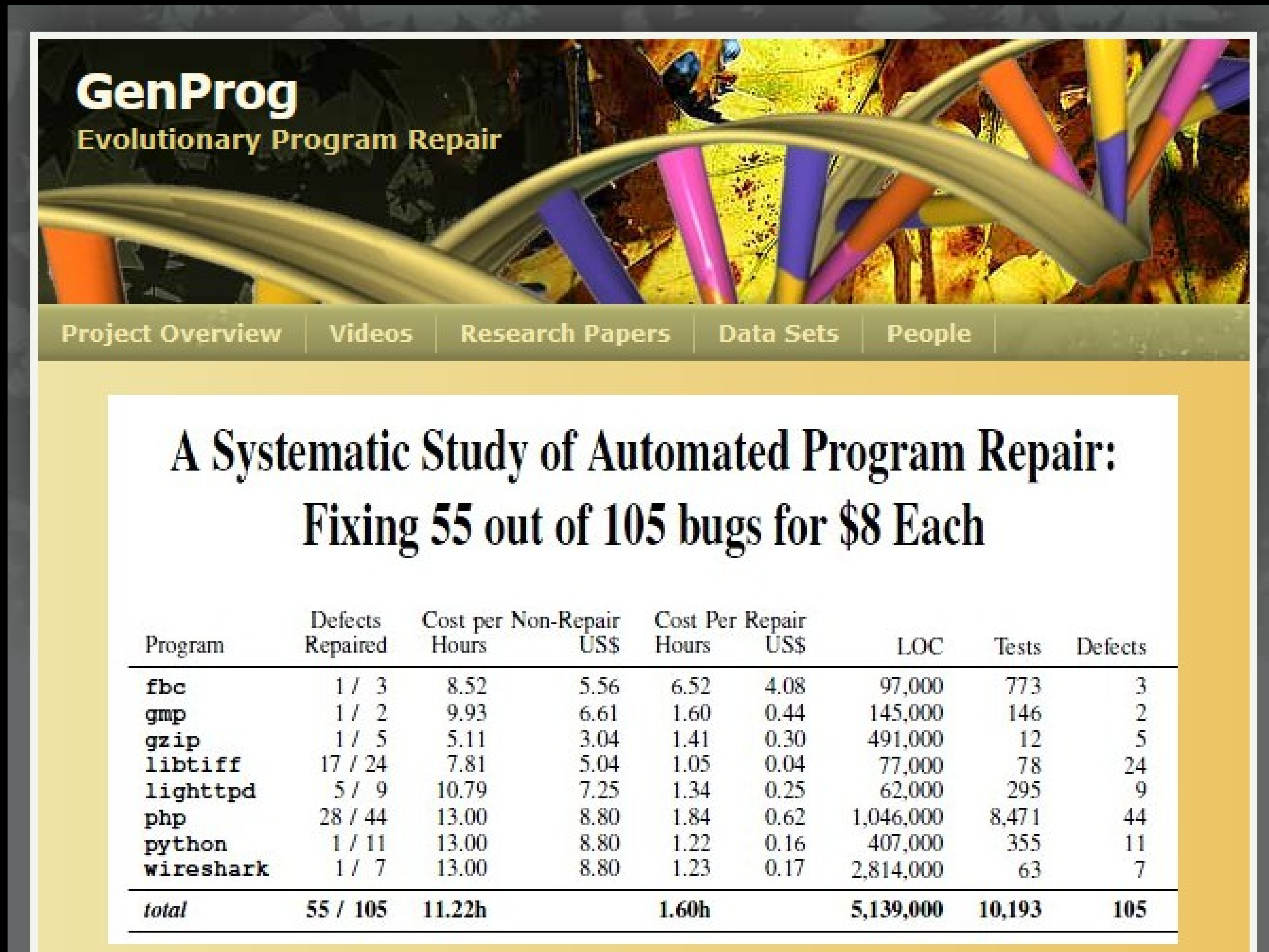


A Systematic Study of Automated Program Repair: Fixing 55 out of 105 bugs for $8 Each

