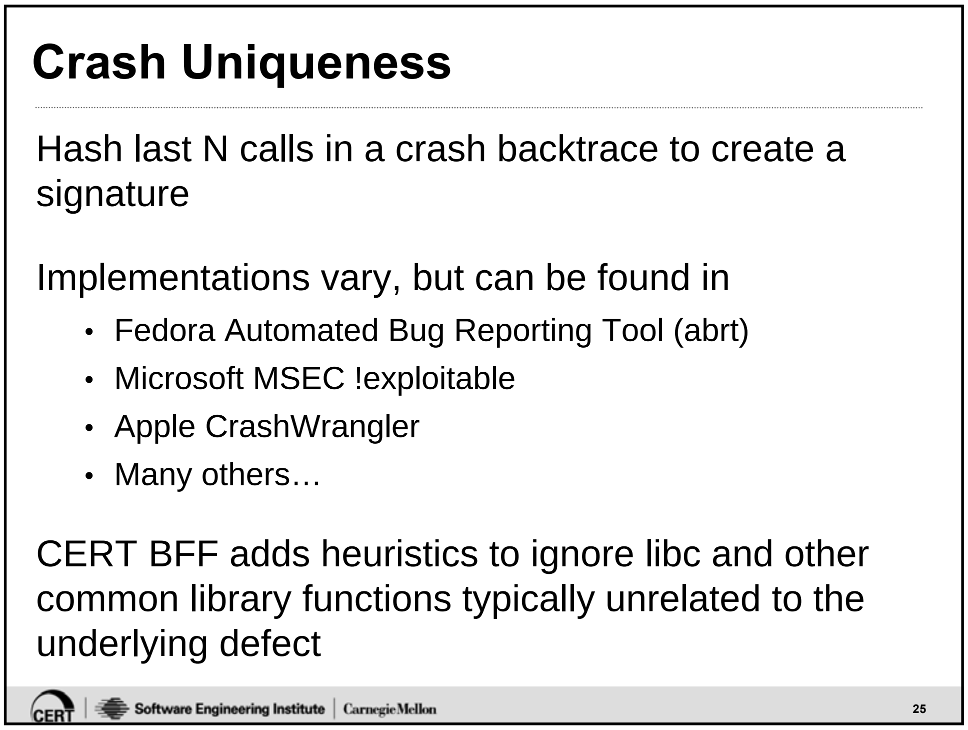
2,000 cores x 1 week to find 20k files with maximal coverage (minset)

Adobe initially triaged to 106 unique bugs

2,000 cores x 3 weeks to fuzz Flash

Final tally was 80 unique bugs

Fuzzing at scale,
Blog post by Evans, Moore, and Ormandy
http://googleonlinesecurity.blogspot.com/2011/08/fuzzing-at-scale.html



