Institute of
Software Security

TUHH
Hamburg University of Technology

Code Quality

Riccardo Scandariato
Institute of Software Security, TUHH, Germany
ric***do . scanda***to @ tuhh.de

MSc Course ”Secure Software Engineering”

Learning objectives

- Understand code review activities for security

Reading material
Michael Howard, *A Process for Performing Security Code Reviews*, IEEE Security & Privacy, July 2006

- Understand emerging techniques (APR) to fix security bugs automatically in source code

Security Code Review

Apple ‘goto fail;’ vulnerability

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer
signedParams,uint8_t *signature, UInt16 signatureLen)
{
...
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
...
```

https://nvd.nist.gov/vuln/detail/CVE-2014-1266
https://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c

Apple ‘goto fail;’ vulnerability

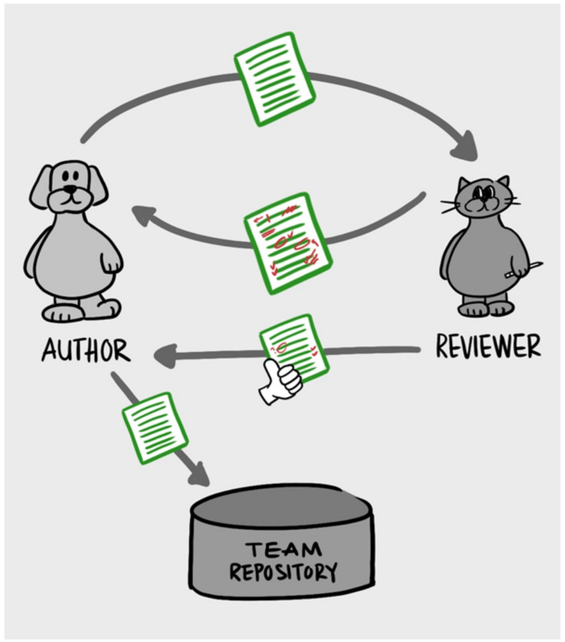
- Problem: two consecutive **`goto fail;`**
  - Indentation makes us think both statements run only when the if-predicate is true
  - **`err`** is returned with the value of zero
  - The caller will believe no error occurs while verifying the signature

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```

Could be identified easily by Code Review!

Security Code Review

Security Code Review: convene people *(reviewer)* to **find faults** in source code written by someone else *(author)*



- Early finding faults, quickly generating fixes
- Reduce testing effort

But also…

- Security code reviews also performed for compliance reasons

- E.g., Requirement 6.3.2 in Payment Card Industry Data Security Standard (PCI-DSS) mandates a code review of custom code

Code Review Types

- Manual Code Review w/ Experts
  - Allow to use the knowledge from reviewer
  - Takes time, expertise and effort
- Code Review w/ Static Analysis Tools
  - Automated, could be very useful for large projects
  - Could produce many false positives

→ **Use both for better results**

Static Analysis

- Inspect code **without running** it to find bugs (common case!) or to gain confidence about bugs absence (i.e., reason about the program's correctness)
- Provide security warnings about (common) mistakes
(e.g., Buffer overflow, API misuse,...)
- E.g., **SonarQube**, **Checkmarx**, **Veracode**, **SpotBugs**, etc..

Why using Static Analysis for Code Review?

- Manual code review usually requires expertise in secure coding
    - Scarcity, cost
- Static Analysis could be integrated into CI/CD to run automatically
- Humans are imperfect and could miss faults (FN)

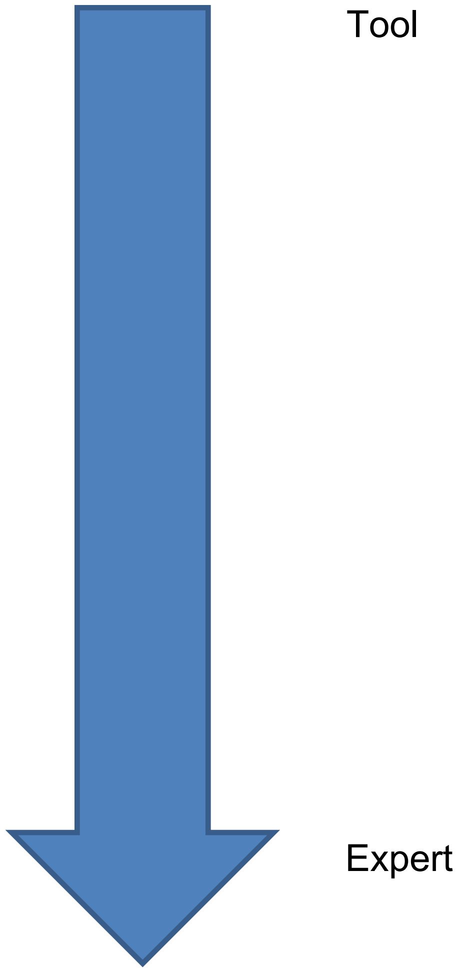
Static Analysis is covered in the **Software Security** course :)

Humans or machines ?

- Track taint
- Find credentials in code or config file
- Correct use of Crypto API
- Handle sensitive data with care

Humans vs machines ;)

- Track taint
  - SAST tool effective
  - Too complex for human?
- Find credentials in code or config file
  - All SAST tools have this rule
  - Human can do that too (but useful?)