Program	Defects Repaired	Cost per Non-Repair Hours	Cost per Non-Repair US$	Cost Per Repair Hours	Cost Per Repair US$	LOC	Tests	Defects
fbc	1 / 3	8.52	5.56	6.52	4.08	97,000	773	3
gmp	1 / 2	9.93	6.61	1.60	0.44	145,000	146	2
gzip	1 / 5	5.11	3.04	1.41	0.30	491,000	12	5
libtiff	17 / 24	7.81	5.04	1.05	0.04	77,000	78	24
lighttpd	5 / 9	10.79	7.25	1.34	0.25	62,000	295	9
php	28 / 44	13.00	8.80	1.84	0.62	1,046,000	8,471	44
python	1 / 11	13.00	8.80	1.22	0.16	407,000	355	11
wireshark	1 / 7	13.00	8.80	1.23	0.17	2,814,000	63	7
total	**55 / 105**	**11.22h**		**1.60h**		**5,139,000**	**10,193**	**105**

Automated Program Repair

Lecture Outline

- Automated Program Repair
- Historical Context, Recent Advances
- Mistakes
- Real-World Deployments

Speculative Fiction

What if large, trusted companies paid strangers to find and fix their normal and critical bugs?

Microsoft Security Response Center

HOME | WHAT WE DO | REPORT A VULNERABILITY | COMMUNITY COLLABO

Microsoft Security Bounty Programs

Print Email Share

..., hackers, researchers! Want to help us protect customers, making some of our most popular products better? And earn money doing so? Step right up...

Microsoft is now offering direct cash payments in exchange for reporting certain types of vulnerabilities and exploitation techniques.

In 2002, we pione... Computing initiative to ... to doing what we believe best helps improve our customers' computing experience. In the years since, we introduced the Security Development Lifecycle (SDL) process to build more secure technologies. We also championed Coordinated Vulnerability Disclosure (CVD), formed industry collaboration programs such as MAPP and MSVR, and created the BlueHat Prize to encourage research into defensive technologies. Our new bounty programs add fresh depth and flexibility to our existing community outreach programs. Having these bounty programs provides a way to harness the collective intelligence and capabilities of security researchers to help further protect customers.

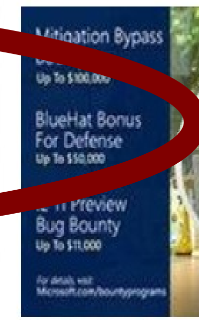
The following programs will launch on June 26, 2013:

1. **Mitigation Bypass Bounty.** Microsoft will pay up to $100,000 USD for truly novel exploitation techniques against protections built into the latest version of our operating system (Windows 8.1 Preview). Learning about new exploitation techniques earlier helps Microsoft improve security by leaps, instead of capturing one vulnerability at a time as a traditional bug bounty alone would. *TIMEFRAME: ONGOING*

2. **BlueHat Bonus for Defense.** Additionally, Microsoft will pay up to $50,000 USD for defensive ideas that accompany a qualifying Mitigation Bypass submission. Doing so highlights our continued support of defensive technologies and provides a way for the research community to help protect more than a billion computer systems worldwide. *TIMEFRAME: ONGOING (in conjunction with the Mitigation Bypass Bounty).*

3. **Internet Explorer 11 Preview Bug Bounty.** Microsoft will pay up to $11,000 USD for

Featured Vide



Trustworthy Comput Jonathan Ness, and introduce new boun researchers.

About the pro

Mitigation Bypass B for Defense Guidelin

Internet Explorer 11 Guidelines

Bounty Programs FA

New Bounty Progra information on bou

Heart of Blue Gold -

Microsoft Security Response Center

Search Microsoft.co

Microsoft Security Response Center

Personal Business Email forgot? Password forgot? Log In Sign Up

PayPal Buy Sell Transfer

For Security Researchers

Bug Bounty Wall of Fame

For Customers: Reporting Suspicious Emails

Customers who think they have received a Phishing email, please learn more about phishing at **https://cms.paypal.com/us/cgi-bin/marketingweb?cmd=_render-content&content_ID=security/hot_security_topics**, or forward it to: spoof@paypal.com

For Customers: Reporting All Other Concerns

Customers who have issues with their PayPal Account, please visit: **https://www.paypal.com/cgi-bin/helpscr?cmd=_help&t=escalateTab**

For Professional Researchers: Bug Bounty Program

Our team of dedicated security professionals works vigilantly to help keep customer information secure. We recognize the important role that security researchers and our user community play in also helping to keep PayPal and our customers secure. If you discover a site or product vulnerability please notify us using the guidelines below.