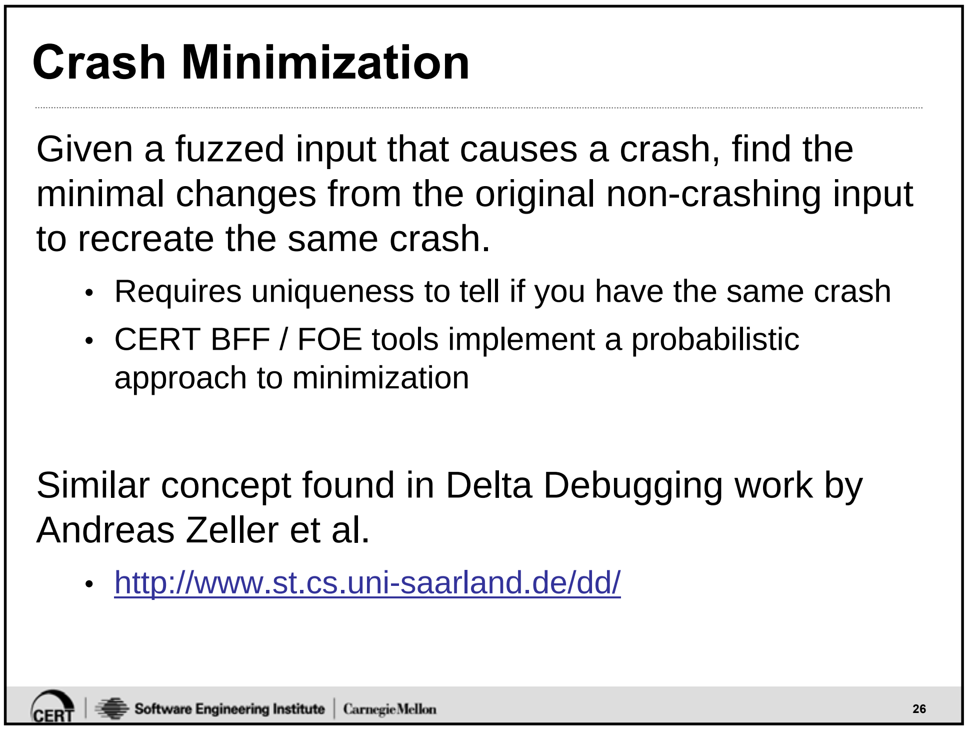
Crash Uniqueness

Hash last N calls in a crash backtrace to create a signature

Implementations vary, but can be found in

- Fedora Automated Bug Reporting Tool (abrt)
- Microsoft MSEC !exploitable
- Apple CrashWrangler
- Many others…

CERT BFF adds heuristics to ignore libc and other common library functions typically unrelated to the underlying defect



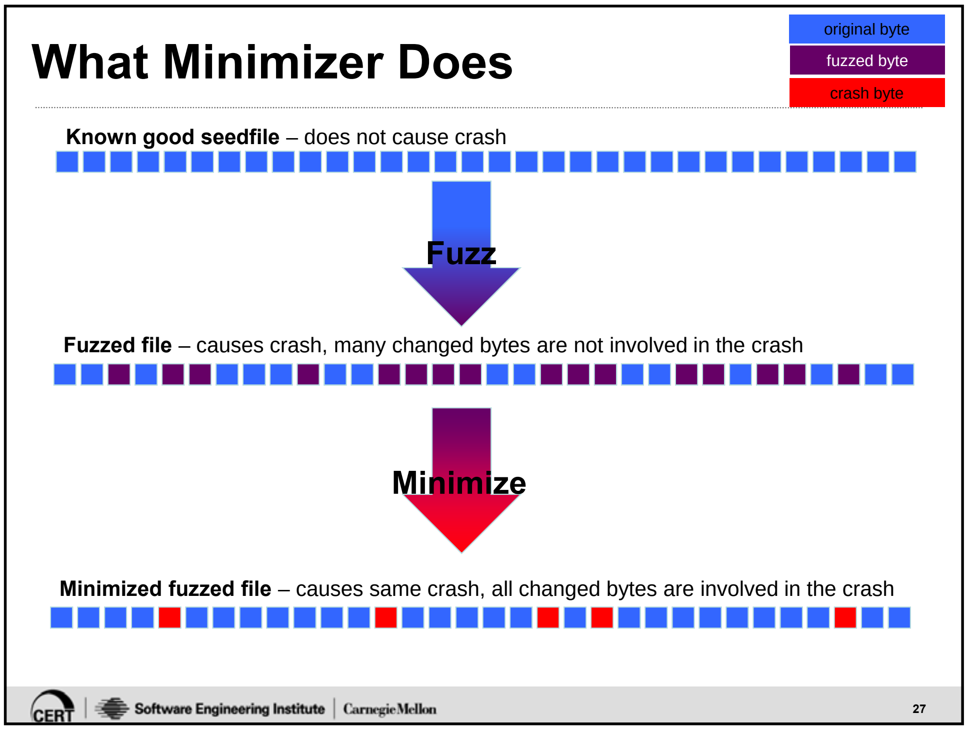
Crash Minimization

Given a fuzzed input that causes a crash, find the minimal changes from the original non-crashing input to recreate the same crash.

- Requires uniqueness to tell if you have the same crash
- CERT BFF / FOE tools implement a probabilistic approach to minimization

Similar concept found in Delta Debugging work by Andreas Zeller et al.