- Correct use of Crypto API
  - Only specialized tools exist (academic)
  - Human
- Handle sensitive data with care
  - Tool: def of sensitive?
  - Human

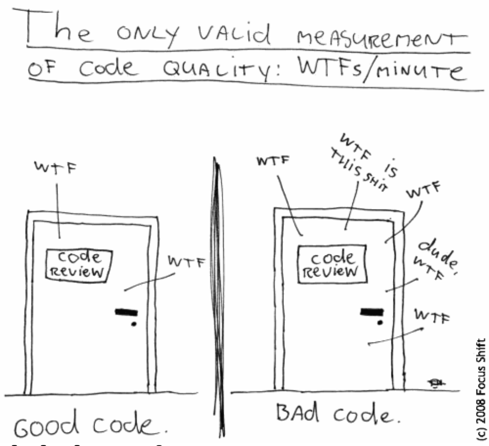


Manual Code Review

Code Review Guidelines

- IEEE Standard for Software Reviews and Audits, IEEE Std 1028-2008 [1]
  - Not specific to security
  - Def of terms and roles
- OWASP Code Review Guideline [2]
  - Focus on reviewing code for security Top 10 vulnerabilities **(220 pages !!!)**

[1] https://standards.ieee.org/standard/1028-2008.html
[2] https://owasp.org/www-project-code-review-guide



Roles in Code Review

- **Author** (who writes code)
  - Can answer any specific questions, or reveal blind spots
- **Reader** (reviewer, not the author)
  - Leads through the review
- **Scribe/Recorder**
  - Documents faults, actions, decisions made in the meeting
- **Inspection Leader/Moderator**
  - Planning and organizational tasks
  - Moderate review meeting
  - Organize follow-up on issues

IEEE Standard for Software Reviews and Audits, IEEE Std 1028-2008

Code Reviewers

- IEEE standard: People with *readability*, but *objectivity*
  - e.g. system architect
  - e.g. developer working on the same project, but different team
- **The above is not valid for security !!!**
  - You need people experienced with security, e.g., consultants, experienced developers
- Including more than four generally slows the process
  - People tend to argue
  - Getting side-tracked on unrelated issues

Code Review Process

- Code review processes vary widely in their formality
- e.g., **Inspection** – most formal process
  - Separated roles
  - Usage of Checklists
  - Formal collection of defect metrics
- e.g., **Walkthrough** – less formal process
  - Author (also acting as Moderator) and Reader
  - Driven by author’s goals