Program Terms

Please note that your participation in the Bug Bounty Program is voluntary and subject to the terms and conditions set forth on this page ("Program Terms"). By submitting a site or product vulnerability to PayPal, Inc. ("PayPal") you acknowledge that you have read and agreed to these Program Terms.

These Program Terms supplement the terms of PayPal User Agreement, the PayPal Acceptable Use Policy, and any other agreement in which you have entered with PayPal (collectively "PayPal Agreements"). The terms of those PayPal Agreements will apply to your use of, and participation in, the Bug Bounty Program as if fully set forth herein. If there is any inconsistency exists between the terms of the PayPal Agreements and these Program Terms, these Program Terms will control, but only with regard to the Bug Bounty Program.

You can jump to particular sections of these Program Terms by using the following links:

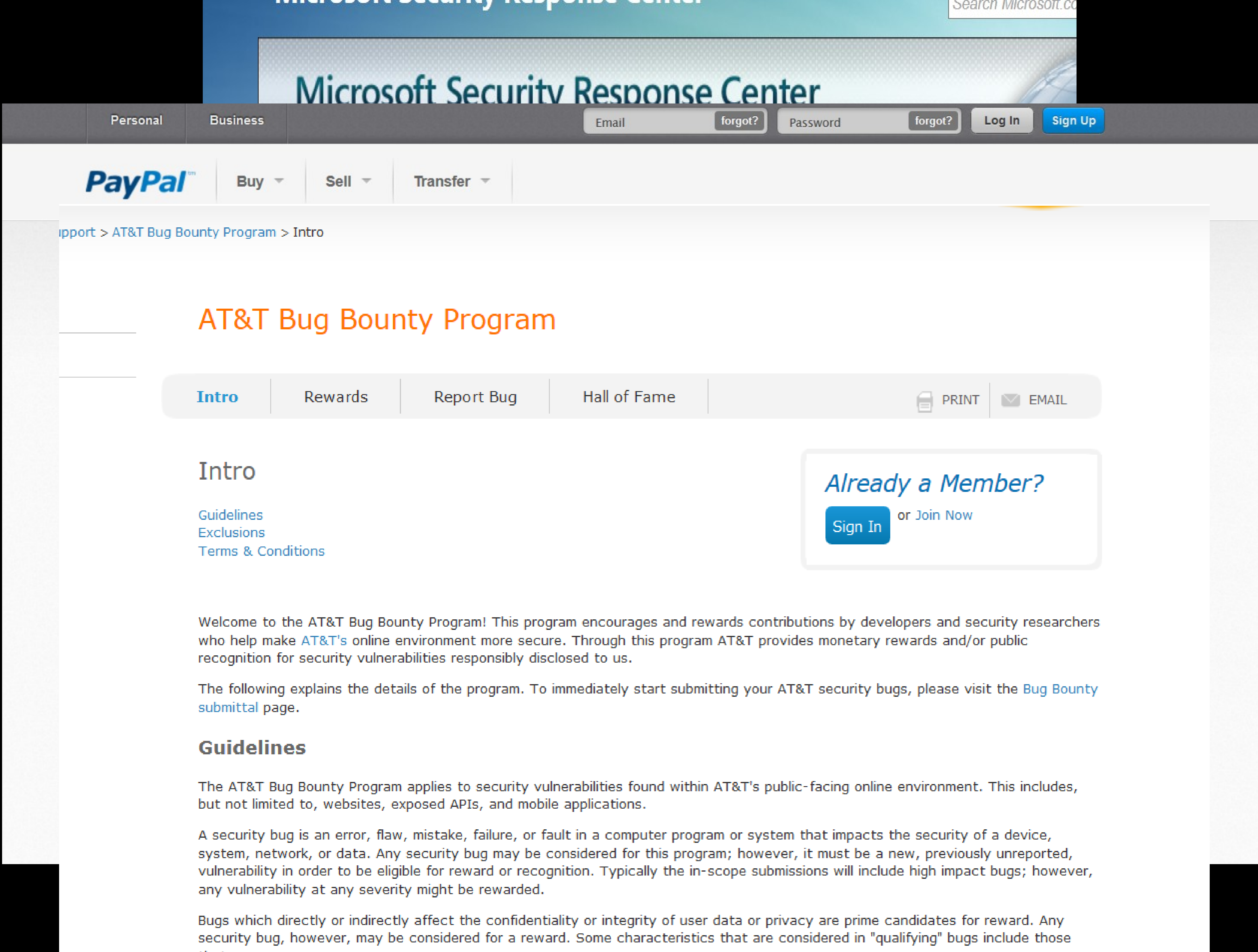
- **Responsible Disclosure Policy**
- **Eligibility Requirements**
- **Bug Submission Requirements and Guidelines**

