Quantitative Cyber-Security

Colorado State University
Yashwant K Malaiya

CS559
L16: CVSS & Testing



CSU Cybersecurity Center
Computer Science Dept

CVSS Scoring System

- How important is a specific vulnerability?
  - Essentially a risk measure*
  - Vulnerabilities with highest scores need addressing quickly. Those with lowest scores are low priority.
- CVSS v1: *National Infrastructure Advisory Council* (NIAC), 2005
- CVSS v2: *Forum of Incident Response and Security Teams* (FIRST)
  - 2007
  - Still common
- CVSS V3: 2015
- CVSS V4: Nov 2023 coming into use

* As we will see

2

CVSS Metrics and Scores

De facto standard for assessing severity of software vulnerabilities

- Objective: prioritize effort to address vulnerabilities
- Metrics: Components by levels, each translated into a numerical metric (detailed tables soon)
- Scores: Computed score using a set of metrics as given by formulas
  - Formulas based on committee judgement, not derived or proven
- Three metric groups and associated scores;
  - Base (mandatory): intrinsic to the vulnerability
  - Thread ~~Temporal~~: time-dependent variation in risk (renamed Threat metrics in V4)
  - Environmental: risk component dependent on the organization's environment
  - Supplemental (V4)

* As we will see

CVSS Base Metrics & Score: Overview

CVSS Base Score = f(Exploitability sub-score, Impact sub-score,)

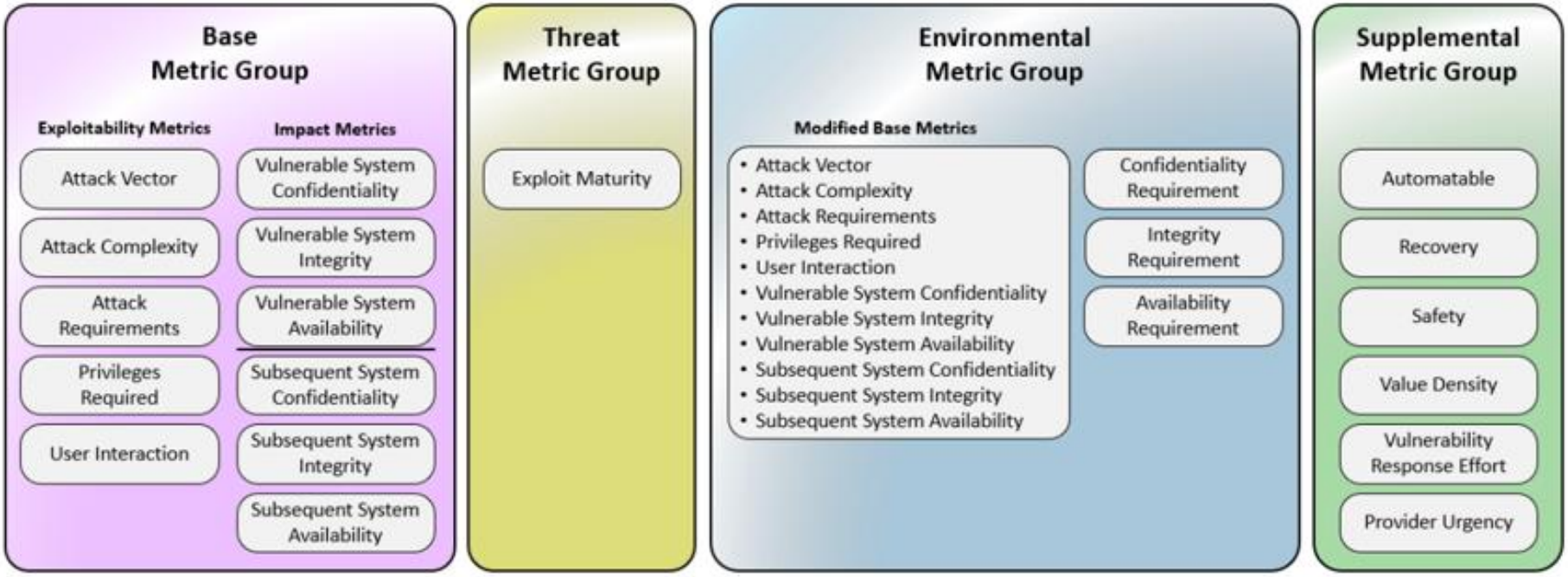
- This is the score used to prioritize vulnerabilities*.
- Ranges from 0 to 10.

- Exploitability metrics: how easy is the vulnerability to exploit
  - Exploitability sub-score = f(Attack Vector, Attack Complexity, Attack Requirements, Privileges Required, User Interaction~~, Scope~~)
- Impact sub-score: what is the extent of the impact.
  - Impact sub-score = f(Confidentiality, Integrity,, Availability, subsequent integrity and availability)
- Intrinsic value of a raw metric ranges from 0 to 1.
- Our observation: exploitability sub-score attempts to measure *likelihood*$_i$, Impact sub-score attempts to measure *impact*$_i$ due to a *vulnerability i.*

* Details to follow

4

CVSS V4.0 Metrics



CVSS Metric Groups

FIRST: [Forum of Incident Response and Security Teams]

883 Teams in nine countries

Colorado State University

CVSS Base Scores: Ratings

All recognized vulnerabilities are assigned Base Scores*. The complete data for recognized vulnerabilities is available at the National Vulnerability Database.

Ranking according to CVSS V3/V4.