- **Anything in between**

Checklists for Code Review

- Identify relevant aspects

- Walk through the functionality of the code
  - Look for too much complexity, functionality
  - Look for common defensive coding mistakes
  - Look for Common Vulnerabilities

Example: Checklist for crypto issue

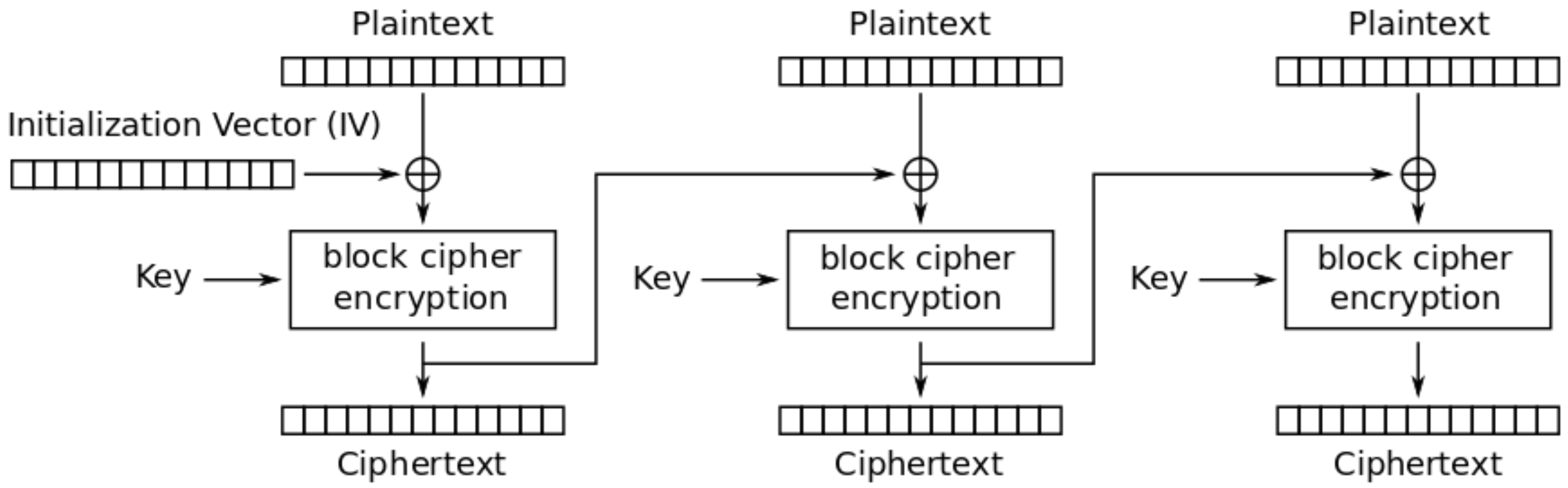
```
// Simple Java code to encrypt/decrypt data
private static byte[] encrypt(byte[] raw, byte[] clear) throws Exception {
    SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
    byte[] encrypted = cipher.doFinal(clear);
    return encrypted;
}

private static byte[] decrypt(byte[] raw, byte[] encrypted) throws
Exception {
    SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, skeySpec);
    byte[] decrypted = cipher.doFinal(encrypted);
    return decrypted;
}
```

Using **AES** in **CBC mode** (default) is insecure

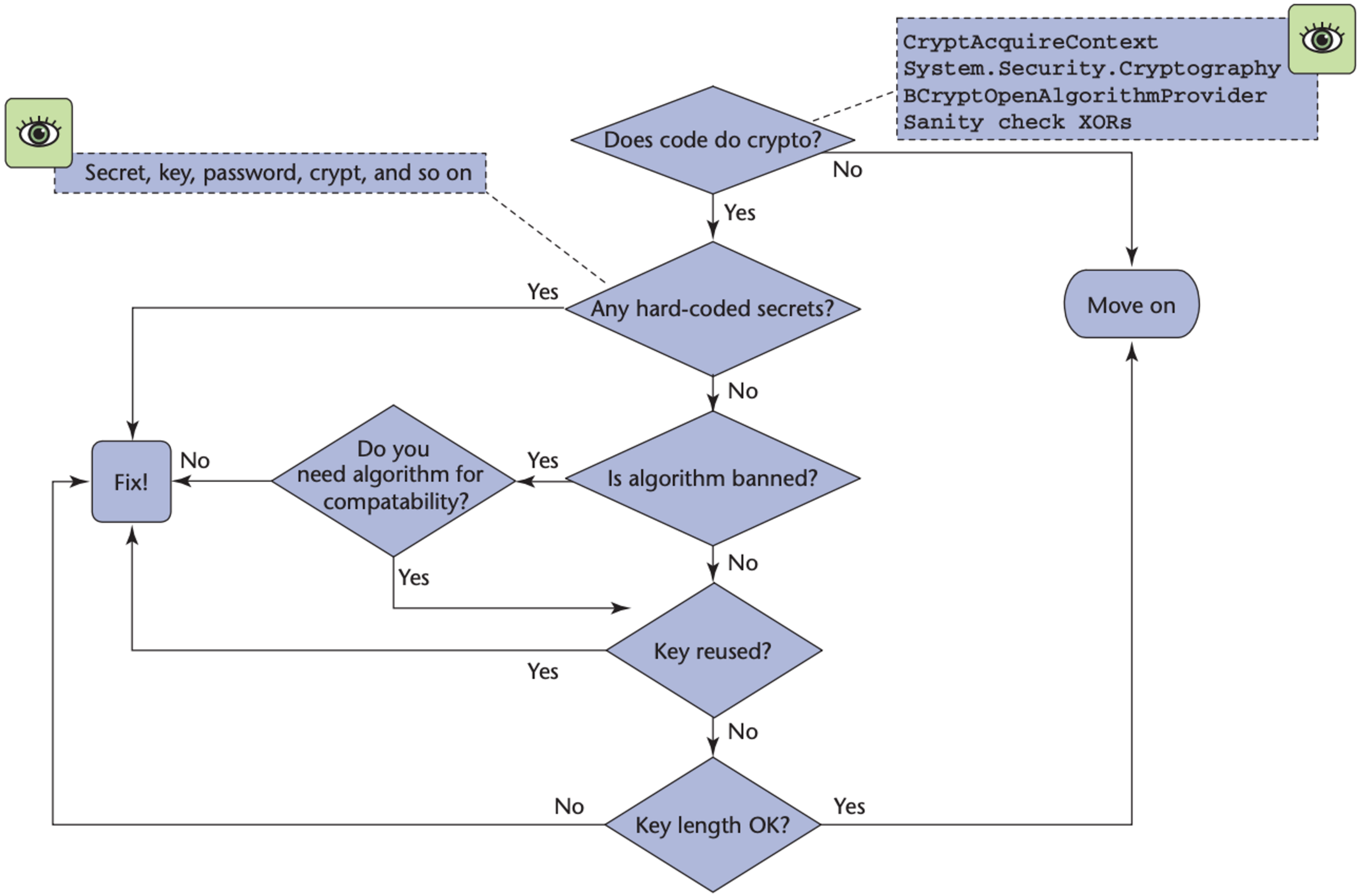
Only for your info

CBC



Cipher Block Chaining (CBC) mode encryption

Example: Checklist for crypto issue



Michael Howard, "A Process for Performing Security Code Reviews". IEEE Security & Privacy.

Example: OWASP checklist for SQLi

- Review all code that calls EXECUTE, EXEC, any SQL calls that can call outside resources or command line
  - Always validate user input by testing type, length, format, and range
  - Test the content of string variables and accept only expected values
  - Never build SQL statements directly from user input
  - Use SQL API provided by platform. i.e. Parameterized Statements
  - …

OWASP Code Review Guideline
https://owasp.org/www-project-code-review-guide

Fatigue

- In this type of activity, people get tired quickly
  - Sessions of max 2 hours
  - Max 2 sessions per day
- What to do in case of larger apps?
  - Set priorities!

Priorities: where to start?