- http://www.st.cs.uni-saarland.de/dd/



What Minimizer Does

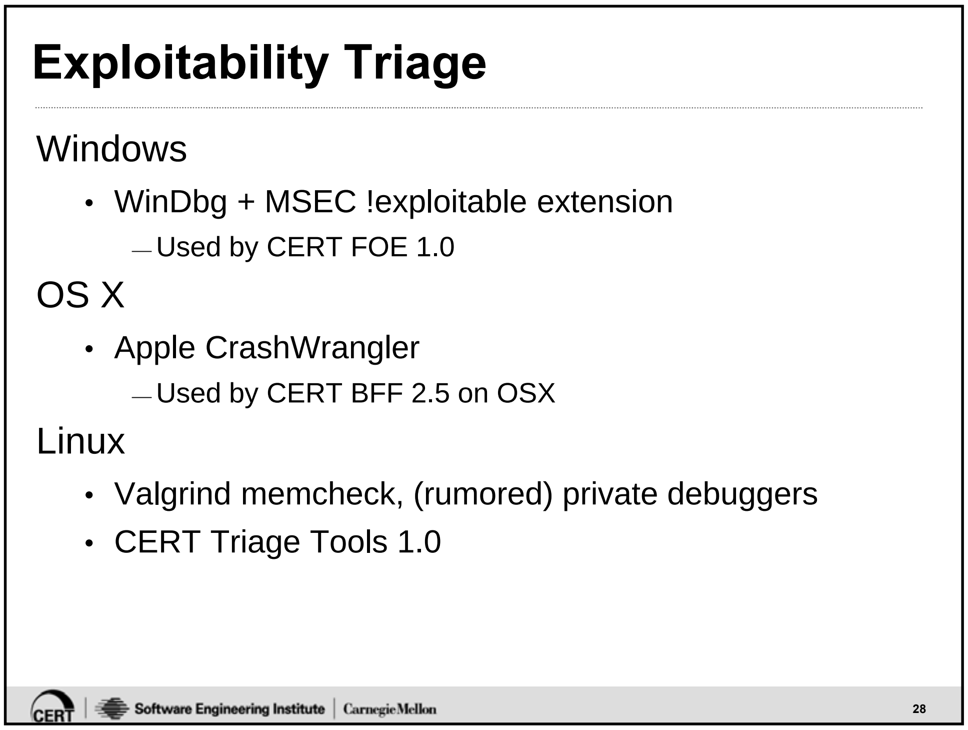
🟦 original byte
🟪 fuzzed byte
🟥 crash byte

Known good seedfile – does not cause crash

Fuzzed file – causes crash, many changed bytes are not involved in the crash

Minimized fuzzed file – causes same crash, all changed bytes are involved in the crash

Software Engineering Institute | Carnegie Mellon 27



Exploitability Triage

Windows

- WinDbg + MSEC !exploitable extension
  - Used by CERT FOE 1.0

OS X

- Apple CrashWrangler
  - Used by CERT BFF 2.5 on OSX

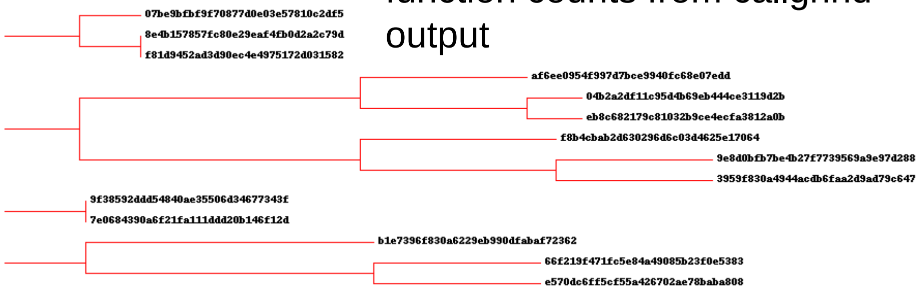
Linux

- Valgrind memcheck, (rumored) private debuggers
- CERT Triage Tools 1.0

Software Engineering Institute | Carnegie Mellon 28

Crash function call similarity

Use TF-IDF + cosine similarity on vectors of function counts from callgrind output


07be9bfbf9f70877d0e03e57810c2df5
8e4b157857fc80e29eaf4fb0d2a2c79d
f81d9452ad3d90ec4e4975172d031582
af6ee0954f997d7bce9940fc68e07edd
04b2a2df11c95d4b69eb444ce3119d2b
eb8c682179c81032b9ce4ecfa3812a0b
f8b4cbab2d630296d6c03d4625e17064
9e8d0bfb7be4b27f7739569a9e97d288
3959f830a4944acdb6faa2d9ad79c647
9f38592ddd54840ae35506d34677343f
7e0684390a6f21fa111ddd20b146f12d
b1e7396f830a6229eb990dfabaf72362
66f219f471fc5e84a49085b23f0e5383
e570dc6ff5cf55a426702ae78baba808

CERT | Software Engineering Institute | CarnegieMellon 29

Future work

The future is already here — it's just not very evenly distributed.

