Joint Degree Program of Fudan University and University College Dublin (UCD)

SOFT620020.02
Advanced Software Engineering

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

https://chenbihuan.github.io

Course Outline

Date	Topic
Sep. 10	Introduction
Sep. 17	Testing Overview
Sep. 24	Holiday
Oct. 01	Holiday
Oct. 08	Guided Random Testing
Oct. 15	Search-Based Testing
Oct. 22	Performance Analysis
Oct. 29	Security Testing

Date	Topic
Nov. 05	Compiler Testing
Nov. 12	Mobile Testing
Nov. 19	Delta Debugging
Nov. 26	Presentation 1
Dec. 03	Bug Localization
Dec. 10	**Automatic Repair**
Dec. 17	Symbolic Execution
Dec. 24	Presentation 2

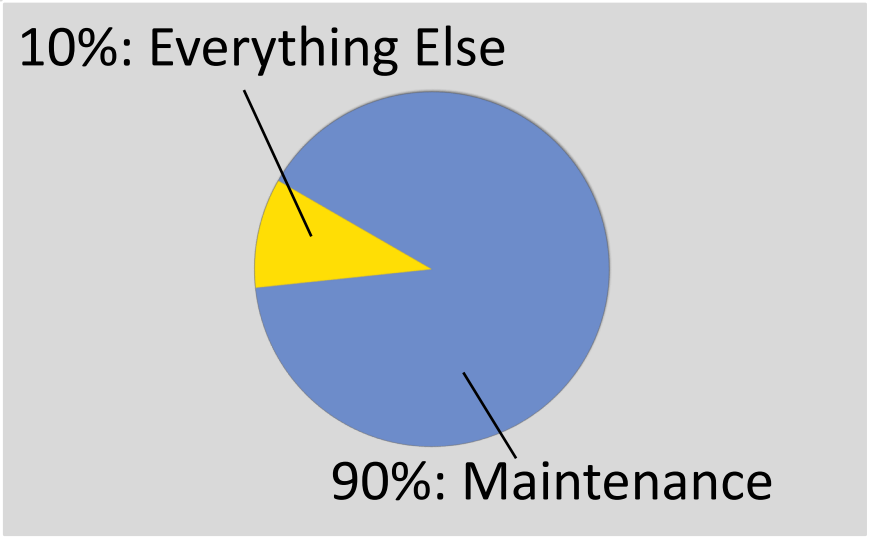
Cost of Buggy Software

"Everyday, almost 300 bugs appear […] far too many for only the Mozilla programmers to handle."

– Mozilla Developer, 2005

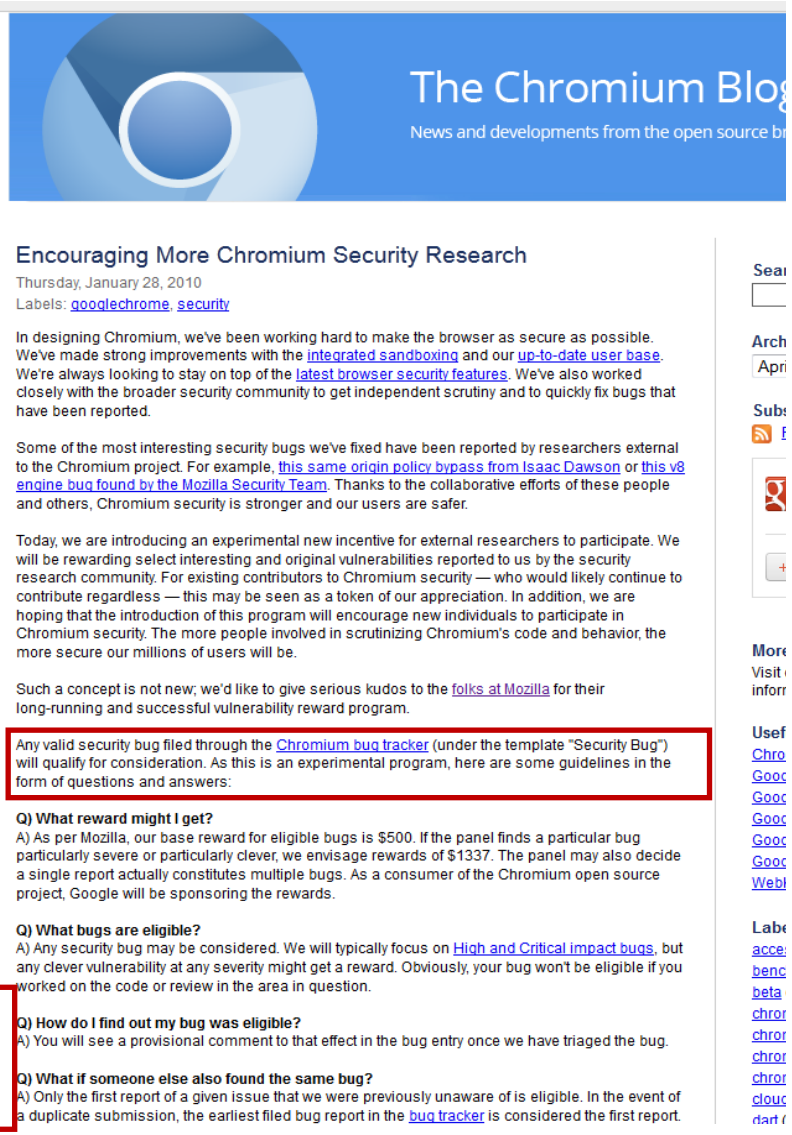
Annual cost of software errors in the US: $59.5 billion (0.6% of GDP).

Average time to fix a security-critical error: 28 days.



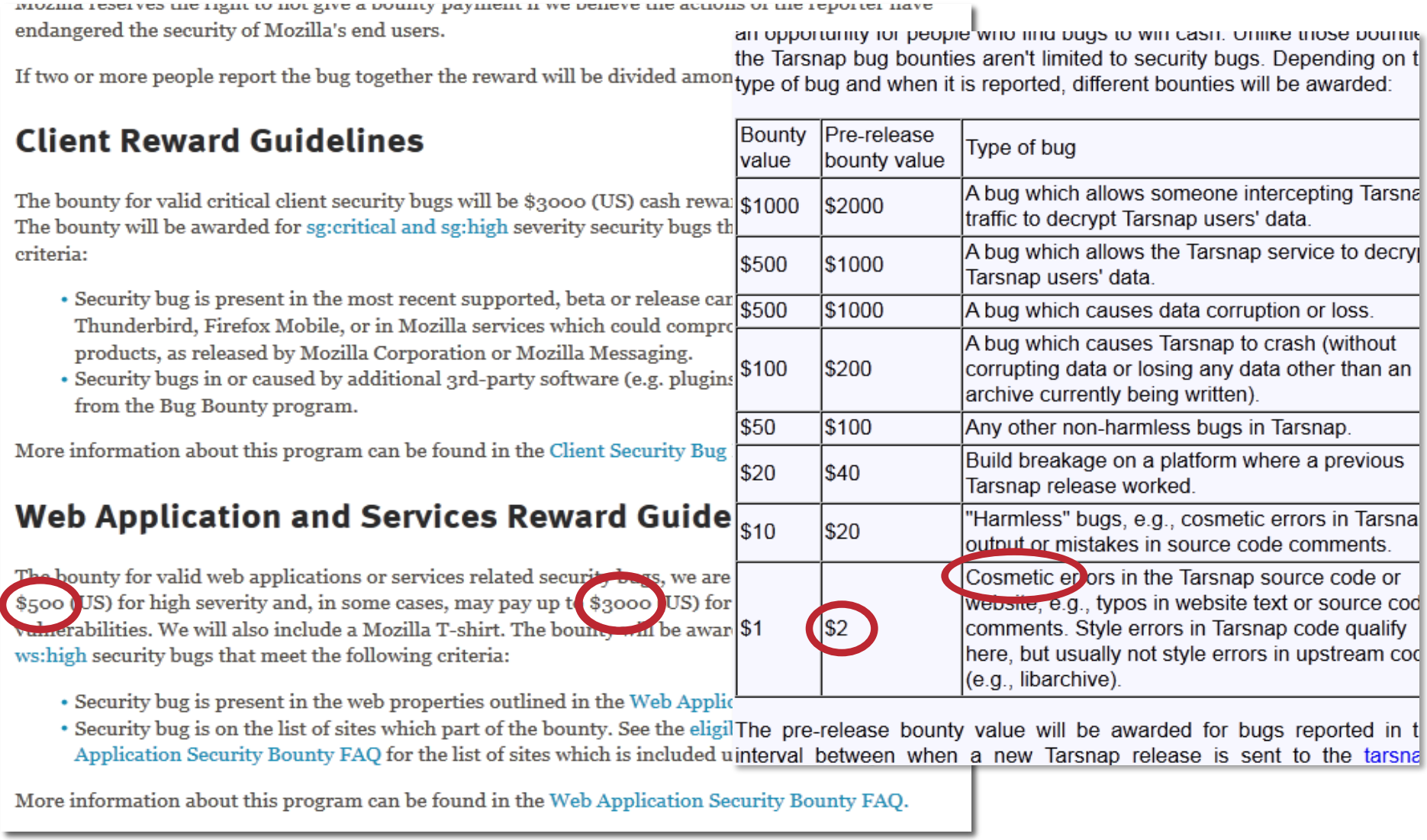
How Bad Is It?