- Code listening on a globally accessible network interface
- Code that runs with elevated privileges
- Code that handles sensitive data

Security considerations

- Old/unmaintained code
- Code with a history of vulnerabilities
- Complex code
- Code that changes frequently

Software engineering considerations

Study at home

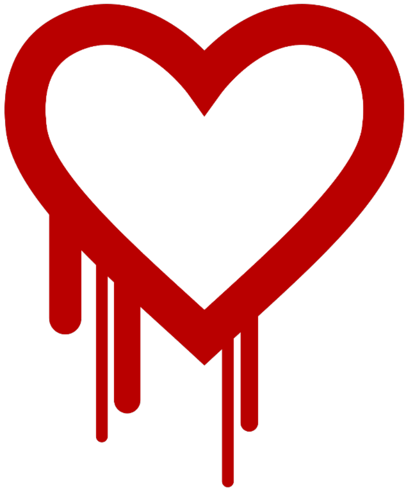
Code review effective?

- How important are these activities (code review) to assure the code quality?

- Recent research found a huge change in development process of an open-source project [1]
  - After a vulnerability scandal

[1] James Walden. The Impact of a Major Security Event on an Open Source Project: The Case of OpenSSL. MSR 2020.

OpenSSL & Heartbleed Vulnerability



- **OpenSSL**
  - One of most common used libraries
  - Secure communications over internet
- **Heartbleed**
  - Discovered in 2014
  - Exploited a buffer over-read vulnerability in the cryptography library of OpenSSL
  - Two-thirds of https-enabled websites worldwide were affected

Study at home

OpenSSL responses after the Heartbleed

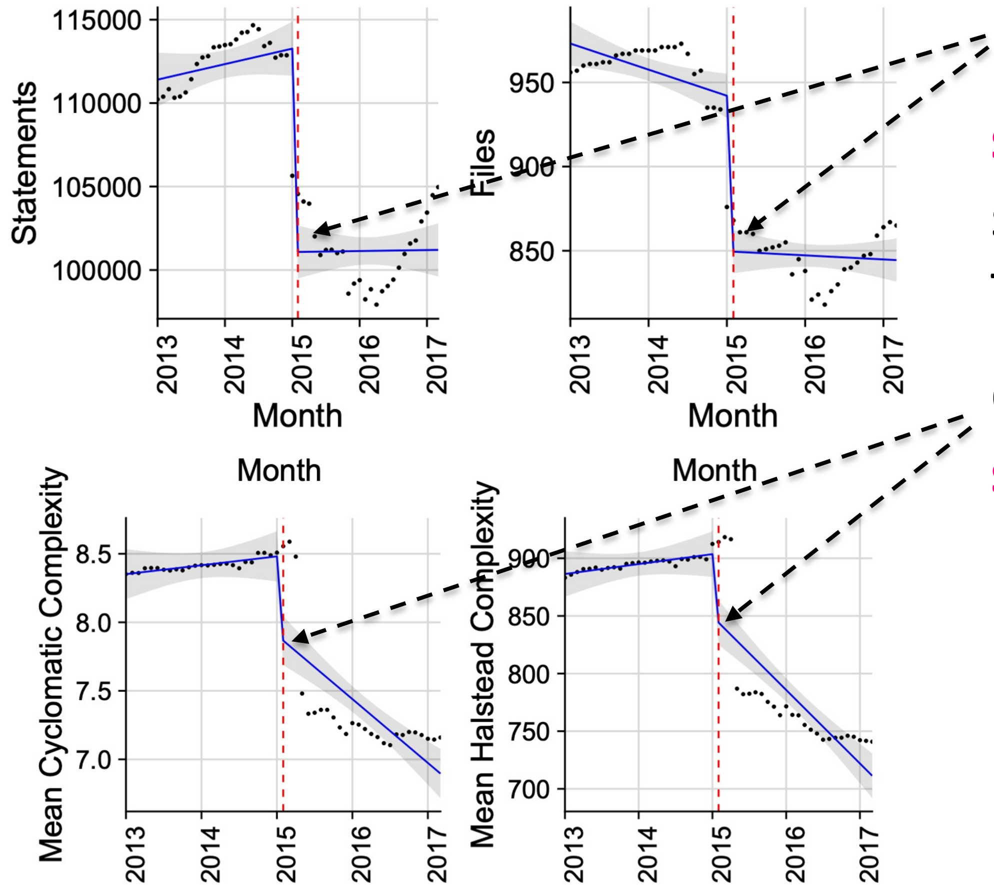
Sep 2014	Publication of security policy on vulnerability handling
2015	Code commits require review before merged
Feb 2015	All code base are reformatted to follow one coding style
Mar 2016	Add tools (directory `/fuzz`) for supporting easy fuzzing
Aug 2016, Sep 2018	Release new versions: remove old algorithms/protocols (3DES, RC4, SSLv2), support for TLS 1.3, SHA3

James Walden. The Impact of a Major Security Event on an Open Source Project: The Case of OpenSSL. MSR 2020.