Rating	CVSS Score	Response time
None	0.0	Next major release accept risk
Low	0.1 - 3.9	Next major release accept risk
Medium	4.0 - 6.9	90-day SLA maintenance window
High	7.0 - 8.9	30-day SLA planned sprint
Critical	9.0 - 10.0	24-72 hours Emergency

* Has this measure been validated? Interesting question.

6

CVSS Metrics: Data bases

NVD (National Vulnerability database): U.S. government repository of standards based vulnerability management data. represented using the Security Content Automation Protocol (SCAP).

- sponsored by the Department of Homeland Security (DHS), NCCIC and US-CERT.

Documentation/Data hosted at NVD

Example: MySQL Stored SQL Injection (CVE-2013-0375)

- Oracle MySQL 5.1.66 and earlier, and 5.1.28 and earlier
- CVSS v3.0 Base Score: 5.4 medium
  - Vector: CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:C/C:L/I:L/A:N
- CVSS v4.0 Base Score: not yet assessed

CVSS v3.1 Severity and Metrics:
Base Score: 5.4 MEDIUM
Vector: AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:L/A:N
Impact Score: 2.5
Exploitability Score: 2.8

Attack Vector (AV): Network
Attack Complexity (AC): Low
Privileges Required (PR): Low
User Interaction (UI): None
Scope (S): Unchanged
Confidentiality (C): Low
Integrity (I): Low
Availability (A): None

* Has this measure been validated? Interesting questions.

Examining CVSS V.3

- CVSS V3.1 is still widely used, CVSS v4.0 is only used by a small fraction of the industry.
  - Database and Metric Incompatibility: CVSS v4.0 completely restructures the scoring string (removing the "Scope" metric and adding subsequent impacts and supplemental metrics).
  - Hard to compute v4.0 scores. Only 25.9% of published CVEs include a CVSS v4.0 score!
  - Formulas have been replaced by a complex procedure:
    - The 15 million possible metric combinations are grouped into 270 equivalence sets called *MacroVectors*. Experts analyzed these sets to establish baseline severity boundaries and scores mapped out from 0.0 to 10.0.
- We thus examine CVSS V3 is detail.

CVSS V.3 Temporal Metrics 1

Temporal metrics measure the current state: exploit availability, patches or workarounds, or the confidence in the description of the vulnerability.

Note: An exploit is a piece of code that exploits the vulnerability. Available exploits are documented at the Exploit databse.

My Comment: Good idea, but the concept needs some work.

Exploit Code Maturity (E)	**context by which vulnerability exploitation is possible**	
Metric	**Represents**	**Value V3.0**
Not Defined (X)	No information available	1
High (H)	widely available exploit code works in every situation	1
Functional (F)	exploit code works in most situations	0.97
Proof-of-Concept (P)	Proof-of-concept exploit available but an attack demonstration is not practical for most systems	0.94
Unproven (U)	No exploit code is available, or an exploit is theoretical	0.91 Lowest risk

Temporal Metrics 2

Remediation Level (RL): patched yet?

Remediation Level (RL)	context by which vulnerability exploitation is possible	
Not Defined (X)	No information available	1
Unavailable (U)	no solution available	1
Workaround (W)	an unofficial, non-vendor solution available	0.97
Temporary Fix (T)	official but temporary fix available	0.96
Official Fix (O)	official patch, or an upgrade is available	0.95

Report Confidence:

Report Confidence (RC)	the degree of confidence in the existence of the vulnerability	
Not Defined (x)	No information available	1
Confirmed (C)	Detailed reports exist	1
Reasonable (R)	researchers do not have full confidence in the root cause	0.96
Unknown (U)	questionnable report	0.92

10

Temporal score and comments

Temporal score

$$= Round\ up(Base\ Score \times Exploit\ Code\ Maturity \times Remediation\ Level \times Report\ Confidence)$$

Comments*: The significance of the presence of an Exploit and a Patch is obvious. It is not clear by all the values are close to 1.0, i.e. the levels did not appear to matter for the FIRST people.*

Note that

- Most vulnerabilities are declared along with patch releases. However users often do not apply the patches immediately.
- Known exploits often appear quickly, if the vulnerability is significant.
  - *However we have found a lack of correlation between vulnerability severity and delay in exploit appearance. It may be explained that a high severity vulnerability may already have disappeared by in new release when it is disclosed, making exploit development largely useless.*

V4: Terms this Threat, retains only Exploit maturity metric in it)

11

Environmental Metrics

The environmental score

= If (Modified Impact Sub score =< 0) 0

else,

Modified Scope Unchanged

= $Round\ up(Minimum\ [(M.Impact+M.Exploitability)\times Exploit\ Code\ Maturity\times Remediation\ Level\times Report\ Confidence,10])$

Modified Scope Changed

= $Round\ up(\ Round\ up[Minimum[1.08\times(M.Impact+M.Exploitability)],10]\times Exploit\ Code\ Maturity\times Remediation\ Level\times Report\ Confidence))$

V4: drops Scope. Adds additional metrics

12

CVSS Base Score vs Vulnerability Rewards Programs

- We examined 1559 vulnerabilities of Mozilla Firefox and Google Chrome browsers for which records were available.
- Looked at Mozilla and Google vulnerability reward programs (VRPs) records for those vulnerabilities.