mozilla
About Us
Community Map
Our Projects
Get
Bug Bounty Program
Introduction
The Mozilla Security Bug Bounty Program is designed to encourage security research in Mozilla softwa
and to reward those who help us create the safest Internet clients in existence.
Many thanks to Linspire and Mark Shuttleworth, who provided start-up funding for this endeavor.
General Bounty Guidelines


The Chromium Blog
News and developments from the open source br
Encouraging More Chromium Security Research
Thursday, January 28, 2010
Labels: googlechrome, security
In designing Chromium, we've been working hard to make the browser as secure as possible. We've made strong improvements with the integrated sandboxing and our up-to-date user base. We're always looking to stay on top of the latest browser security features. We've also worked closely with the broader security community to get independent scrutiny and to quickly fix bugs that have been reported.
Some of the most interesting security bugs we've fixed have been reported by researchers external to the Chromium project. For example, this same origin policy bypass from Isaac Dawson or this v8 engine bug found by the Mozilla Security Team. Thanks to the collaborative efforts of these people and others, Chromium security is stronger and our users are safer.
Today, we are introducing an experimental new incentive for external researchers to participate. We will be rewarding select interesting and original vulnerabilities reported to us by the security research community. For existing contributors to Chromium security — who would likely continue to contribute regardless — this may be seen as a token of our appreciation. In addition, we are hoping that the introduction of this program will encourage new individuals to participate in Chromium security. The more people involved in scrutinizing Chromium's code and behavior, the more secure our millions of users will be.
Such a concept is not new; we'd like to give serious kudos to the folks at Mozilla for their long-running and successful vulnerability reward program.
Any valid security bug filed through the Chromium bug tracker (under the template "Security Bug") will qualify for consideration. As this is an experimental program, here are some guidelines in the form of questions and answers:
Q) What reward might I get?
A) As per Mozilla, our base reward for eligible bugs is $500. If the panel finds a particular bug particularly severe or particularly clever, we envisage rewards of $1337. The panel may also decide a single report actually constitutes multiple bugs. As a consumer of the Chromium open source project, Google will be sponsoring the rewards.
Q) What bugs are eligible?
A) Any security bug may be considered. We will typically focus on High and Critical impact bugs, but any clever vulnerability at any severity might get a reward. Obviously, your bug won't be eligible if you worked on the code or review in the area in question.
Q) How do I find out my bug was eligible?
A) You will see a provisional comment to that effect in the bug entry once we have triaged the bug.
Q) What if someone else also found the same bug?
A) Only the first report of a given issue that we were previously unaware of is eligible. In the event of a duplicate submission, the earliest filed bug report in the bug tracker is considered the first report.


Tarsnap
Online backups for the truly paranoid
Tarsnap
News
About
Legal
Infrastructure
Bug Bounty
Winners
Design
Tarsnap Bug Bounties
According to Linus' Law, "given enough eyeballs, all bugs ar
This is one of the reasons why the Tarsnap client source code
available; but merely making the source code available doesn't
anything if people don't bother to read it.
For this reason, Tarsnap has a series of bug bounties. Sim
bounties offered by Mozilla and Google, the Tarsnap bug bount
an opportunity for people who find bugs to win cash. Unlike thos
the Tarsnap bug bounties aren't limited to security bugs. Depen

How Bad Is It? (cont.)



Mozilla reserves the right to not give a bounty payment if we believe the actions of the reporter have endangered the security of Mozilla's end users.

If two or more people report the bug together the reward will be divided amon

Client Reward Guidelines

The bounty for valid critical client security bugs will be $3000 (US) cash rewa
The bounty will be awarded for sg:critical and sg:high severity security bugs th
criteria:

- Security bug is present in the most recent supported, beta or release car
  Thunderbird, Firefox Mobile, or in Mozilla services which could compro
  products, as released by Mozilla Corporation or Mozilla Messaging.
- Security bugs in or caused by additional 3rd-party software (e.g. plugins
  from the Bug Bounty program.

More information about this program can be found in the Client Security Bug

Web Application and Services Reward Guide

The bounty for valid web applications or services related securi bugs, we are
$500 (US) for high severity and, in some cases, may pay up t $3000 US) for
vulnerabilities. We will also include a Mozilla T-shirt. The boun will be awar
ws:high security bugs that meet the following criteria:

- Security bug is present in the web properties outlined in the Web Applic
- Security bug is on the list of sites which part of the bounty. See the eligi
  Application Security Bounty FAQ for the list of sites which is included u

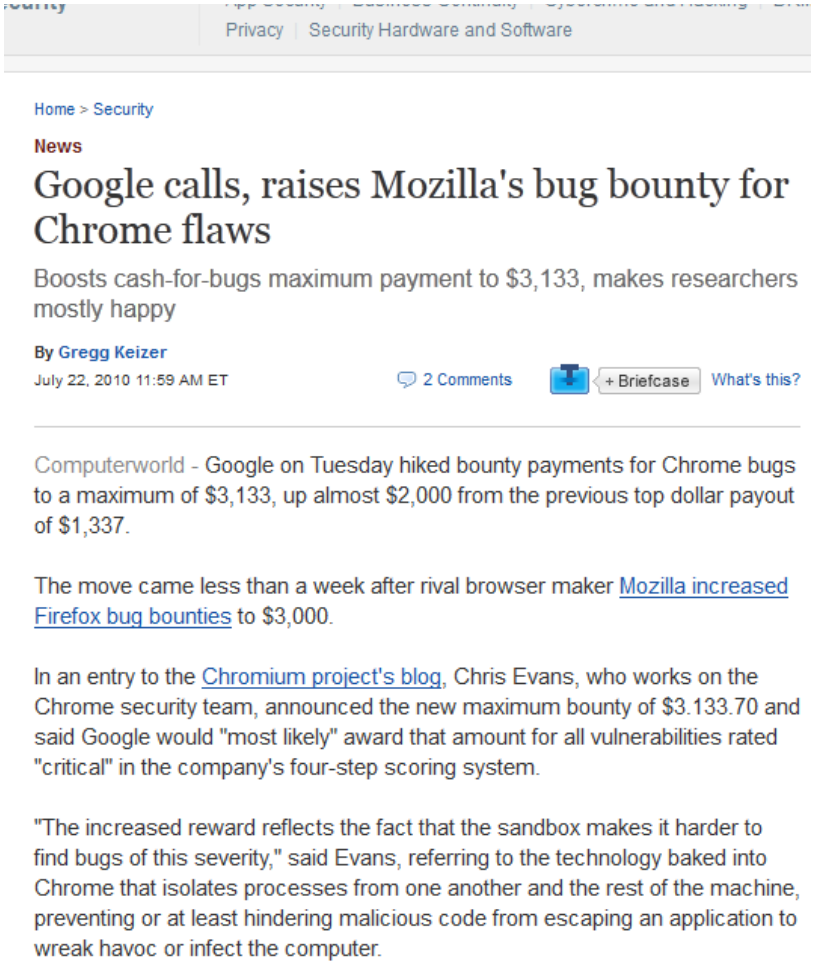
More information about this program can be found in the Web Application Security Bounty FAQ.

an opportunity for people who find bugs to win cash. Unlike those bounti
the Tarsnap bug bounties aren't limited to security bugs. Depending on t
type of bug and when it is reported, different bounties will be awarded:

Bounty value	Pre-release bounty value	Type of bug
$1000	$2000	A bug which allows someone intercepting Tarsna traffic to decrypt Tarsnap users' data.
$500	$1000	A bug which allows the Tarsnap service to decry Tarsnap users' data.
$500	$1000	A bug which causes data corruption or loss.
$100	$200	A bug which causes Tarsnap to crash (without corrupting data or losing any data other than an archive currently being written).
$50	$100	Any other non-harmless bugs in Tarsnap.
$20	$40	Build breakage on a platform where a previous Tarsnap release worked.
$10	$20	"Harmless" bugs, e.g., cosmetic errors in Tarsna output or mistakes in source code comments.
$1	$2	Cosmetic errors in the Tarsnap source code or website, e.g., typos in website text or source cod comments. Style errors in Tarsnap code qualify here, but usually not style errors in upstream cod (e.g., libarchive).