Study at home

OpenSSL metrics after the Heartbleed

Code Size and Complexity



Project size **decreases significantly** then increases slowly (with new features TLS 1.3, SHA3)

Code complexity **decreases significantly** and remains low

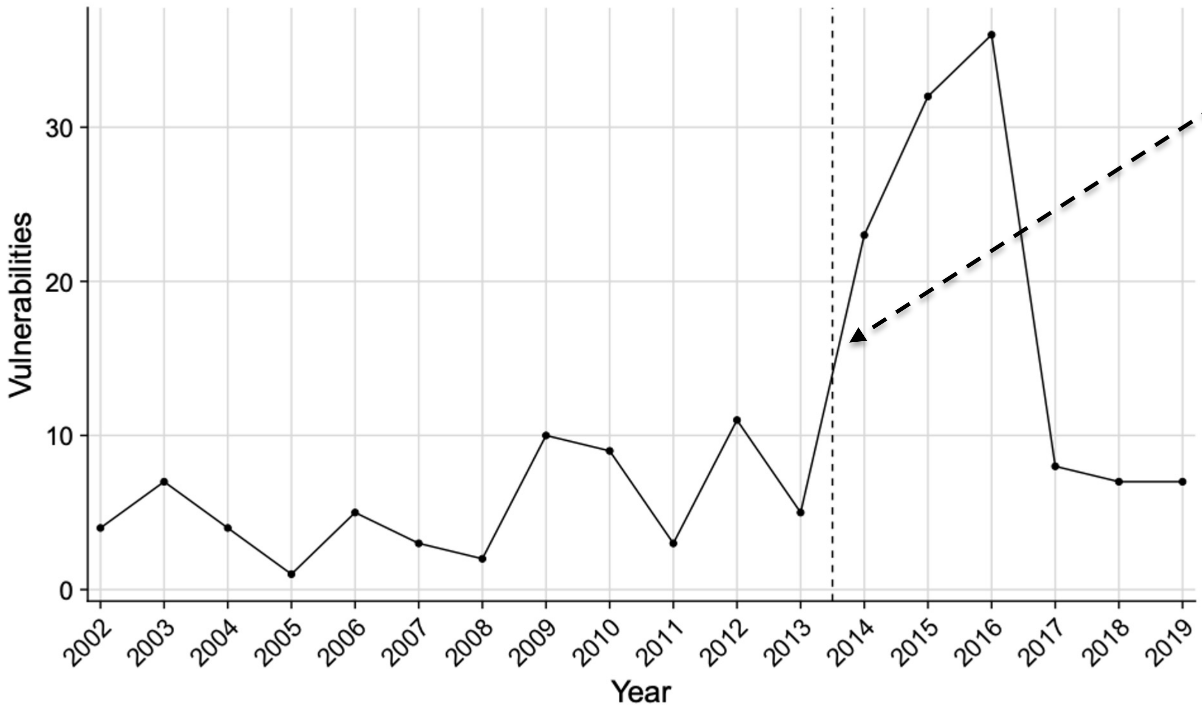
→ Results of refactoring and reformatting code

James Walden. The Impact of a Major Security Event on an Open Source Project: The Case of OpenSSL. MSR 2020.

Study at home

OpenSSL metrics after the Heartbleed

# of vulnerabilities



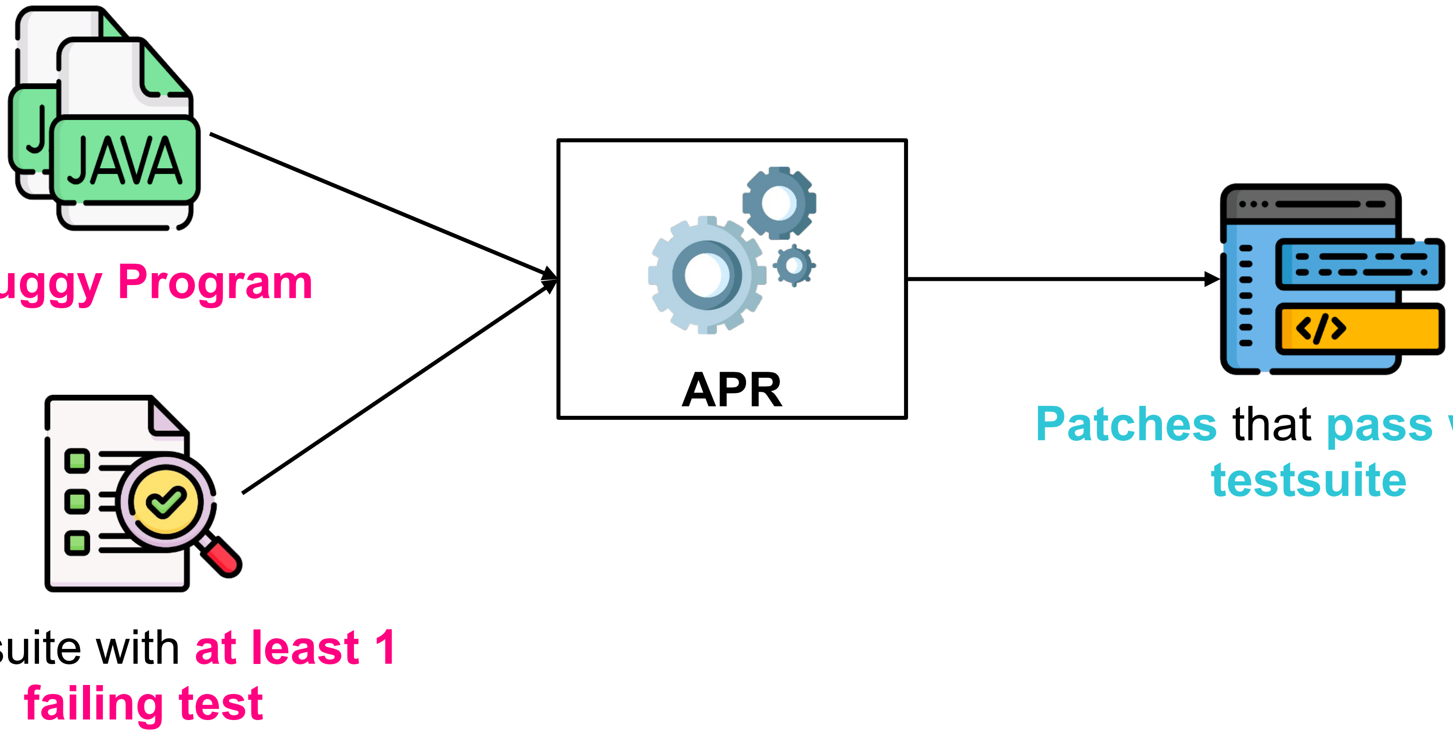
# of vulnerabilities found **increase dramatically**

Figure 1: OpenSSL Vulnerabilities Reported by Year

James Walden. The Impact of a Major Security Event on an Open Source Project: The Case of OpenSSL. MSR 2020.