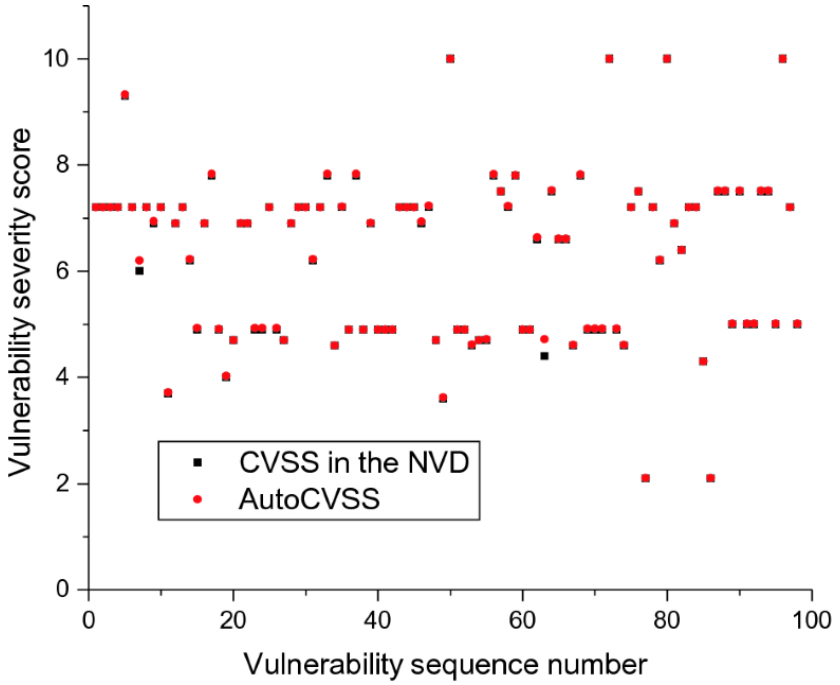
Firefox		
Vulnerabilities	Rewarded	Not rewarded
547	225	322
VRP severity	Rewarded	Not rewarded
Critical & High	210	202
Medium	15	89
Low	0	31

A. Younis, Y. Malaiya and I. Ray, "Evaluating CVSS Base Score Using Vulnerability Rewards Programs", Proc. 31th Int. Information Security and Privacy Conference, IFIP SEC, Ghent, Belgium, 2016, pp. 62-75

Colorado State University

AutoCVSS?

- Zou et al. 2019: 98 vulnerabilities from Linux kernel, FTP service, and Apache service with their exploits from exploit-db.
- CVSS relies on human experts to determine metric values during the process of vulnerability severity assessment. They have attempted to automate the process.
- Result is that only two vulnerability severity scores assessed by AutoCVSS are obviously different from those in the NVD for CVSS v2.



D. Zou, J. Yang, Z. Li, X. Ma, , AutoCVSS: An Approach for Automatic Assessment of Vulnerability Severity Based on Attack Process, Int. Conf. on Green, Pervasive, and Cloud Computing, April 2019

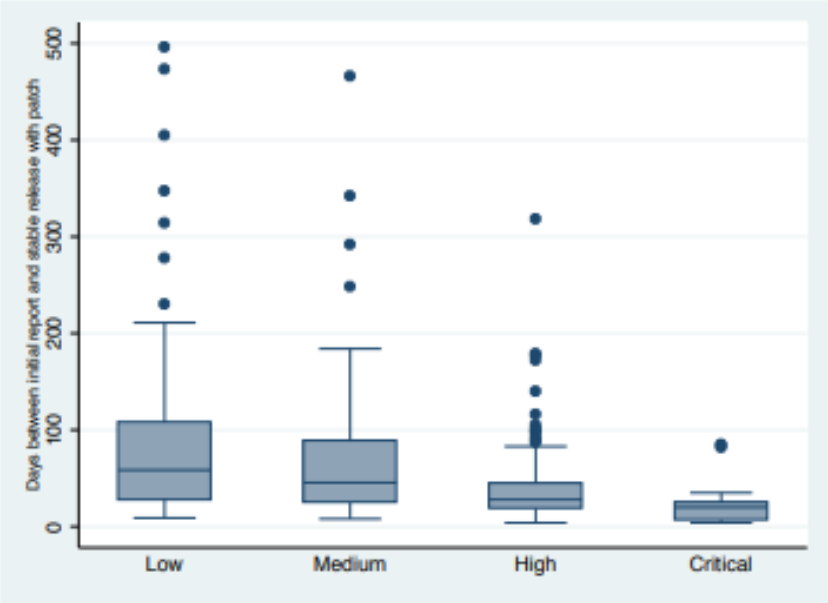
Colorado State University

VRP Cost effectiveness

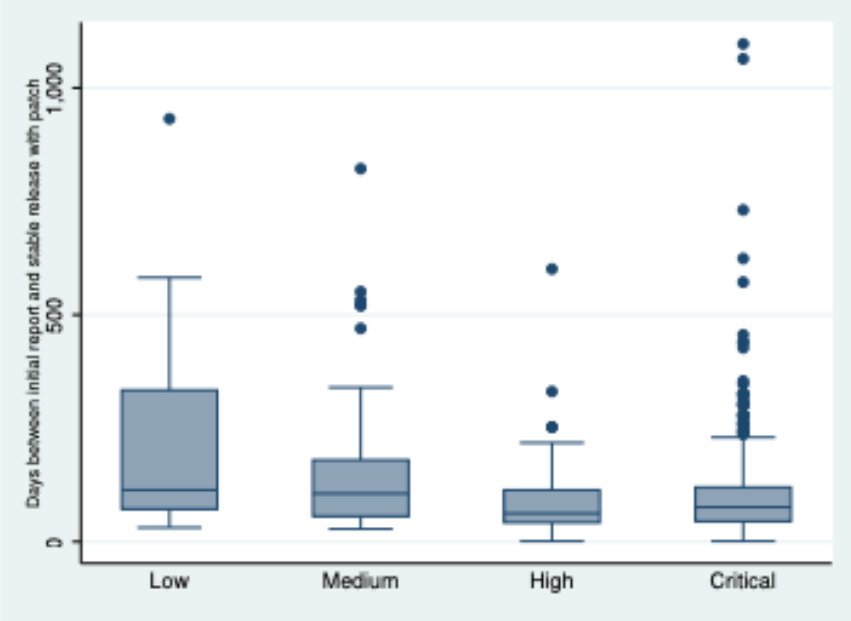
- Hypothesis: A VRP can be a cost-effective mechanism for finding security vulnerabilities.
  - Period studied: 7/09-1/13
  - Chrome's VRP has cost $485 per day on average, and that of Firefox has cost $658 per day.
  - Average North American developer on a browser security team (i.e., that of Chrome or Firefox) would cost around $500 per day (assuming a $100,000 salary with a 50% overhead).
- Hypothesis: Contributing to a single VRP is, in general, not a viable full-time job, though contributing to multiple VRPs may be, especially for unusually successful vulnerability researchers.
- Hypothesis: Successful independent security researchers bubble to the top, where a full-time job awaits them