The pre-release bounty value will be awarded for bugs reported in t
interval between when a new Tarsnap release is sent to the tarsna

How Bad Is It? (cont.)



Privacy | Security Hardware and Software

Home > Security

News

Google calls, raises Mozilla's bug bounty for Chrome flaws

Boosts cash-for-bugs maximum payment to $3,133, makes researchers mostly happy

By Gregg Keizer

July 22, 2010 11:59 AM ET

2 Comments + Briefcase What's this?

Computerworld - Google on Tuesday hiked bounty payments for Chrome bugs to a maximum of $3,133, up almost $2,000 from the previous top dollar payout of $1,337.

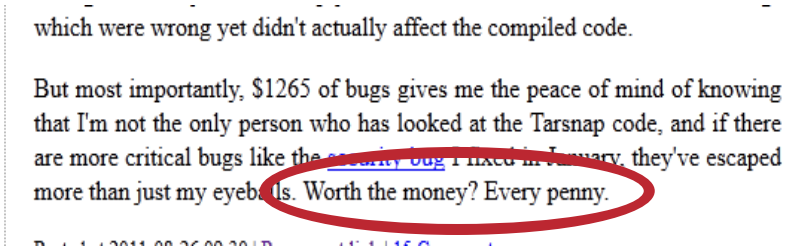
The move came less than a week after rival browser maker Mozilla increased Firefox bug bounties to $3,000.

In an entry to the Chromium project's blog, Chris Evans, who works on the Chrome security team, announced the new maximum bounty of $3.133.70 and said Google would "most likely" award that amount for all vulnerabilities rated "critical" in the company's four-step scoring system.

"The increased reward reflects the fact that the sandbox makes it harder to find bugs of this severity," said Evans, referring to the technology baked into Chrome that isolates processes from one another and the rest of the machine, preventing or at least hindering malicious code from escaping an application to wreak havoc or infect the computer.

- Tarsnap:
  - 125 spelling/style
    - 63 harmless
    - 11 minor
  - \+ 1 major

- 75/200 = 38% TP rate
- $17 + 40 hour per TP



which were wrong yet didn't actually affect the compiled code.

But most importantly, $1265 of bugs gives me the peace of mind of knowing that I'm not the only person who has looked at the Tarsnap code, and if there are more critical bugs like the [redacted] bug I fixed in January, they've escaped more than just my eyeballs. Worth the money? Every penny.

Debugging

- Question 1: Bug Localization
  - How do we know where the bugs are?
- Question 2: Bug Fixing
  - How do we fix the bugs automatically?

GenProg: A Generic Method for Automatic Software Repair

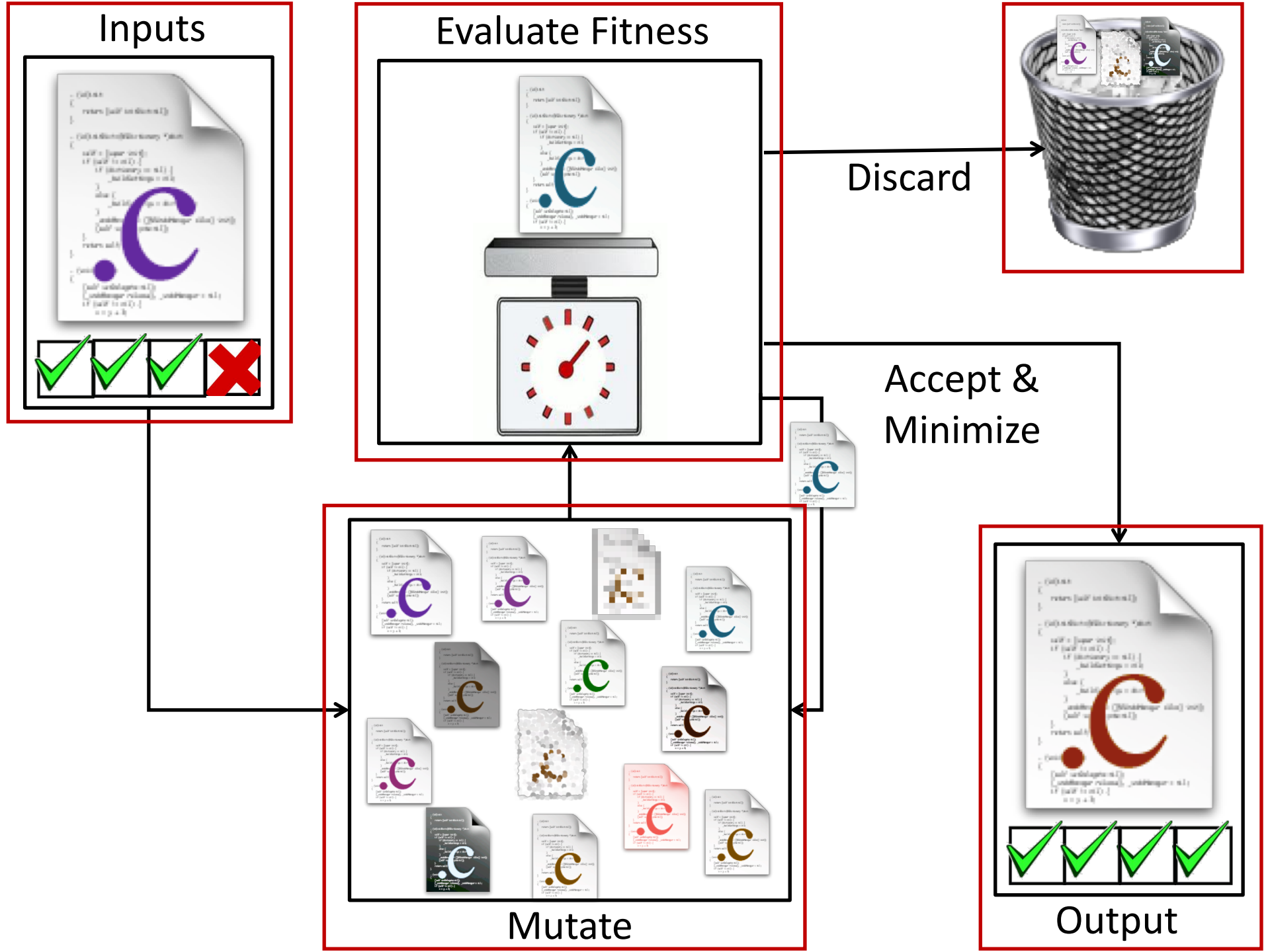
Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, Westley Weimer

TSE 2012, Citation: 454

Contributions

- GenProg, an **automatic** approach that uses Genetic Programming (GP) to generate patches for bugs in programs, as validated by test cases

- Experimental results showing that GenProg can efficiently repair errors in 16 C programs (a total of 1.25M lines of C code and 120K lines of module code)


Inputs
Evaluate Fitness
Discard
Accept &
Minimize
Mutate
Output

Bird's Eye View

- Approach: compose small random edits
  - Where to change?
  - How to change it?
- Search: random (GP) search through **nearby patches**