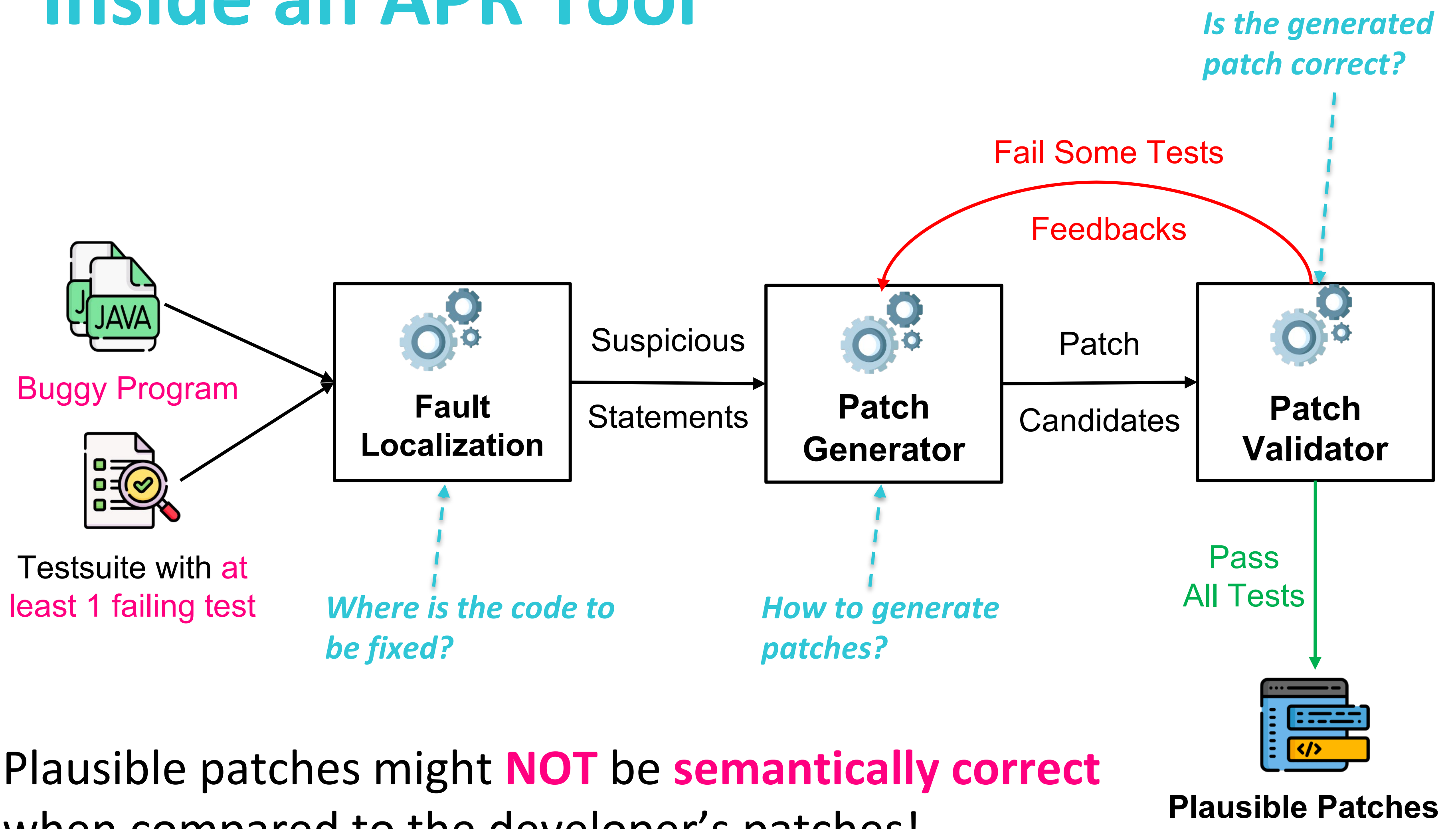
Automated Vulnerability Repair in Source Code

Automated Program Repair - APR


JAVA
Buggy Program
APR
Patches that pass whole testsuite
Testsuite with at least 1 failing test

Automated Program Repair aims to repair software bugs automatically, help to reduce or even remove human intervention from bug fixing process

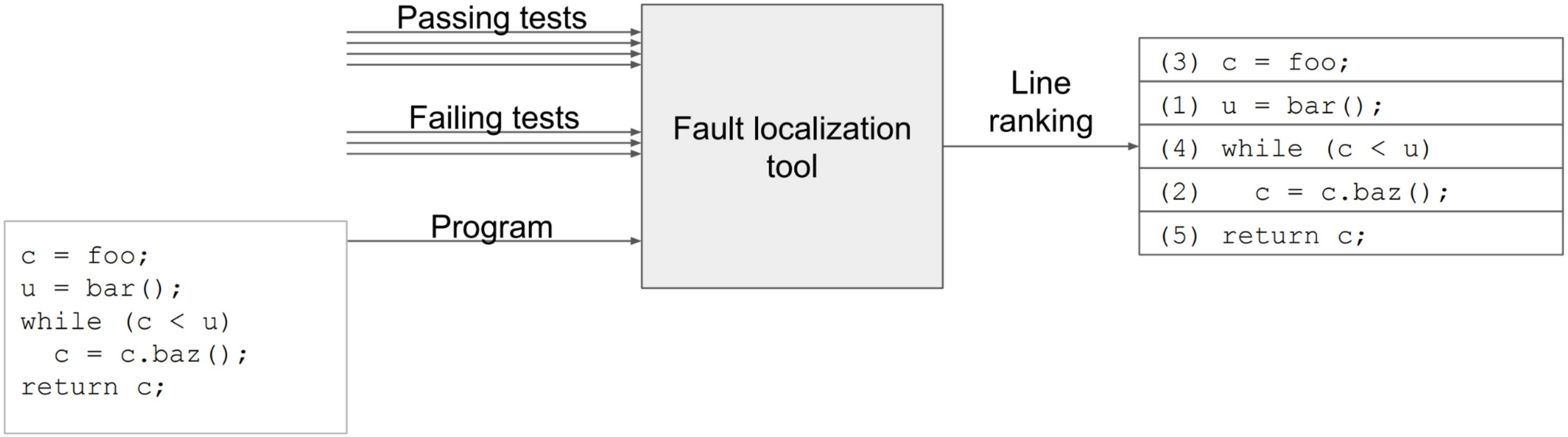
Inside an APR Tool



Plausible patches might **NOT** be **semantically correct** when compared to the developer's patches!

→ Overfitting Problem of APR

Fault Localization



```
c = foo;
u = bar();
while (c < u)
  c = c.baz();
return c;
```

```
(3) c = foo;
(1) u = bar();
(4) while (c < u)
(2)   c = c.baz();
(5) return c;
```

- Based on testing results
- Spectrum-based Fault Localization (SBFL)
- Mutation-based Fault Localization (MBFL)

https://homes.cs.washington.edu/~mernst/pubs/fault-localization-icse2017-slides-long.pdf

Only for your info

Mutant-based Fault Localization (MBFL)

- Change a single line of code
- Execute P/F tests
- Collect results
- Compute **suspiciousness** and sort accordingly

- VERY expensive
  (run all the tests on a large number of mutants)

Spectrum-based Fault Localization

- Leveraging the coverage information of passing tests and failing tests
- The more **failing** tests execute the statement **S**, the more suspicious it is
- Many similarity coefficients to compute **suspiciousness**