M. Finifter, D. Akhawe, D. Wagner, An empirical study of vulnerability rewards programs. In USENIX Security Symposium 2013 (2013), pp. 273-288

Time to patch



(a) Chrome



(b) Firefox

M. Finifter, D. Akhawe, D. Wagner, An empirical study of vulnerability rewards programs. In USENIX Security Symposium 2013 (2013), pp. 273-288

CVSS Calculators & Vector Strings

Calculators:

- **Common Vulnerability Scoring System Version 3.1 Calculator**
  - **Example: GNU Bourne-Again Shell (Bash) 'Shellshock' Vulnerability (CVE-2014-6271) Vulnerability**
  - CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
    - Attack Vector: Network, Attack complexity: Low etc …
  - **Base score:** 9.8 (Critical)
- **Common Vulnerability Scoring System Version 4.0 Calculator**

CVSS: How useful it is?

- What if they had multiplied exploitability and impact sub-scores instead of adding?
- Correlation among
  - CVSS Exploitability
  - Microsoft Exploitability metric
  - Presence of actual exploits
- Time to discovery?
- Reward program?
- Can metric/score determination be automated?

Course notes

- HW2a due 10/16/2026
- HW2b Should be available soon. Involves SRGMs.
  - Due Nov 4..
- Midterm Nov 3 (4 for S801)

Quantitative Security

Colorado State University
Yashwant K Malaiya

CS 559
Testing



CSU CyberCenter
***Course Funding Program* – 2019**

Testing for bugs



21

Faults

- Faults cause a system to respond in a way different from expected.
- Faults can be associated with bugs in the system/software structure or functionality.
  - Structure: viewed as an interconnection of components like statements, blocks, functions, modules.
  - Functionality: Described by the input/output/state behavior, described externally. Black-box view.
  - Both structure and functionality can be described at a higher level and a lower (finer) level.
    - Example: a file > classes > methods etc. > statements

22

Testing

- Testing and debugging is an essential part of software development and maintenance. 15-75% cost
  - Static analysis: code inspection
  - Dynamic: involves execution
- Defects cause functionality/reliability and security problem.
- Vulnerabilities are a subset of the defects (1-5%)
  - If exploited, allow violation of security related assumptions.
  - Vulnerability discovery can involve testing with
    - Random tests (Fuzzing)
    - Generated tests based on security requirements
- The following discussion is general for all defects.

Testing

- We assume that tests are applied at the inputs and the response is observed at the outputs of the unit-under-test.
- A test detects the presence of a fault(s), if the output is different from the expected output.
- Two test approaches:
  - Functional (or Black-box): uses only the functional description of the unit, not its structure to obtain tests. Often random ("fuzzing")
  - Structural testing: uses the structural information to generate tests. Requires more effort but can be more thorough.
  - Combined

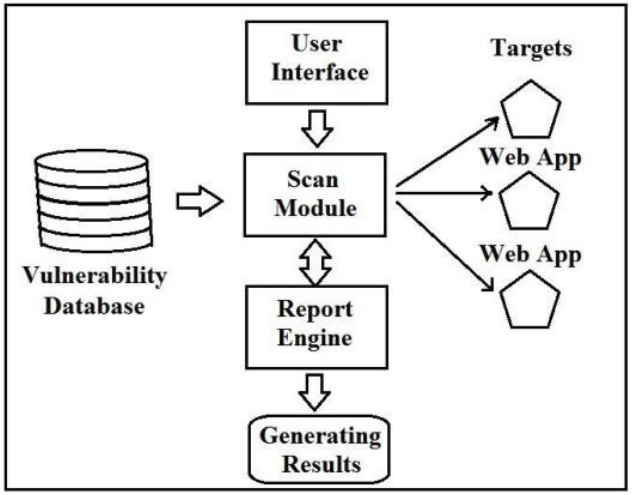
24

Random vs Directed Testing

- Random testing: Termed Black-box, fuzzing when used for vulnerabilities
  - Random testing is a form of functional testing. In random testing, each test is chosen such that it does not depend on past tests.
  - In actual practice, the "random" tests are generated using Pseudo-random algorithms that approximate randomness.
  - As we will discuss later, random testing can be effective for moderate degree of testing, but not for thorough testing.
- Directed Testing: Specifically trying to cover the input space or structural components (statements, branches, modules etc.)