- Willam Gibson

Unique crashes != Unique bugs

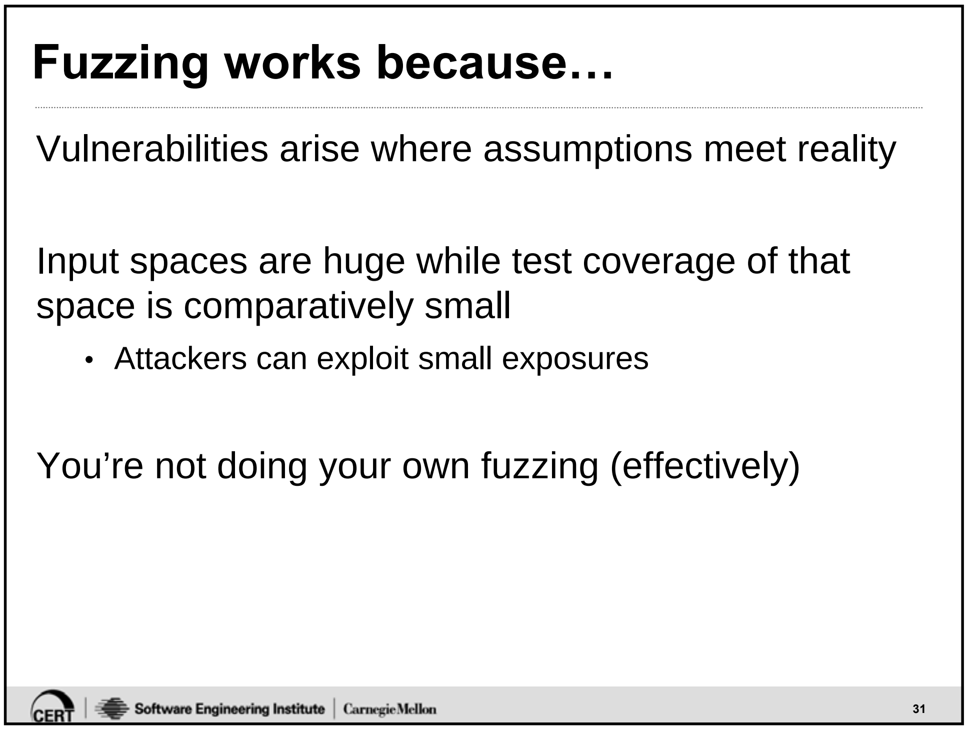
Binary instrumentation & coverage analysis