$$Ochiai = \frac{a_{11}}{\sqrt{(a_{11} + a_{01}) \times (a_{11} + a_{10})}}$$

$$Tarantula = \frac{\frac{a_{11}}{a_{11} + a_{01}}}{\frac{a_{11}}{a_{11} + a_{01}} + \frac{a_{10}}{a_{10} + a_{00}}}$$

a_{11} = executed and failed

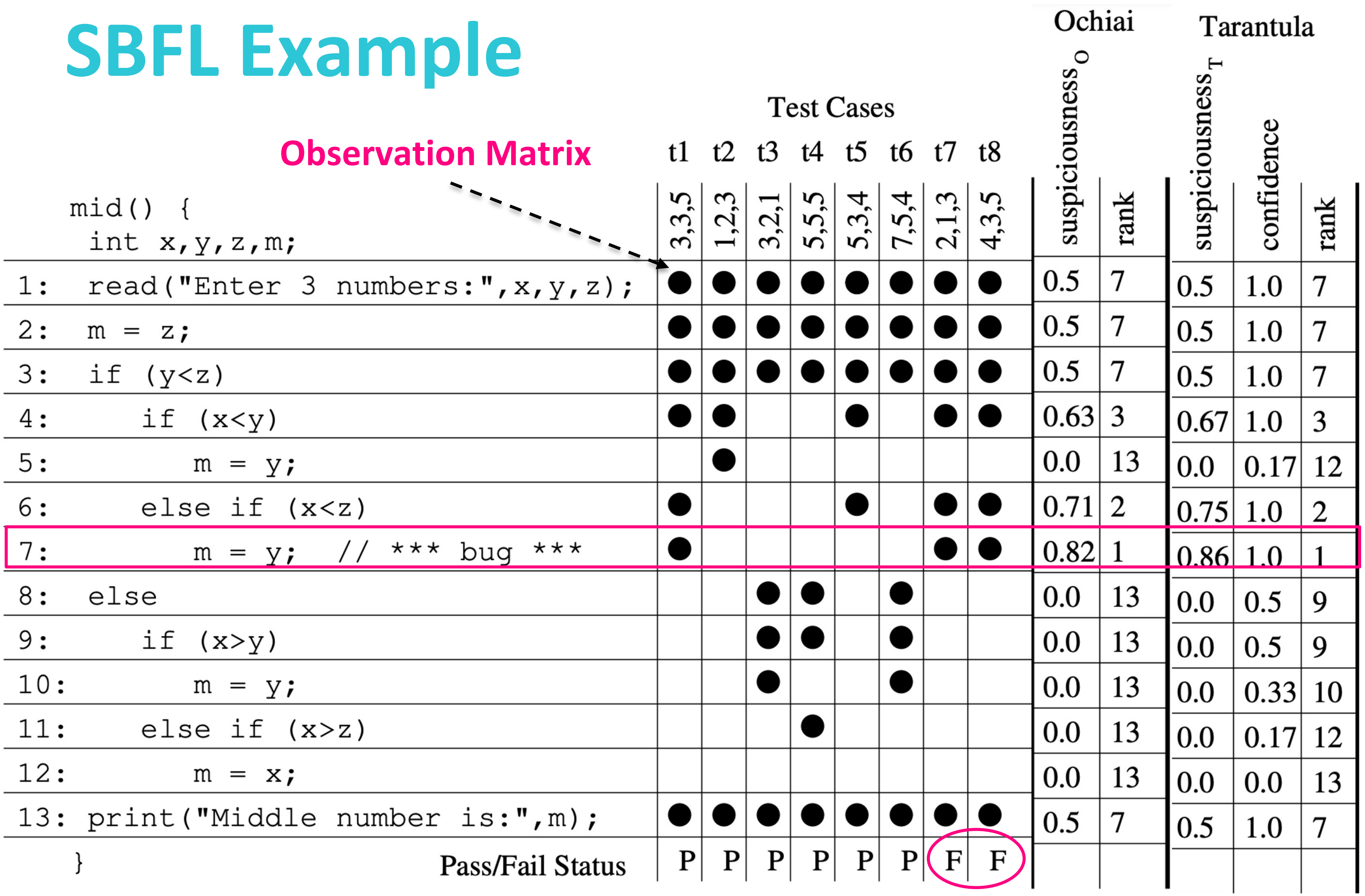
a_{10} = executed and passed

a_{01} = not executed and failed

a_{00} = not executed and passed

SBFL Example

Observation Matrix



	Test Cases								Ochiai		Tarantula		
	t1	t2	t3	t4	t5	t6	t7	t8					
`mid() {` `int x,y,z,m;`	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	7,5,4	2,1,3	4,3,5	suspiciousness$_O$	rank	suspiciousness$_T$	confidence	rank
`1: read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●	●	●	0.5	7	0.5	1.0	7
`2: m = z;`	●	●	●	●	●	●	●	●	0.5	7	0.5	1.0	7
`3: if (y<z)`	●	●	●	●	●	●	●	●	0.5	7	0.5	1.0	7
`4:     if (x<y)`	●	●			●		●	●	0.63	3	0.67	1.0	3
`5:         m = y;`		●							0.0	13	0.0	0.17	12
`6:     else if (x<z)`	●				●		●	●	0.71	2	0.75	1.0	2
`7:         m = y;   // *** bug ***`	●						●	●	0.82	1	0.86	1.0	1
`8: else`			●	●		●			0.0	13	0.0	0.5	9
`9:     if (x>y)`			●	●		●			0.0	13	0.0	0.5	9
`10:        m = y;`			●			●			0.0	13	0.0	0.33	10
`11:    else if (x>z)`				●					0.0	13	0.0	0.17	12
`12:        m = x;`									0.0	13	0.0	0.0	13
`13: print("Middle number is:",m);`	●	●	●	●	●	●	●	●	0.5	7	0.5	1.0	7
`}` Pass/Fail Status	P	P	P	P	P	P	F	F					

Yanbing Yu et. al. An empirical study of the effects of test-suite reduction on fault localization. ICSE 2008.