Testing objectives

- Ordinary faults:
  - Fault detection: Apply a test input. Is the output what is expected? Triggering fault and propagating error.
  - Fault location: where is the fault?
  - Fixing: what will fix the fault? (debugging)
- Known vulnerabilities
  - use vulnerability reports/database for the product (Vulnerability scanners)
- Discovering unknown Vulnerabilities
  - Apply a slightly unexpected input. Does a program crash or hang?
  - If it does, examine it to see if it leads to a vulnerability.
  - Can the vulnerability be exploited?



Partitioning

- Software can be partitioned to ensure that the software is thoroughly exercised during testing
- It is necessary to partition it to identify tests that would be effective for detecting the defects in different sections of the code.
- For testing purposes, a program may be partitioned either functionally or structurally.
- *Functional partitioning* refers to partitioning the input space of a program.
  - For example, if a program performs five separate operations, its input space can be partitioned into five partitions.
  - Functional partitioning only requires the knowledge of the functional description of the program, the actual implementation of the code is not required.
- *Structural partitioning* requires the knowledge of the structure at the code level.
  - If a software is composed of ten modules (which may be classes, functions or other types of units), it can be thought of as having ten partitions

Y. K. Malaiya, "Assessing Software Reliability Enhancement Achievable through Testing", Recent Advancements in Software Reliability Assurance 2019, pp. 107-138

Sub-Partitioning

- A partition of either type can be subdivided into lower-level partitions, which may themselves be further partitionable at a lower level if higher resolution is needed (Elbaum 2001).
- Let us assume that a partition p_i can be subdivided into sub-partitions $\{p_{i1}, p_{i2} \ldots p_{in}\}$.
  - Random testing within the partition p_i will randomly select from $\{p_{i1}, p_{i2} \ldots p_{in}\}$. It is possible that some of them will get selected more often in a non-optimal manner.
  - Code within a sub-partition may be correlated relative to the probability of exercising some faults. Thus, the effectiveness of testing may be diluted if the same sub-partition frequently gets chosen.
  - Sub-partitioning has a practical disadvantage when the *operational profile* is constructed, it will require estimating the operational probabilities of the associated sub-partitions.

Test coverage

- A single test typically covers (i.e., tests for related faults) several sub-partitions (elements such as functions, branches, statements)
- The coverage obtained by a test-set (test suite) can be obtained using coverage tools.
- The test coverage achieved by a test-set is given by ratio:

$$coverage = \frac{number\ of\ elements\ covered}{total\ number\ of\ elements}$$

- 100% coverage may be very hard or impossible to achieve in most practical cases.

29

Coverage Tools

- There are several code coverage tools: Jcov, Gcov etc. for Java, C/C++ etc.
  - Compilation using the tool, instruments the compiled code to collect metrics covered.
  - Coverage metrics:
    - Statements/Blocks
    - Branches/Edges
    - Paths
    - Methods/Functions
    - Data-flow coverage metrics
  - Subsumption hierarchy
    - Complete Path coverage => 100% Branch coverage
    - Complete Branch coverage => 100% Statement coverage

Assumptions:
- A fault is associated with one or more elements.
- Exercising the element may trigger the fault to create an error
- Complete coverage does not guarantee finding all the faults.

Input mix: Test Profile

- The inputs to a system can represent different types of operations. The input mix called "Profile" can impact effectiveness of testing.
- Example:
  - elements e_1, e_2, …e_i, …e_n exercised with probabilities p_1, p_2, …p_i, …p_n
  - Profile then is $\{(e_i, p_i)\}$ for all elements
- For example, a Search program can be tested for text data, numerical data, data already sorted etc. If most testing is done using numerical data, more bugs related to text data may remain unfound.

Input mix: Test Profile

- **The ideal Profile (input mix) will depend on the objective**
  - A. Find bugs fast? or
  - B. Estimate operational failure rate?