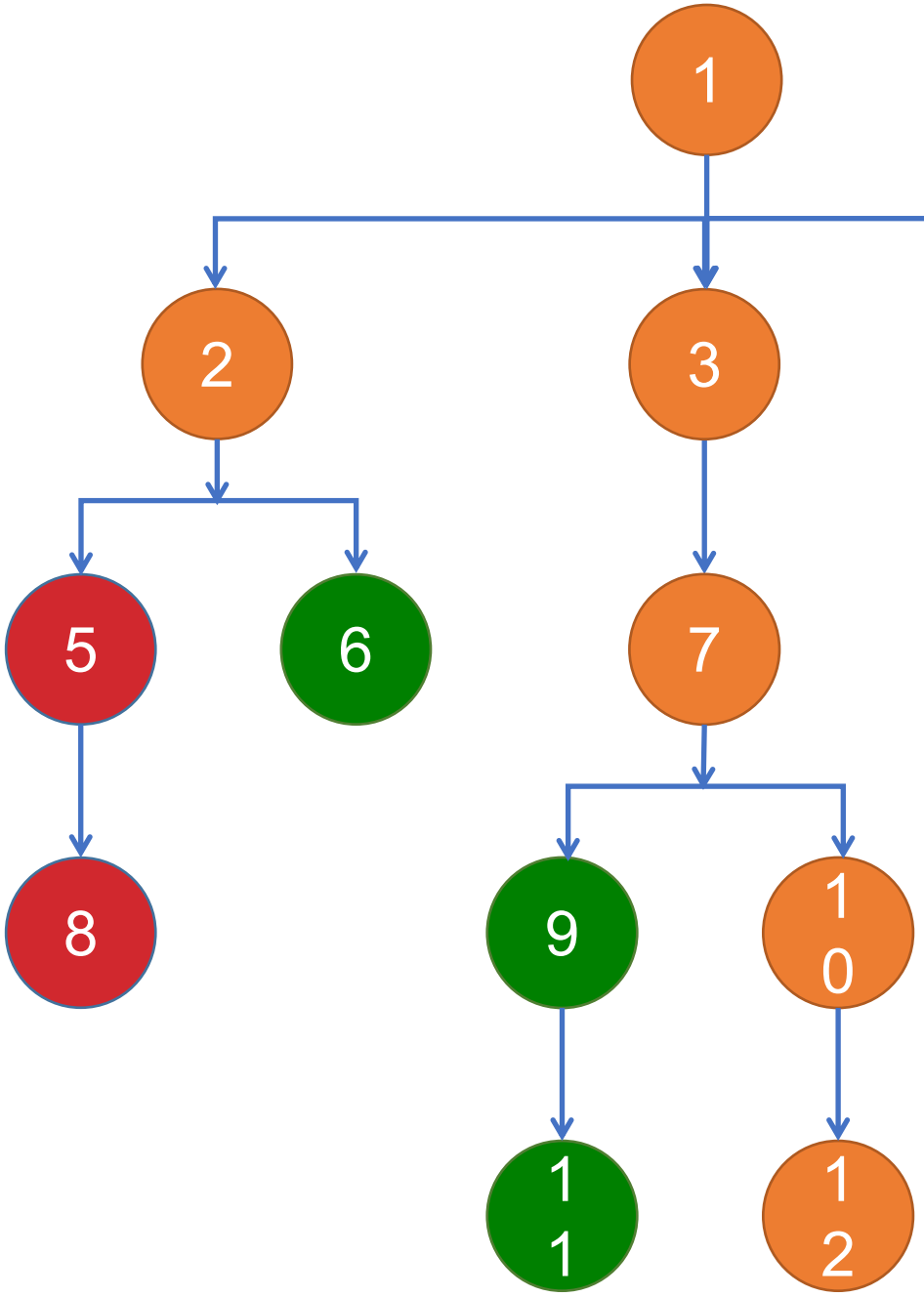
Input:

Input: 

Bug Localization!

Legend:

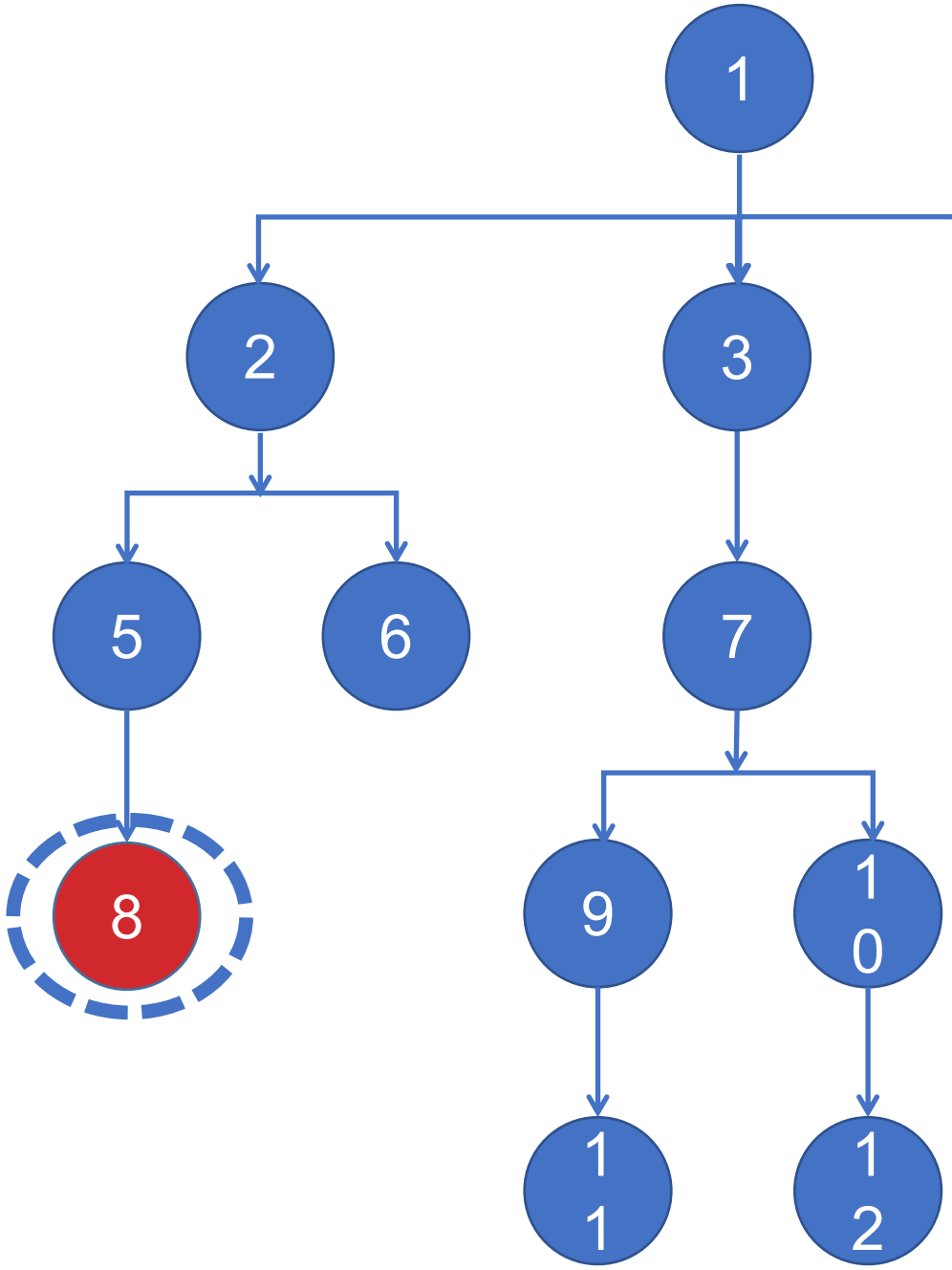
- **High change probability**
- **Low change probability**
- **Not changed**



An edit is:

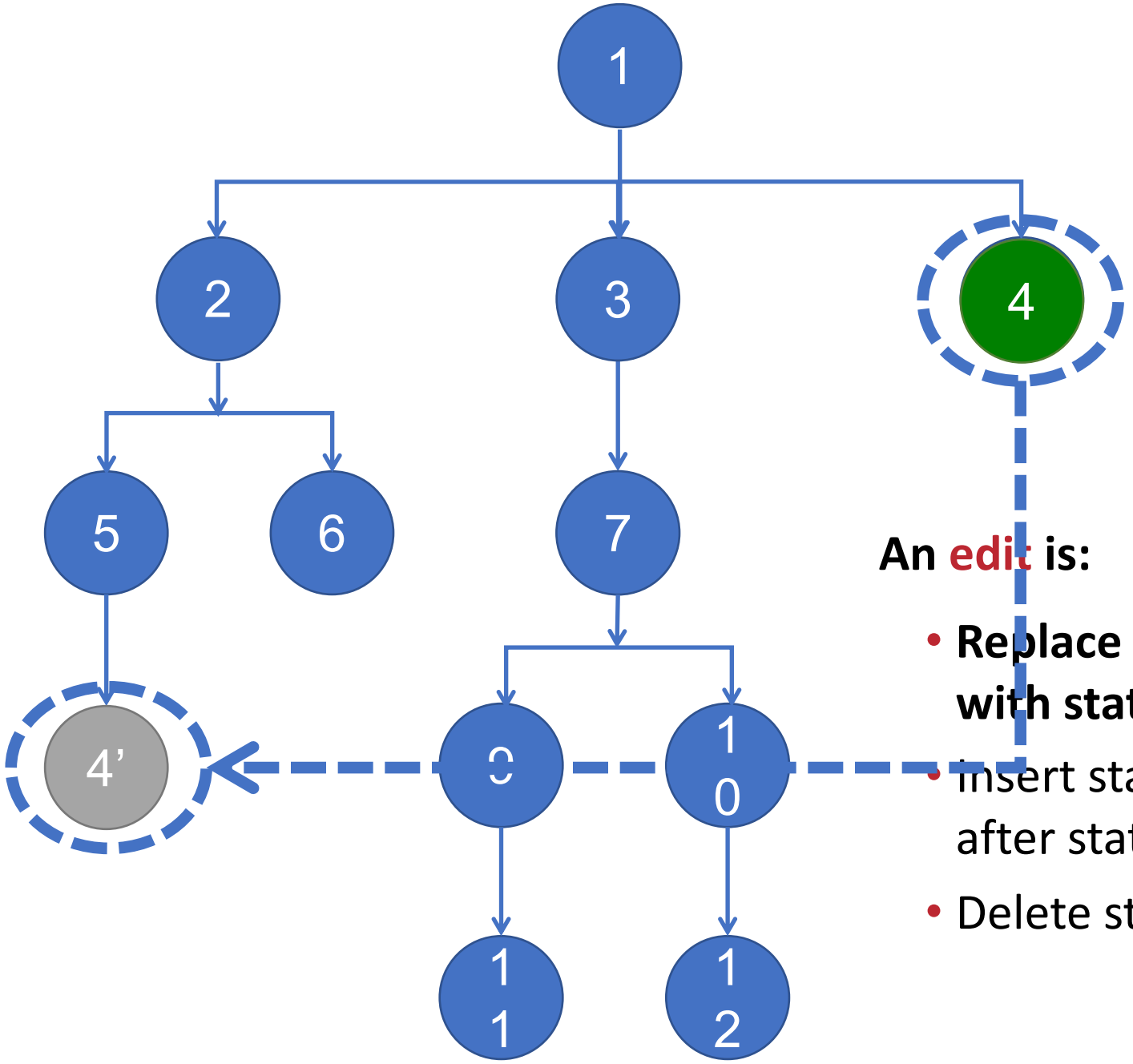
- Replace statement X with statement Y
- Insert statement X after statement Y
- Delete statement X

Y can be any statement in the program!



An edit is:

- **Replace statement X with statement Y**
- Insert statement X after statement Y
- Delete statement X



An edit is:

- **Replace statement X with statement Y**
- Insert statement X after statement Y
- Delete statement X

nullhttpd: Remote Heap Buffer Overflow

- nullhttpd trusts the Content-Length value provided by the user in the HTTP header of POST requests

```
108  // http.c
109  conn[sid].PostData=
110      calloc(conn[sid].dat->in_ContentLength+1024,
111             sizeof(char));
112  pPostData=conn[sid].PostData;
113  ...
114  do {
115    rc=recv(conn[sid].socket,
116            pPostData, 1024, 0); /* overflow! */
117    ...
118    pPostData+=rc;
119  } while ((rc==1024) ||
120            (x<conn[sid].dat->in_ContentLength));

267  // cgi.c
268  if (conn[sid].dat->in_ContentLength>0) {
269    write(local.out, conn[sid].PostData,
270          conn[sid].dat->in_ContentLength);
271  }
```

Benchmark Programs

Program	Lines of Code Total	Lines of Code Module	Description	Fault
`gcd`	22	22	example	infinite loop
`zune`	28	28	example [33]	infinite loop†
`uniq utx 4.3`	1146	1146	duplicate text processing	segmentation fault
`look utx 4.3`	1169	1169	dictionary lookup	segmentation fault
`look svr 4.0 1.1`	1363	1363	dictionary lookup	infinite loop
`units svr 4.0 1.1`	1504	1504	metric conversion	segmentation fault
`deroff utx 4.3`	2236	2236	document processing	segmentation fault
`nullhttpd 0.5.0`	5575	5575	webserver	remote heap buffer overflow (code)†
`openldap 2.2.4`	292598	6519	directory protocol	non-overflow denial of service†
`ccrypt 1.2`	7515	7515	encryption utility	segmentation fault†
`indent 1.9.1`	9906	9906	source code processing	infinite loop
`lighttpd 1.4.17`	51895	3829	webserver	remote heap buffer overflow (variables)†
`flex 2.5.4a`	18775	18775	lexical analyzer generator	segmentation fault
`atris 1.0.6`	21553	21553	graphical tetris game	local stack buffer exploit†
`php 4.4.5`	764489	5088	scripting language	integer overflow†
`wu-ftpd 2.6.0`	67029	35109	FTP server	format string vulnerability†
total	1246803	121337		

- 2-6 positive test cases and one negative test case
- perform 100 random trials for each program
  - percentage of trials that produce a repair
  - average time to the initial repair in a successful trial
  - time to minimize a final repair

Repair Results

Program	Initial Repair				Final Repair			
	Time	Fitness	Success	Size	Time	Fitness	Size	Effect
`gcd`	153 s	45.0	54%	21	4 s	4	2	Insert
`zune`	42 s	203.5	72%	11	1 s	2	3	Insert
`uniq`	34 s	15.5	100%	24	2 s	6	4	Delete
`look-u`	45 s	20.1	99%	24	3 s	10	11	Insert
`look-s`	55 s	13.5	100%	21	4 s	5	3	Insert
`units`	109 s	61.7	7%	23	2 s	6	4	Insert
`deroff`	131 s	28.6	97%	61	2 s	7	3	Delete
`nullhttpd`	578 s	95.1	36%	71	76 s	16	5	Both
`openldap`	665 s	10.6	100%	73	549 s	10	16	Delete
`ccrypt`	330 s	32.3	100%	34	13 s	10	14	Insert
`indent`	546 s	108.6	7%	221	13 s	13	2	Insert
`lighttpd`	394 s	28.8	100%	214	139 s	14	3	Delete
`flex`	230 s	39.4	5%	52	7 s	6	3	Delete
`atris`	80 s	20.2	82%	19	11 s	7	3	Delete
`php`	56 s	15.5	100%	139	94 s	11	10	Delete
`wu-ftpd`	2256 s	48.5	75%	64	300 s	6	5	Both
average	356.5 s	33.63	77.0%	67.0	76.3 s	8.23	5.7	

Scalability and Performance

Program	Fitness Total	Positive Tests	Negative Tests	Compile
`gcd`	91.2%	44.4%	46.8%	8.4%
`zune`	94.7%	23.2%	71.5%	3.7%
`uniq`	71.0%	17.2%	53.8%	25%
`look-u`	76.4%	17.1%	59.3%	20.7%
`look-s`	83.8%	29.9%	53.9%	12.9%
`units`	57.8%	20.7%	37.1%	35.5%
`deroff`	44.5%	8.9%	35.6%	46.5%
`nullhttpd`	74.9%	22.9%	52.0%	12.8%
`openldap`	93.7%	82.6%	11.1%	6.2%
`indent`	36.5%	16.4%	20.1%	43.1%
`ccrypt`	87.3%	65.2%	22.0%	4.7%
`lighttpd`	68.0%	67.9%	0.06%	25.3%
`flex`	23.2%	18.3%	4.8%	39.5%
`atris`	1.9%	0.8%	1.1%	64.9%
`php`	12.0%	2.0%	10.0%	78.4%
`wu-ftpd`	87.2%	38.6%	48.6%	6.6%
Average	62.75	29.76	32.99	27.14
StdDev	30.37	24.00	23.17	22.55