Symbolic execution & constraint solvers

Further application of machine learning & evolutionary computing techniques

Improve crash density estimators

CERT | Software Engineering Institute | CarnegieMellon 30



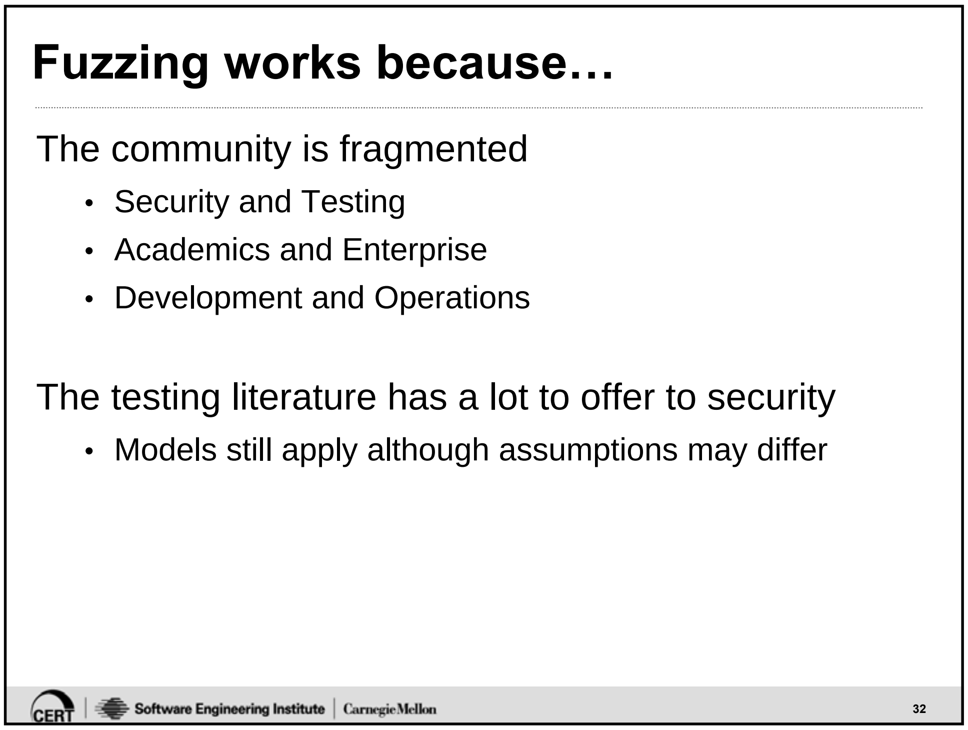
Fuzzing works because…

Vulnerabilities arise where assumptions meet reality

Input spaces are huge while test coverage of that space is comparatively small

- Attackers can exploit small exposures

You're not doing your own fuzzing (effectively)



Fuzzing works because…

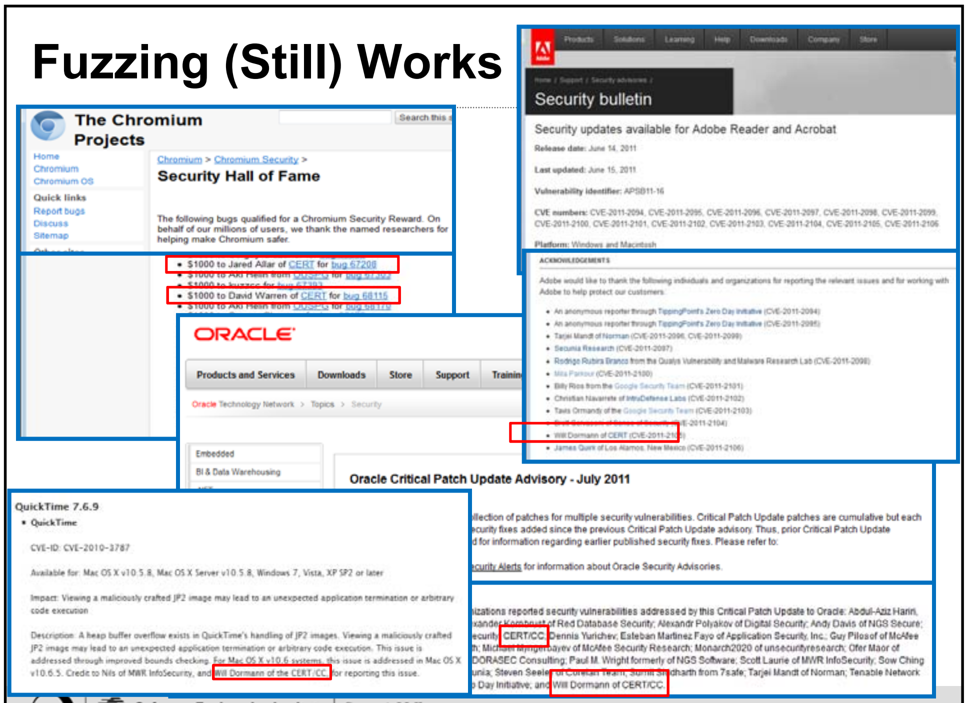
The community is fragmented

- Security and Testing
- Academics and Enterprise
- Development and Operations

The testing literature has a lot to offer to security

- Models still apply although assumptions may differ

Fuzzing (Still) Works



Chromium > Chromium Security >

Security Hall of Fame

The following bugs qualified for a Chromium Security Reward. On behalf of our millions of users, we thank the named researchers for helping make Chromium safer.