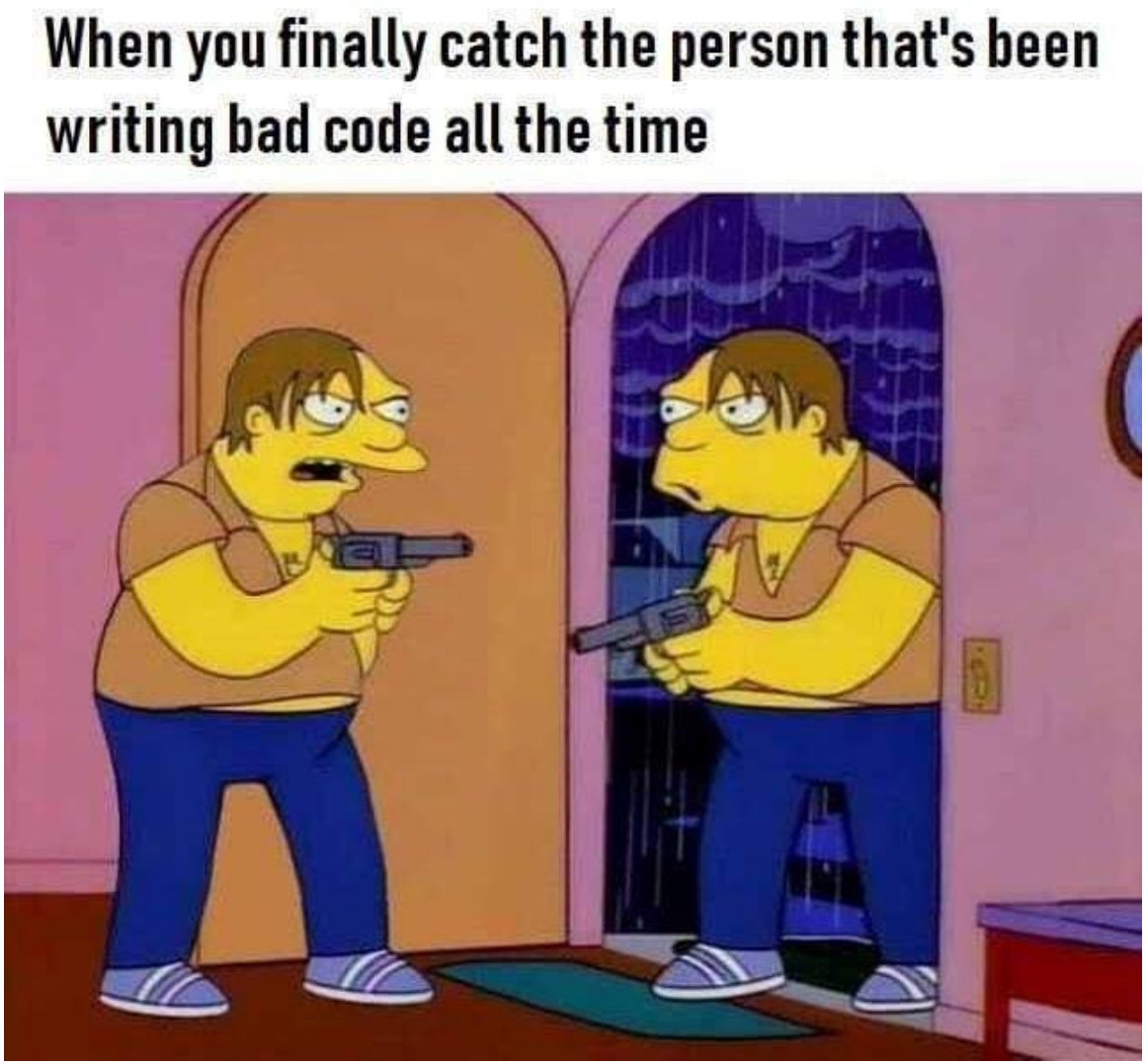
A. Best mix for **functional** bug finding (Li & Malaiya' 94)
  - Quick & limited testing: *Use **operational profile: how the inputs are encountered in actual operation.***
  - High reliability: *Probe input space evenly*
    - Operational profile will not execute rare and special cases, the main cause of failures in highly reliable systems.
  - Very high reliability: corner cases and rare combinations

B. For **security bugs**: corner cases and rare combinations
  - Vulnerability finders / exploiters look for these.

N. Li and Y.K. Malaiya, On Input Profile Selection for Software Testing, Proc. Int. Symp. Software Reliability Engineering, Nov. 1994, pp. 196-205.

H. Hecht, P. Crane, Rare conditions and their effect on software failures, Proc. Annual Reliability and Maintainability Symposium, 1994, pp. 334-337

Who is writing bad code?



Modeling Bug Finding Process

- The number of bugs found depend on the effort (measured by testing time) and directedness of testing.
- Directedness: looking for bugs
  - In elements not yet exercised enough
    - These will include corner cases
  - Where bugs of a specific type (specially vulnerabilities) are likely to be present.
    - Experience, expertise, intuition

Nature of faults: Detectability Profile

- All faults are not alike.
- There is no such thing as an *average* fault.
- As testing progresses, the remaining faults are the ones harder to find.

Detection Probability

- **Detection probability** of a fault: if there are N distinct possible input vectors, and if a fault is detected by k of them, then its *detection probability* is k/N.
- A fault with detection probability (dp) 1/N would be hardest to test, since it is tested by only one specific test and none other.
- A fault which is detected by almost all vectors, would have a detection probability close to 1 and will be found with minimal texting effort. It is a low hanging fruit.



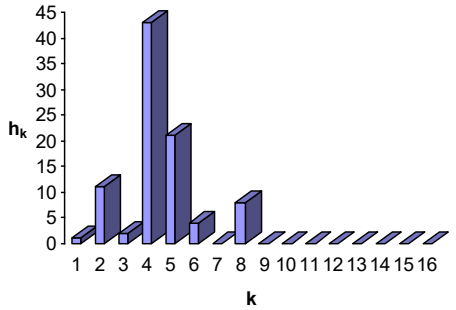
Detectability Profile of a unit under test

- The Detectability Profile of a unit under test describes how the defects are distributed relative to their detectability.
- Total M faults, total N possible input combinations. The set of faults can be partitioned into these subsets:
- $H = \{h_1, h_2, \ldots h_N\}$
- Where h_k is the number of faults detectable by exactly k inputs. The vector H describes the detectability profile.
  - h_1 is the number of faults that are hardest to find.
  - As testing and debugging continues, harder to find faults will tend to remain. Easy to find faults will get eliminated soon.