APR Technique Families

Patch Generator

- Heuristics-based Repair (Aka Generate-and-Test Repair)
- Constraint-based Repair (Aka Synthesis-based Repair)
  - Based on symbolic execution
- Learning-based Repair
  - E.g., deap learning on AST-to-AST transformation templates that summarize how patches modify buggy code into correct code
  - Learning can also be used to assess the generated patch (”similarity with regard to the code corpus”)

Claire Le Goues, Michael Pradel, Abhik Roychoudhury, *Automated Program Repair*, Communications of ACM, 2019

APR for Security Vulnerabilities

- Security vulnerabilities don’t often come with proof-of-vulnerability test cases
  - SBFL could not be applied to locate the faults
- FL module could be replaced/ combined with SAST tools or in-house prediction techniques
  - SAST tools provide useful information about the location and the presence of vulnerable code
- Only few work done for vulnerability repair in the literature so far

APR for Security Vulnerabilities

- C/C++
  - Zhen Huang et al. **Using Safety Properties to Generate Vulnerability Patches**. S&P'19.
  - Jacob Harer et al. **Learning to Repair Software Vulnerabilities with Generative Adversarial Networks**. NIPS'18.
- Java: Siqi Ma et al. **VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples**. ESORICS'17.
- Android: Ruian Duan et al. **Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries**. NDSS'19.