A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, Westley Weimer

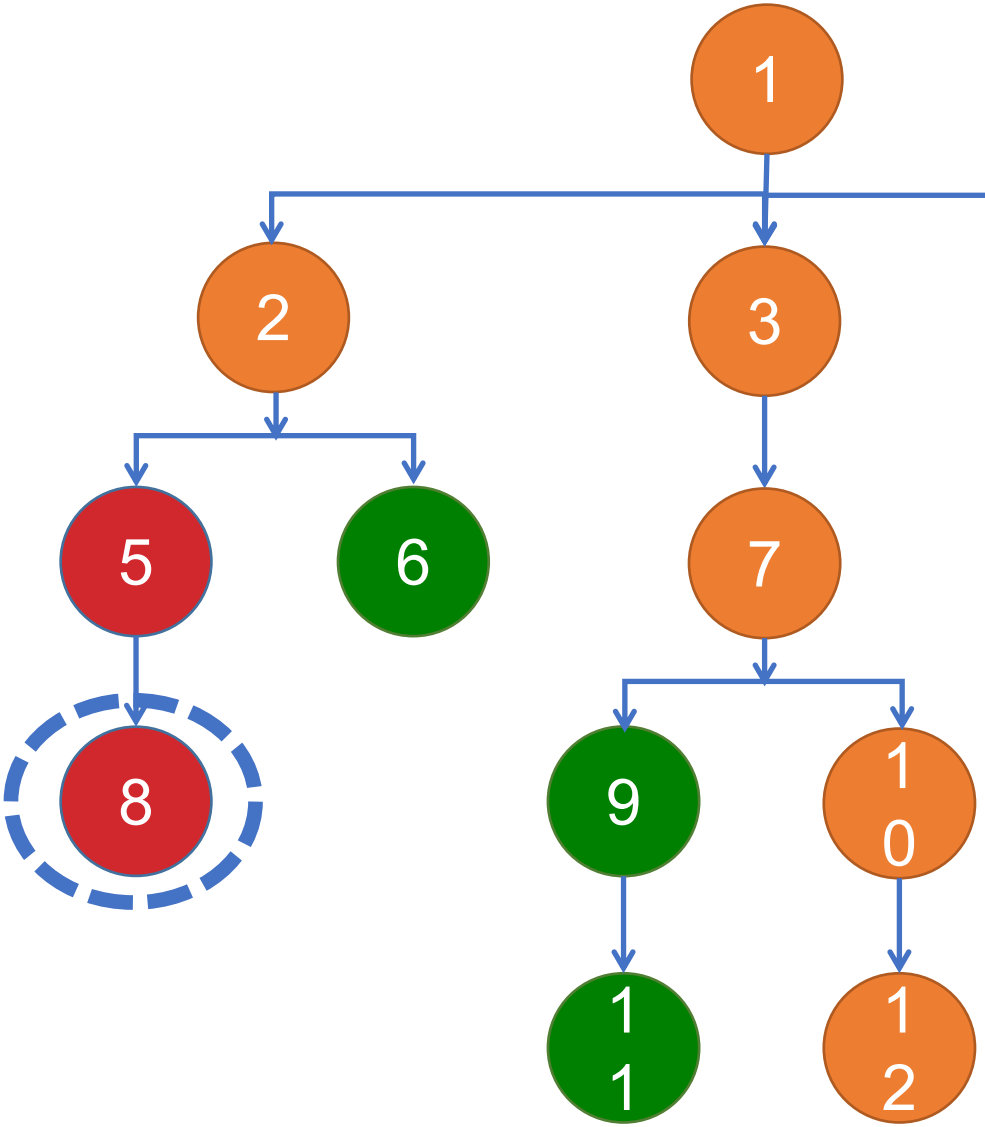
ICSE 2012, Citation: 349

Contribution

- GenProg, a **scalable** approach to automated program repair based on Genetic Programming

- A systematic evaluation of GenProg on 105 defects from 5.1 MLOC of open-source projects, conducted using cloud computing and virtualization

Scalable: Search Space

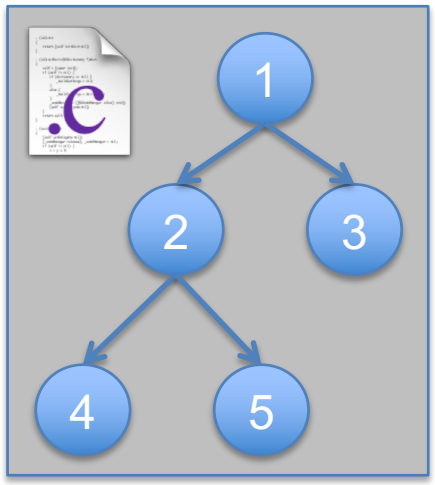


Fix localization: restrict inserted code

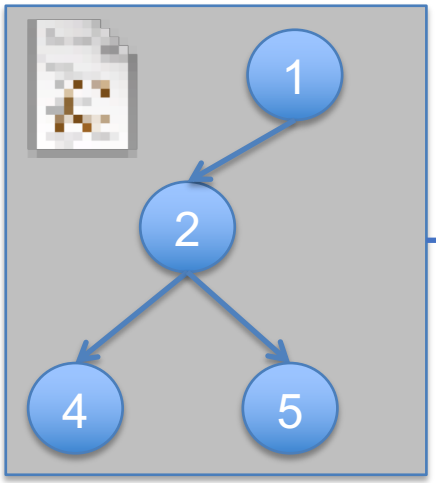
- include variables that are in-scope at the destination
- visited by at least one test case

Scalable: Representation

Input:

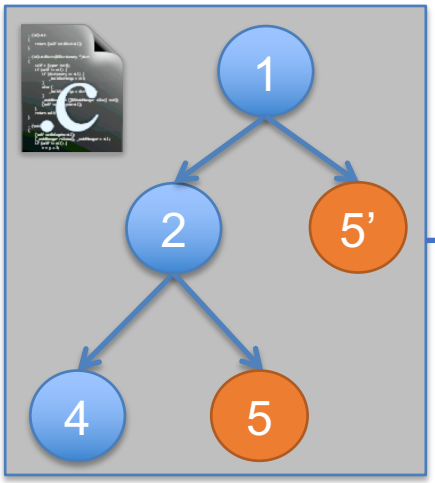


Naïve:



New:

Delete(3)



Replace(3,5)

Scalable: Parallelism

- Fitness evaluation
  - Subsample test cases
  - Evaluate in parallel
- Random runs
  - Multiple simultaneous runs on different seeds

Subject Programs and Defects

- 1st goal: a large set of **important**, **reproducible** bugs in **non-trivial** programs
- Approach: use historical revision data to approximate discovery and repair of bugs in the wild
  - Consider top programs from SourceForge, Google Code, Fedora SRPM, etc.
  - Look for revisions that caused the program to pass test cases that it failed in a previous revision

Subject Programs and Defects (cont.)

- 2nd goal: try to repair every bug in the benchmark set, establish **grounded cost measurements**
- Approach: use program that can run without modification under cloud computing virtualization

Subject Programs and Defects (cont.)

Program	LOC	Tests	Bugs	Description
fbc	97,000	773	3	Language (legacy)
gmp	145,000	146	2	Multiple precision math
gzip	491,000	12	5	Data compression
libtiff	77,000	78	24	Image manipulation
lighttpd	62,000	295	9	Web server
php	1,046,000	8,471	44	Language (web)
python	407,000	355	11	Language (general)
wireshark	2,814,000	63	7	Network packet analyzer
Total	**5,139,000**	**10,193**	**105**	

Success Rate and Cost

Program	Defects Repaired	Cost per non-repair		Cost per repair	
		Hours	US$	Hours	US$
fbc	1/3	8.52	5.56	6.52	4.08
gmp	1/2	9.93	6.61	1.60	0.44
gzip	1/5	5.11	3.04	1.41	0.30
libtiff	17/24	7.81	5.04	1.05	0.04
lighttpd	5/9	10.79	7.25	1.34	0.25
php	28/44	13.00	8.80	1.84	0.62
python	1/11	13.00	8.80	1.22	0.16
wireshark	1/7	13.00	8.80	1.23	0.17
Total	**55/105**	**11.22h**		**1.60h**	

$403 for all 105 trials, leading to 55 repairs; $7.32 per repaired bug

Public Comparison

- JBoss issue tracking: median 5.0, mean 15.3 hours[1]
- IBM: $25 per defect during coding, rising at build, Q&A, post-release, etc.[2]
- Tarsnap.com: $17, 40 hours per non-trivial repair[3]
- Bug bounty programs in general:
  - At least $500 for security-critical bugs
  - One of our php bugs has an associated security CVE

- [1]C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller, “How long will it take to fix this bug?” in *Workshop on Mining Software Repositories,* May 2007.
- [2] L. Williamson, “IBM Rational software analyzer: Beyond source code,” in *Rational Software Developer Conference,* Jun. 2008.
- [3]http://www.tarsnap.com/bugbounty.html

Effectiveness of Patch Representation

Program	Fault	LOC	Repair Ratio
gcd	infinite loop	22	1.07
uniq-utx	segfault	1146	1.01
look-utx	segfault	1169	1.00
look-svr	infinite loop	1363	1.00
units-svr	segfault	1504	3.13
deroff-utx	segfault	2236	1.22
nullhttpd	buffer exploit	5575	1.95
indent	infinite loop	9906	1.70
flex	segfault	18775	3.75
atris	buffer exploit	21553	0.97
Average		**6325**	**1.68**

Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results

Westley Weimer, Zachary P. Fry, Stephanie Forrest

ASE 2013, Citation: 137

The Problem

- The (test execution) cost model of GenProg

$$\begin{aligned} \text{Cost} &= \text{Fault} \times \text{Fix}(\text{Fault}) \times \text{Suite}(\text{Fault}, \text{Fix}) \\ &\times \text{RepairStratCost}(\text{Fault}, \text{Fix}, \text{Suite}) \\ &\times \text{TestStratCost}(\text{Fault}, \text{Fix}, \text{Suite}, \text{RepairStratCost}) \end{aligned}$$

- Fault: the size of fault localization
- Fix: the number of possible mutations
- Suite: the number of test cases
- RepairStratCost: range from 1/(Fault * Fix) to 1
  - **Repair strategy**: the equivalence and order of candidate repairs
- TestStratCost: range from 1/Suite to 1
  - **Test strategy**: the order of test cases

Determining Semantic Equivalence

- Syntactic equality
  - Duplicate statements in the existing program yield duplicate insertions and thus duplicate repairs
- Dead code elimination
  - Dead code is inserted (liveness dataflow analysis can achieve this in polynomial time)
- Instruction scheduling
  - If two adjacent instructions reference no common resources (or if both references are reads), reordering them produces a semantically equivalent program

Adaptive Search Strategies

- Repair strategy
  - Use the suspiciousness value in fault localization
- Test strategy
  - Favor the test that has the highest historical chance of failure

Experimental Results

			Order 1 Search Space		Defects Repaired			Test Suite Evals.		US$ (2011)	
			AE	GP	AE	GP	GP	AE	GP	AE	GP
Program	LOC	Tests	$k = 1$	$k = 1$	$k = 1$	$k = 1$	$k \leq 10$	$k = 1$	$k \leq 10$	$k = 1$	$k \leq 10$
fbc	97,000	773	507	1568	1	0	1	1.7	1952.7	0.01	5.40
gmp	145,000	146	9090	40060	1	0	1	63.3	119.3	0.91	0.44
gzip	491,000	12	11741	98139	2	1	1	1.7	180.0	0.01	0.30
libtiff	77,000	78	18094	125328	17	13	17	3.0	28.5	0.03	0.03
lighttpd	62,000	295	15618	68856	4	3	5	11.1	60.9	0.03	0.04
php	1,046,000	8,471	26221	264999	22	18	28	1.1	12.5	0.14	0.39
python	407,000	355	—	—	2	1	1	—	—	–	—
wireshark	2,814,000	63	6663	53321	4	1	1	1.9	22.6	0.04	0.17
weighted sum	—	—	922,492	7,899,073	53	37	55	186.0	3252.7	4.40	14.78

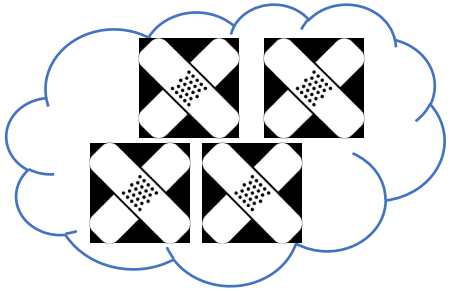
An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems

Zichao Qi, Fan Long, Sara Achour, Martin Rinard

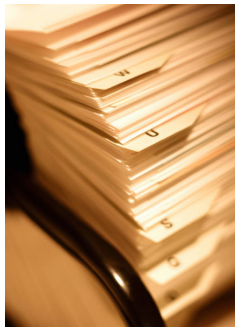
ISSTA 2015, Citation: 147

Generate-And-Validate Patch Generation

Buggy Program



Candidate Patch Space



Test Suite

Generate-And-Validate Patch Generation

- **GenProg** – Genetic Programming
  1. C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. **ICSE 2012**
  2. W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. **ICSE 2009**
  3. S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. **GECCO 2009**
  4. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. **TSE 38(1), 2012**
- **AE** – Adaptive Search
  1. W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. **ASE 2013**
- **RSRepair** – Random Search
  1. Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. **ICSE 2014**

All of Them Report Impressive Results

	GenProg	AE	RSRepair
Benchmark Defects	**105**	**105**	**24**
Reported Fixed Defects	**55**	**54**	**24**

- Patches generated by these systems are different from human written patch?
- No systematic analysis

Analyzing Their Reported Patches

Plausible?
Produce correct outputs for all test cases in the test suite

All generated patches should be plausible

Plausibility

	GenProg	AE	RSRepair
Benchmark Defects	**105**	**105**	**24**
Reported Fixed Defects	**55**	**54**	**24**
Defects With Plausible Patches	**18**	**27**	**10**

- Reason – Weak Proxy
  - Patch evaluation does not check for correct output
  - php, libtiff – only check exit code but **not output**

Analyzing Their Reported Patches

Plausible?
Produce correct outputs for all test cases in the test suite

All generated patches should be plausible

Correct?
Eliminate the defect

Passing test suite != correctness

Correctness

	GenProg	AE	RSRepair
Benchmark Defects	**105**	**105**	**24**
Reported Fixed Defects	**55**	**54**	**24**
Defects With Plausible Patches	**18**	**27**	**10**
Defects With Correct Patches	**2**	**3**	**2**

Developed new test cases that expose defects for all plausible but incorrect patches

GenProg Statistics

2 Correct

16 Plausible but Incorrect

37 Implausible

Stronger Test Suites?

Will GenProg generate correct patches given new test cases that eliminate incorrect patches?

Fixed Test Scripts?

Will GenProg generate plausible patches given fixed patch evaluation scripts?

Analyzing Their Reported Patches

Plausible?
Produce correct outputs for all test cases in the test suite

All generated patches should be plausible

Correct?
Eliminate the defect

Passing test suite != correctness

Do stronger test suites help?

Rerun GenProg with fixed patch evaluation scripts and new test cases that eliminate incorrect patches

Reexecuting GenProg on Remaining 103 Defects

First Reexecution
Fixed patch evaluation
New test cases
Patches for 2 defects

Second Reexecution
2 additional test cases
Patches for 0 defects

Why?

- Developer patches are not in GenProg search space
- GenProg search space may not contain **any** correct patch for these 103 defects
- May need richer search space to generate correct patches

Examples of Correct GenProg Patch

Developer

```
-if (y < 1000) {
-    PyObject *accept = PyDict_GetItemString(moddict,
-                                            "accept2dy
-    if (accept != NULL) {
-        int acceptval =  PyObject_IsTrue(accept);
-        if (acceptval == -1)
-            return 0;
-        if (acceptval) {
-            if (0 <= y && y < 69)
-                y += 2000;
-            else if (69 <= y && y < 100)
-                y += 1900;
-            else {
-                PyErr_SetString(PyExc_ValueError,
-                                "year out of range");
-                return 0;
-            }
-            if (PyErr_WarnEx(PyExc_DeprecationWarning,
-                "Century info guessed for a 2-digit ye
-                return 0;
-        }
-    }
-    else
-        return 0;
-}
 p->tm_year = y - 1900;
 p->tm_mon--;
 p->tm_wday = (p->tm_wday + 1) % 7;
```

GenProg

```
- if (y < 1000) {
-   tmp___0 = PyDict_GetItemString(moddict, "accept2dyear");
-   accept = tmp___0;
-   if ((unsigned int )accept != (unsigned int )((void *)0)) {
-     tmp___1 = PyObject_IsTrue(accept);
-     acceptval = tmp___1;
-     if (acceptval == -1) {
-       return (0);
-     } else {

-     }

-     if (acceptval) {
-       if (0 <= y) {
-         if (y < 69) {
-           y += 2000;
-         } else {
-           goto _L;
-         }
-       } else {
-         _L: /* CIL Label */
-         if (69 <= y) {
-           if (y < 100) {
-             y += 1900;
-           } else {
-             PyErr_SetString(PyExc_ValueError,
                              "year out of range");
-             return (0);
-           }
-         } else {
-           PyErr_SetString(PyExc_ValueError,
                           "year out of range");
-           return (0);
-         }
-       }

-       tmp___2 = PyErr_WarnEx(PyExc_DeprecationWarning,
-                              "Century info guessed for a 2-digit year.", 1);
-       if (tmp___2 != 0) {
-         return (0);
-       } else {

-       }
-     } else {

-     }
-   } else {
-     return (0);
-   }
- } else {

- }
  p->tm_year = y - 1900;
  (p->tm_mon) --;
  p->tm_wday = (p->tm_wday + 1) % 7;
```

Examples of Correct GenProg Patch (cont.)