Ex: CECL Full adder Inputs=4 (N=16)

$H=(h_1,h_2,h_3,h_4,h_5,h_6,h_8)$

$=(1,11,2,43,21,4,8)$



Y. K. Malaiya, S. Yang, "The Coverage Problem for Random Testing," IEEE Int Test Conf. 1984, pp. 237-245.

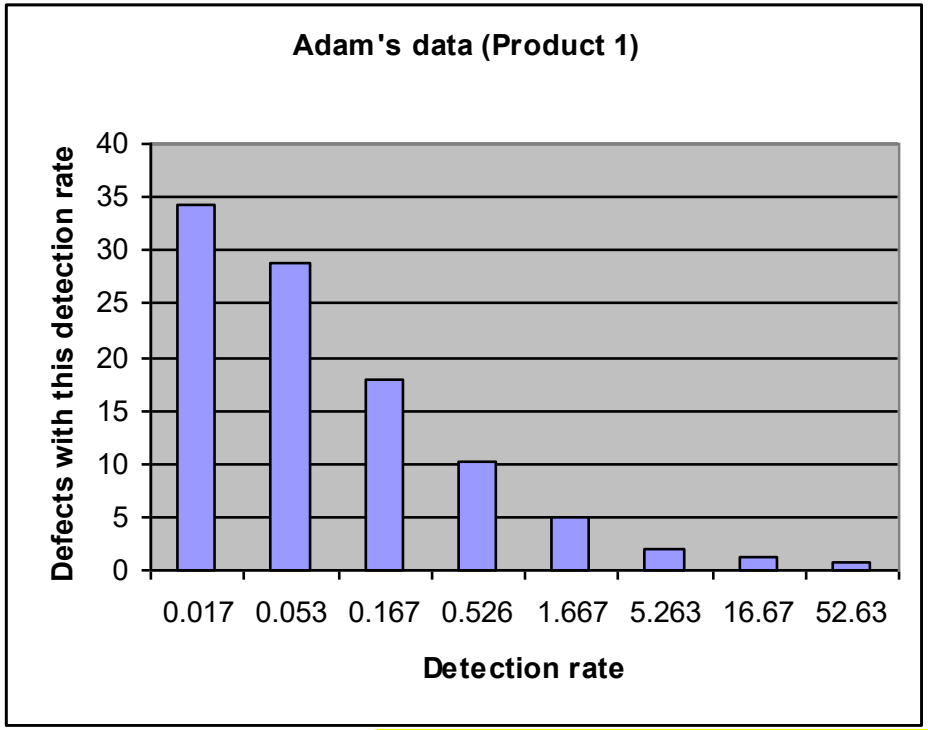
K. Wagnor, C. Chin, and E. McCluskey, "Pseudorandom testing". IEEE Trans. Computer, Mar. 1987, pp. 332—343.

Detectability Profile: software

- Regardless of initial profile, after some initial testing, the profile will become asymmetric.
- In the early development phases, inspection and early testing are likely to remove most easy to test bugs, while leaving almost all hardest to test bugs still in.

Detectability Profile: software

- Adam's Data for a large IBM software product. Note bugs with high detection rates are mostly gone.



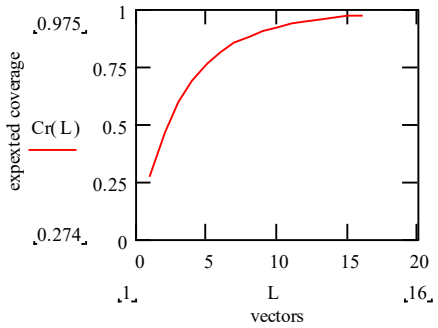
Adams, IBM Journal of Research and Development, Jan. 1984

Coverage Obtained by L Vectors

What fault coverage is achieved by applying L test vector?

- h_k out of M defects detectable by exactly k vectors: detection probability k/N
- P{a defect with dp k/N not detected by a vector} = $(1-\frac{k}{N})$
- P{a defect with dp k/N not detected by L vectors} = $(1-\frac{k}{N})^L$
- Of h_k faults, expected number not covered is $(1-\frac{k}{N})^L h_k$
- Expected test coverage with L vectors

$$C(L) = 1 - \sum_{k=1}^{N} (1-\frac{k}{N})^L \frac{h_k}{M}$$



Y.K. Malaiya and S. Yang, ""The Coverage Problem for Random Testing" Proc. International Test Conference, October 1984, pp. 237-245.

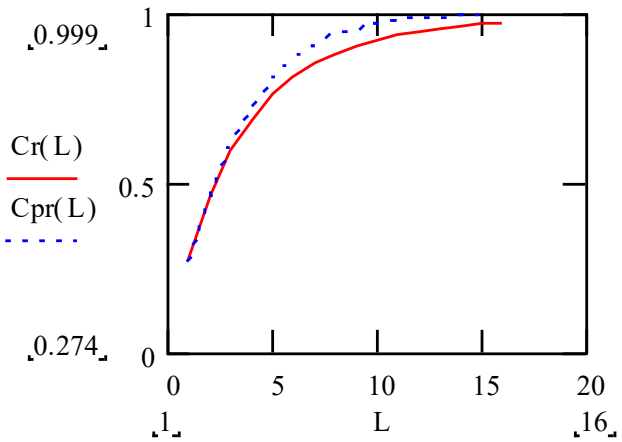
Coverage Obtained by L Vectors

Pseudorandom (PR) testing: a vector cannot repeat, unlike in true Random testing.

- For PR tests (McClusky 87)

$$C(L) = 1 - \sum_{k=1}^{N-L} \frac{{}^{N-L}C_k}{{}^{N}C_k} \frac{h_k}{M}$$

$$\approx 1 - \sum_{k=1}^{N} (1 - \frac{k}{N})^L \frac{h_k}{M} \text{ (for Random)}$$



- For large L, terms with only low k (i.e. faults that are hard to test) have an impact. Thus only lower elements of H need to be estimated.
- For CECL Full Adder,