- $1000 to Jared Allar of CERT for bug 67208
- $1000 to Aki Helin from OUSPG for bug 67363
- $1000 to kuzzcc for bug 67393
- $1000 to David Warren of CERT for bug 68115
- $1000 to Aki Helin from OUSPG for bug 68170

Security updates available for Adobe Reader and Acrobat

Release date: June 14, 2011

Last updated: June 15, 2011

Vulnerability identifier: APSB11-16

CVE numbers: CVE-2011-2094, CVE-2011-2095, CVE-2011-2096, CVE-2011-2097, CVE-2011-2098, CVE-2011-2099, CVE-2011-2100, CVE-2011-2101, CVE-2011-2102, CVE-2011-2103, CVE-2011-2104, CVE-2011-2105, CVE-2011-2106

Platform: Windows and Macintosh

ACKNOWLEDGEMENTS

Adobe would like to thank the following individuals and organizations for reporting the relevant issues and for working with Adobe to help protect our customers:

- An anonymous reporter through TippingPoint's Zero Day Initiative (CVE-2011-2094)
- An anonymous reporter through TippingPoint's Zero Day Initiative (CVE-2011-2095)
- Tarjei Mandt of Norman (CVE-2011-2096, CVE-2011-2099)
- Secunia Research (CVE-2011-2097)
- Rodrigo Rubira Branco from the Qualys Vulnerability and Malware Research Lab (CVE-2011-2098)
- Mila Parkour (CVE-2011-2100)
- Billy Rios from the Google Security Team (CVE-2011-2101)
- Christian Navarrete of IntruDefense Labs (CVE-2011-2102)
- Tavis Ormandy of the Google Security Team (CVE-2011-2103)
- Brett Gervasoni of Sense of Security (CVE-2011-2104)
- Will Dormann of CERT (CVE-2011-2105)
- James Quirk of Los Alamos, New Mexico (CVE-2011-2106)

Oracle Critical Patch Update Advisory - July 2011

...llection of patches for multiple security vulnerabilities. Critical Patch Update patches are cumulative but each ...curity fixes added since the previous Critical Patch Update advisory. Thus, prior Critical Patch Update ...d for information regarding earlier published security fixes. Please refer to:

...curity Alerts for information about Oracle Security Advisories.

...izations reported security vulnerabilities addressed by this Critical Patch Update to Oracle: Abdul-Aziz Hariri, ...xander Kornbrust of Red Database Security; Alexandr Polyakov of Digital Security; Andy Davis of NGS Secure; ...ecurity; CERT/CC; Dennis Yurichev; Esteban Martinez Fayo of Application Security, Inc.; Guy Pilosof of McAfee ...h; Michael Myngerbayev of McAfee Security Research; Monarch2020 of unsecurityresearch; Ofer Maor of ...DORASEC Consulting; Paul M. Wright formerly of NGS Software; Scott Laurie of MWR InfoSecurity; Sow Ching ...unia; Steven Seeley of Corelan Team; Sumit Siddharth from 7safe; Tarjei Mandt of Norman; Tenable Network ...Day Initiative; and Will Dormann of CERT/CC.

QuickTime 7.6.9

- QuickTime

CVE-ID: CVE-2010-3787

Available for: Mac OS X v10.5.8, Mac OS X Server v10.5.8, Windows 7, Vista, XP SP2 or later

Impact: Viewing a maliciously crafted JP2 image may lead to an unexpected application termination or arbitrary code execution

Description: A heap buffer overflow exists in QuickTime's handling of JP2 images. Viewing a maliciously crafted JP2 image may lead to an unexpected application termination or arbitrary code execution. This issue is addressed through improved bounds checking. For Mac OS X v10.6 systems, this issue is addressed in Mac OS X v10.6.5. Credit to Nils of MWR InfoSecurity, and Will Dormann of the CERT/CC for reporting this issue.

For More Information

Visit CERT® web sites:

http://www.cert.org/vuls/discovery/

http://www.cert.org/blogs/certcc/

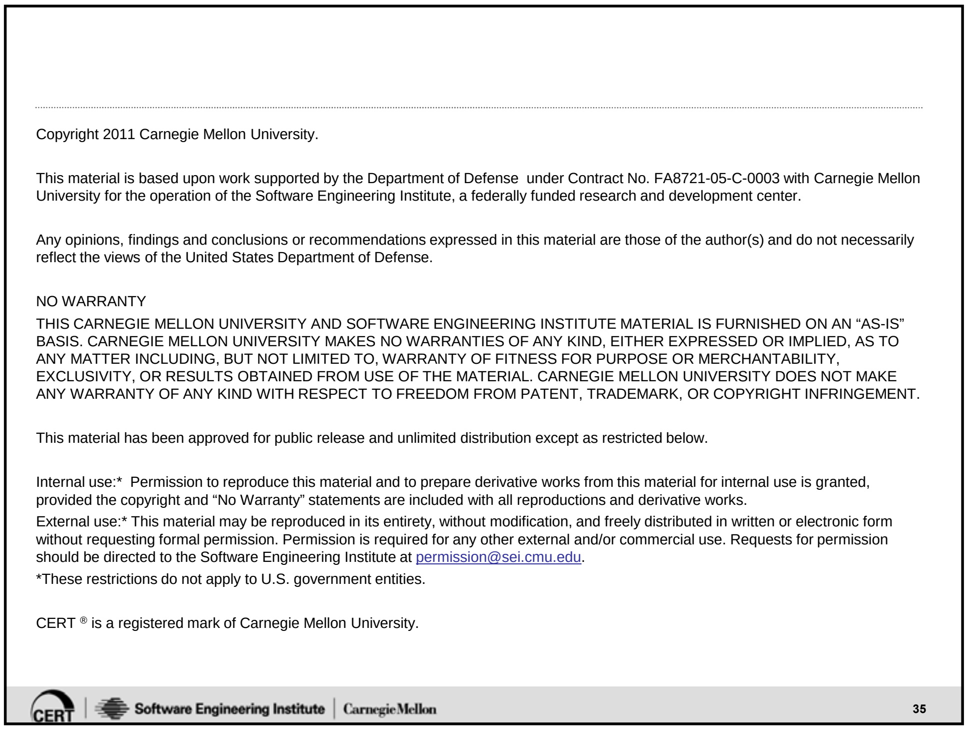
http://www.kb.cert.org/vuls

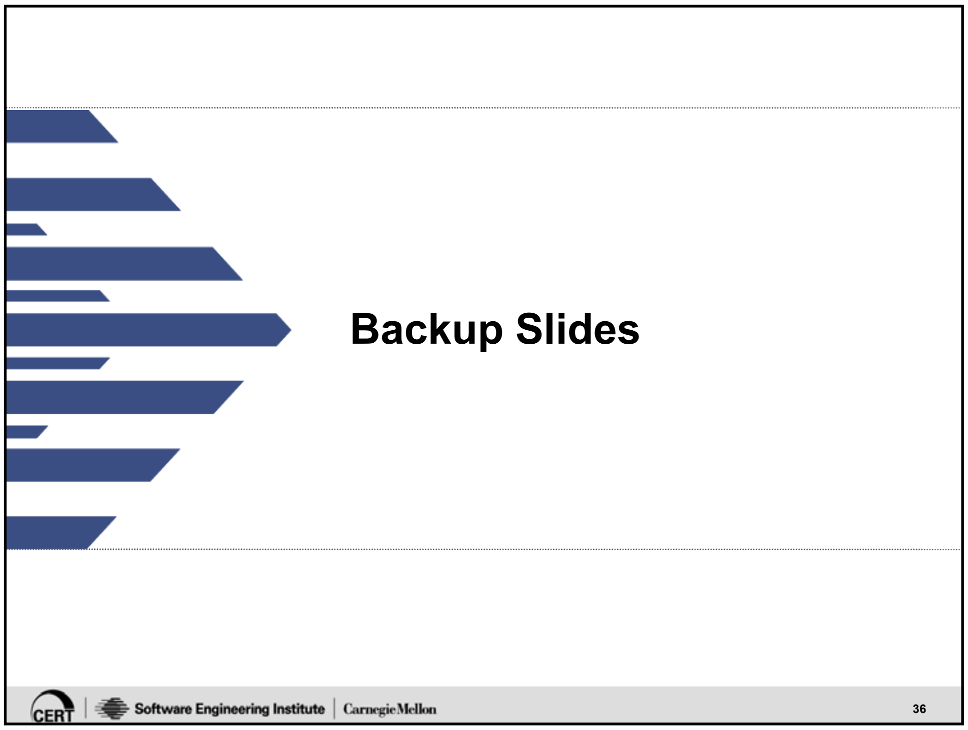
Contact Presenter

Allen D. Householder
adh@cert.org
(412) 268-5651

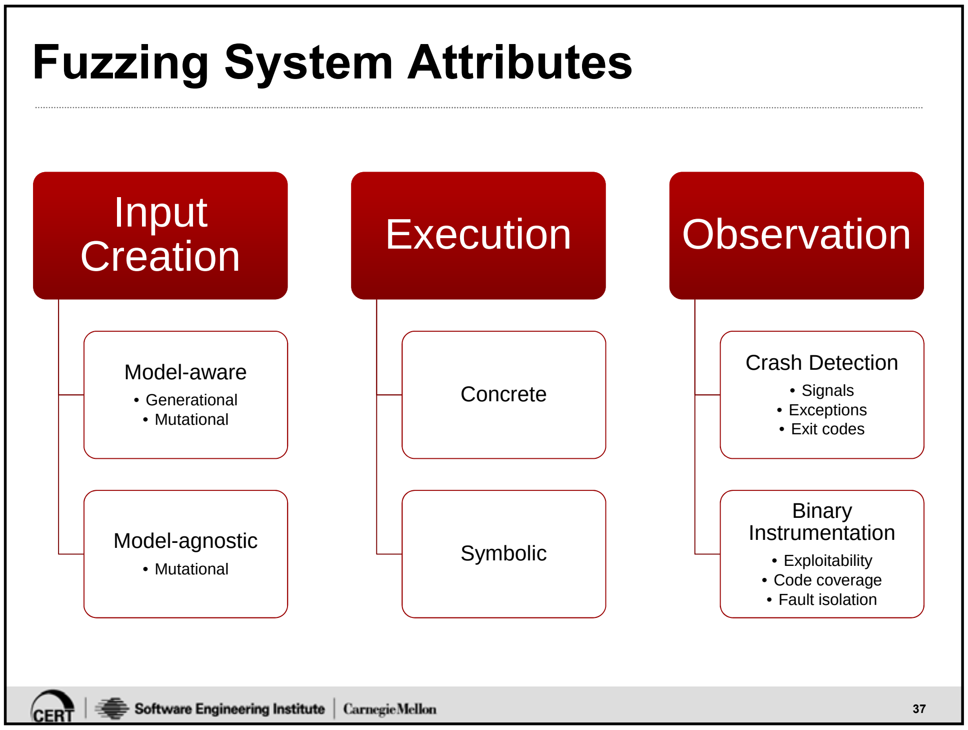
Contact CERT:

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh PA 15213-3890





Backup Slides



Fuzzing System Attributes

Model-based Testing: The State of the Practice

Robert Binder

System Verification Associates, LLC

Abstract:

A recent survey of model-based testing (MBT) users indicates some interesting trends. MBT usage spans a wide range of application stacks, software processes, application domains and development organizations. This talk will present the findings of the study and offer some reflections on the state of the practice and its prospects.

The full report may be viewed at

Http://www.robertvbinder.com/docs/arts/MBT-User-Survey.pdf

Security testing: a key challenge for software engineering of web apps

Yves Le Traon

Abstract:

While important efforts are dedicated to system functional testing, very few work study how to specifically and systematically test security mechanisms. In this position paper, we plead for a systematic standardization of security testing benchmarks. We will illustrate the security testing issues with two categories of approaches, taken from our ongoing research. The first ones aim at assessing security mechanisms compliance with declared policies. Any security policy is strongly connected to system functionality: testing function includes exercising many security mechanisms. However, testing functionality does not intend at exercising all security mechanisms. We thus propose test selection criteria to produce tests from a security policy. Empirical results will be presented about access control policies and about Android apps permission checks.

The second ones concern the attack surface of web apps, with a particular focus on web browser sensitivity to XSS attacks. Indeed, one of the major threats against web applications is Cross-Site Scripting (XSS) that crosses several web components: web server, security components and finally the client's web browser. The final target is thus the client running a particular web browser. During this last decade, several competing web browsers (IE, Netscape, Chrome, Firefox) have been upgraded to add new features for the final users benefit. However, the improvement of web browsers is not related with systematic security regression testing. Beginning with an analysis of their current exposure degree to XSS, we extend the empirical study to a decade of most popular web browser versions. The results reveal a chaotic behavior in the evolution of most web browsers attack surface over time. This particularly shows an urgent need for regression testing strategies to ensure that security is not sacrificed when a new version is delivered.

In both cases, security must become a specific target for testing in order to get a satisfying level of confidence in security mechanisms.