Developer

```
1   if (offset >= s1_len) {
2           php_error_docref(NULL TSRMLS
3           "The start position cannot e
4           RETURN_FALSE;
5   }
6
7   -if (len > s1_len - offset) {
8   -           len = s1_len - offset;
9   -}
10
11  cmp_len = (uint) (len ? len : MAX(s2
```

GenProg

```
1   if (offset >= (long )s1_len) {
2       php_error_docref0((char const    *)((void
3       "The start position cannot exceed initia
4       while (1) {
5         __z___1 = return_value;
6         __z___1->value.lval = 0L;
7         __z___1->type = (unsigned char)3;
8         break;
9       }
10      return;
11    } else {
12
13    }
14  - if (len > (long )s1_len - offset) {
15  -   len = (long )s1_len - offset;
16  - } else {
17
18  - }
19    if (len) {
20      tmp___1 = len;
21    } else {
22      if ((long )s2_len > (long )s1_len - offs
23        tmp___0 = (long )s2_len;
24      } else {
25        tmp___0 = (long )s1_len - offset;
26      }
27      tmp___1 = tmp___0;
28    }
29    cmp_len = (unsigned int )tmp___1;
```

Analyzing Their Reported Patches

Plausible?
Produce correct outputs for all test cases in the test suite

All generated patches should be plausible

Correct?
Eliminate the defect

Passing test suite != correctness

Do stronger test suites help?

Rerun GenProg with fixed patch evaluation scripts and new test cases that eliminate incorrect patches

Are patches equivalent to functionality deletion?

A single deletion or return insertion modification

Functionality Deletion

	GenProg	AE	RSRepair
Benchmark Defects	105	105	24
Reported Fixed Defects	55	54	24
Defects With Plausible Patches	18	27	10
Defects With Plausible Patches Equivalent to Single Function Deletion	14	22	8

A Common Scenario

- A negative test case exposes a defect
  - The defect is in a feature that is otherwise unexercised
  - The patch simply deletes the functionality
    - Introduces new security vulnerabilities
    - Disables critical functionality
- Weak test suites
  - May be appropriate for human developers
  - May not be appropriate for automatic patch generation

Motivation for KALI

If all these patches simply delete functionality

Why not build a patch generation system that ONLY deletes functionality?

(1) Redirect branch

(2) Insert return

(3) Remove statement

Experimental Results of Kali

	GenProg	AE	RSRepair	Kali
Benchmark Defects	**105**	**105**	**24**	**105**
Reported Fixed Defects	**55**	**54**	**24**	
Defects With Plausible Patches	**18**	**27**	**10**	**27**
Defects With Correct Patches	**2**	**3**	**2**	**3**

- Kali is as good as previous systems
  - Much simpler
- Can pinpoint the defective code
- Can provide insight into important defect characteristics

Path To Success

- Richer search spaces
- More efficient search algorithms
- Incorporate additional sources of information
  - Correct code from other applications
  - Learned characteristics of human patches
  - Learned invariants
  - Specifications

Promising Directions

- Learn invariant from correct execution
  - J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. SOSP 2009.
  - Patches security vulnerabilities in **9** of **10** defects
  - At least **4** patches are correct
- Solvers
  - F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. CSTVA 2014
  - H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. ICSE 2013

Promising Directions (cont.)

- Specifications
  - Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Andreas Zeller, and Bertrand Meyer. Automated Fixing of Programs with Contracts. TSE, 2014
  - Etienne Kneuss, Manos Koukoutos and Viktor Kuncak. Deductive Program Repair. CAV 2015
- Correctness evaluation
  - Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, Jifeng Xuan. Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset. ESE, 2017
- Code from another application
  - S. Sidiroglou, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by multi-application code transfer. PLDI 2015

Promising Directions (cont.)

	SPR Staged condition synthesis	**Prophet** Learn from successful patches
Benchmark Defects	**105**	**105**
Defects With Plausible Patches	**41**	**42**
Defects With Correct Patches	**11**	**15**

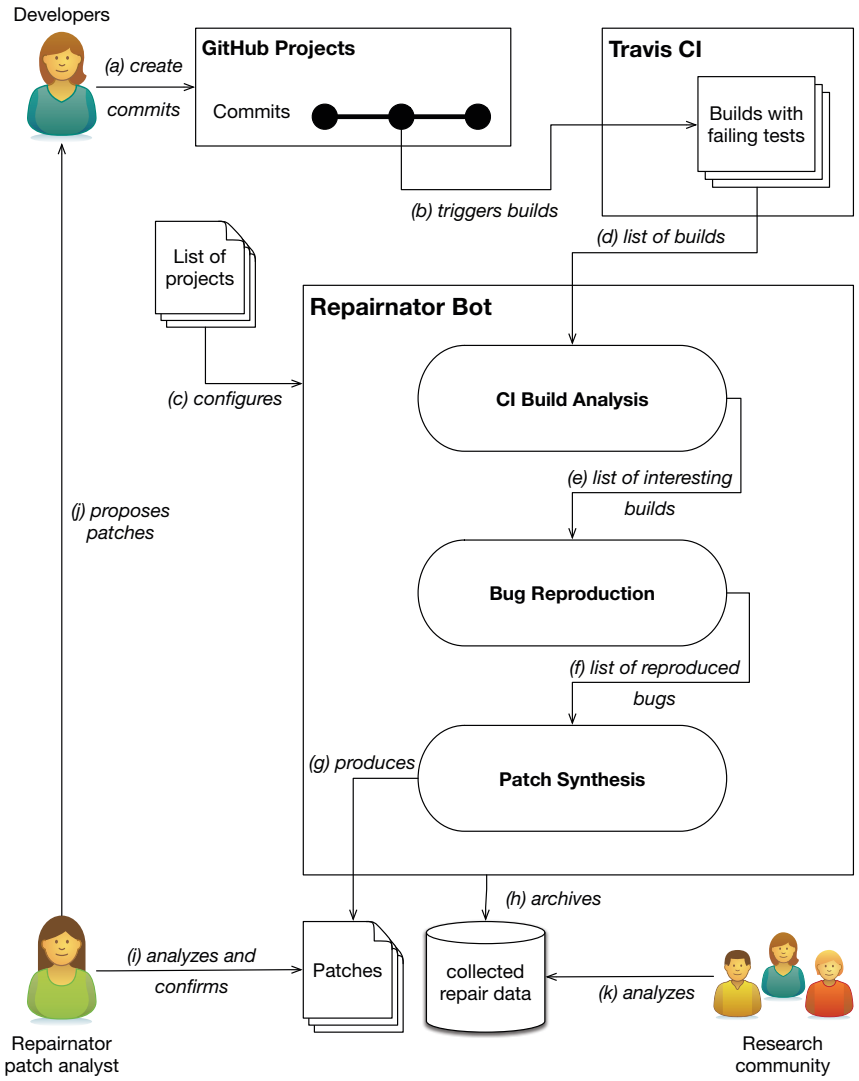
F. Long and M. Rinard. Staged program repair in SPR. ESEC/FSE 2015
F. Long and M. Rinard. Prophet: Automatic patch generation via learning from successful human patches. POPL 2016

How to Design a Program Repair Bot? Insights from the Repairnator Project

Simon Urli, Zhongxing Yu, Lionel Seinturier, Martin Monperrus

ICSE-SEIP 2018

Workflow of Program Repair Bot



Preliminary Results

- Pilot Experiment: January 2017
  - Validate the design and initial implementation
  - Capable of performing around 30 repair attempts per day
- Expedition #1: February – December 2017
  - Analyze 11,523 builds with test failures from 14,188 projects
  - Reproduce 3,551 test failures and find 15 patches
  - Not human-competitive (either too late or of low quality)
- Expedition #2: January – June 2018
  - Produce 5 human-competitive patches (accepted and merged by human developers)
  - **https://github.com/Spirals-Team/repairnator/issues/758**

Reading Materials

- C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. ICSE 2012
- W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. ICSE 2009
- S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. GECCO 2009
- C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. TSE 38(1), 2012
- W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. ASE 2013
- Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. ICSE 2014
- J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. SOSP 2009
- H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. ICSE 2013
- Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Andreas Zeller, and Bertrand Meyer. Automated Fixing of Programs with Contracts. TSE, 2014
- Etienne Kneuss, Manos Koukoutos and Viktor Kuncak. Deductive Program Repair. CAV 2015
- S. Sidiroglou, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by multi-application code transfer. PLDI 2015
- F. Long and M. Rinard. Staged program repair in SPR. ESEC/FSE 2015
- F. Long and M. Rinard. Prophet: Automatic patch generation via learning from successful human patches. POPL 2016

Q&A?

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

https://chenbihuan.github.io