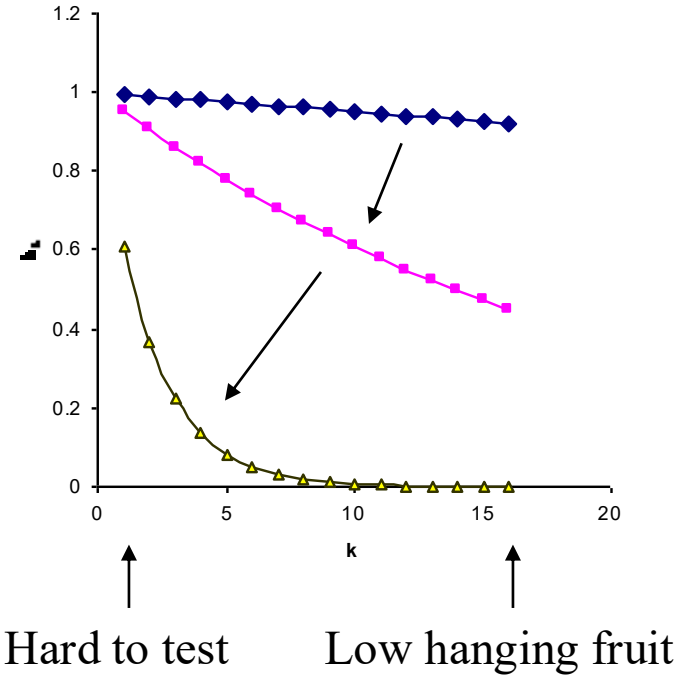
$$C(15) = 1 - [4.2 + 16.4 + 0.9 + 6.3 + 0.84 + 0.03 + 0 + ...] \cdot 10^{-3}$$

K. Wagnor, C. Chin, and E. McCluskey, "Pseudorandom testing. IEEE Trans. Computer, Mar. 1987, pp. 332—343.

Detectability Profile: Software

- Software detectability profile is exponential
- Justification: Early testing will find & remove easy-to-test faults.
- Testing methods need to focus on hard-to-find faults.

As testing time progresses, more of the faults are clustered to the left.



Directed testing

Testing may be directed rather than random because

- Tester may wish to focus on functionality not adequately exercised by random testing (for example recovery code)
- Tester may wish to focus on more critical sections of the code.
- The probability of detecting a fault can be give by p_i, where p_i may be *greater or less than* k/N.

  P{a defect with dp p_i *not* detected by L vectors} = $(1 - p_i)^L$

- Where $p_i > \frac{k}{N}$ if the previous tests are not repeated, or the test has a good idea of where to look.
- When the *exhaustive set* (ES) of inputs are applied, then P{a defect with dp p_i not detected by ES} ≈ 0
  - Unlikely in most real situations.

Some common models

- Several models for ordinary bug finding process. Termed **Software Reliability Growth Models** (SRGMs).
- **Exponential SRGM**: assumes bug finding rate $\lambda(t)$ is proportional to remaining bugs at time N(t).

$$\lambda(t) = -\frac{dN(t)}{dt} = \beta_1 N(t)$$

- Which has the solution

$$\lambda(t) = \beta_0 \beta_1 e^{-\beta_1 t}$$

- Where β_0 and β_1 are parameters to be determined. B_0 represents the initial number of bugs and β_1 a measure of test effectiveness.

Defect Density

- Exponential defect finding model is
$$\lambda(t) = \beta_0 \beta_1 e^{-\beta_1 t}$$
- β_0 represents the initial number of bugs.
- If the initial *defect density* is D(0), and the software size (measured in 1000 lines of code, i.e. KLOC) is S, then
$$\beta_0 = D(0) \times S$$
- The initial defect density is a function of the software development process and the degree of prior defect removal.
- The defect finding rate gradually declines, it takes infinite time to find them all according to the exponential model.
- The final defect density is sometimes used as a release criterion.

SRGM : "Logarithmic Poisson"

- If testing combines random and directed testing, the Logarithmic Poisson arises.
- **Logarithmic Poisson** model, by Musa-Okumoto, has been found to have a good predictive capability

$$\mu(t) = \beta_o \ln(1 + \beta_1 t) \qquad \lambda(t) = \frac{\beta_o \beta_1}{1 + \beta_1 t}$$

- Applicable as long as $\mu(t) \leq N(0)$. Practically always satisfied.
- Parameters β_o and β_1 don't have a simple interpretation. An interpretation has been given by Malaiya and Denton (What Do the Software Reliability Growth Model Parameters Represent?).

References

- Y. K. Malaiya, S. Yang, “The Coverage Problem for Random Testing,” IEEE International Test Conference 1984, pp. 237-245.
- Y.K. Malaiya, A. von Mayrhauser and P. Srimani, “An Examination of Fault Exposure Ratio,” IEEE Trans. Software Engineering, Nov. 1993, pp. 1087-1094.
- S. C. Seth, V. D. Agrawal, H. Farhat, "A Statistical Theory of Digital Circuit Testability," IEEE Trans. Computers, 1990, pp. 582-586.
- K. Wagnor, C. Chin, and E. McCluskey, “Pseudorandom testing. IEEE Trans. Computer, Mar. 1987, pp. 332—343.
- E. N. Adams, "Optimizing Preventive Service of Software Products," in IBM Journal of Research and Development, vol. 28, no. 1, pp. 2-14, Jan. 1984.
- J R Dunham, "Experiments in software reliability: Life-critical applications," IEEE Tran. SE, January 1986, pp. 110 - 123
- H. Hashempour, F.J. Meyer, F. Lombardi,, "Analysis and measurement of fault coverage in a combined ATE and BIST environment," Instrumentation and Measurement, IEEE Transactions on , vol.53, no.2, pp.300,307, April 2004.
- Y. K. Malaiya, "Assessing Software Reliability Enhancement Achievable through Testing", Recent Advancements in Software Reliability Assurance 2019, pp. 107-138