research community to help protect more than a billion computer systems worldwide. *TIMEFRAME: ONGOING (in conjunction with the Mitigation Bypass Bounty).*

3. **Internet Explorer 11 Preview Bug Bounty.** Microsoft will pay up to $11,000 USD for

New Bounty Progra
information on bou

Heart of Blue Gold -


Microsoft Security Response Center
Search Microsoft.co
Microsoft Security Response Center
Personal
Business
Email
forgot?
Password
forgot?
Log In
Sign Up
PayPal
Buy
Sell
Transfer
ipport > AT&T Bug Bounty Program > Intro
AT&T Bug Bounty Program
Intro
Rewards
Report Bug
Hall of Fame
PRINT
EMAIL
Intro
Guidelines
Exclusions
Terms & Conditions
Already a Member?
Sign In
or Join Now
Welcome to the AT&T Bug Bounty Program! This program encourages and rewards contributions by developers and security researchers who help make AT&T's online environment more secure. Through this program AT&T provides monetary rewards and/or public recognition for security vulnerabilities responsibly disclosed to us.
The following explains the details of the program. To immediately start submitting your AT&T security bugs, please visit the Bug Bounty submittal page.
Guidelines
The AT&T Bug Bounty Program applies to security vulnerabilities found within AT&T's public-facing online environment. This includes, but not limited to, websites, exposed APIs, and mobile applications.
A security bug is an error, flaw, mistake, failure, or fault in a computer program or system that impacts the security of a device, system, network, or data. Any security bug may be considered for this program; however, it must be a new, previously unreported, vulnerability in order to be eligible for reward or recognition. Typically the in-scope submissions will include high impact bugs; however, any vulnerability at any severity might be rewarded.
Bugs which directly or indirectly affect the confidentiality or integrity of user data or privacy are prime candidates for reward. Any security bug, however, may be considered for a reward. Some characteristics that are considered in "qualifying" bugs include those

(Raise hand if true)

I have used software produced by Microsoft, PayPal, AT&T, Facebook, Mozilla, Google *or* YouTube.

Client Reward Guidelines

All bounty payments will be made in United States dollars (USD). You will be responsible for any tax implications related to bounty payments you receive, as
the laws of your jurisdiction of residence or citizenship.

Nevertheless, vulnerability reporters who work with us to resolve security bugs in our products will be credited on the Hall of Fame. If we file an inte
will acknowledge your contribution on that page.

Even though only 38% of the submissions were true positives (harmless, minor or major):

“Worth the money? Every penny.”

$20	$40	Build breakage on a platform where a previous Tarsnap release worked.
$10	~~$20~~	"Harmless" bugs, e.g., cosmetic errors in Tarsnap 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 code comments. Style errors in Tarsnap code qualify here, but usually not style errors in upstream code (e.g., libarchive).

If two or more people report the bug together the reward will be divided among them.

Client Reward Guidelines

All bounty payments will be made in United States dollars (USD). You will be responsible for any tax implications related to bounty payments you receive, as
the laws of your jurisdiction of residence or citizenship.

Nevertheless, vulnerability reporters who work with us to resolve security bugs in our products will be credited on the Hall of Fame. If we file an inte
will acknowledge your contribution on that page.

"We get hundreds of reports every day. Many of our best reports come from people whose English isn't great – though this can be challenging, it's something we work with just fine and we have paid out over $1 million to hundreds of reporters."

– Matt Jones, Facebook Software Engineer

$20	$40	Build breakage on a platform where a previous Tarsnap release worked.
$10	~~$20~~	"Harmless" bugs, e.g., cosmetic errors in Tarsnap 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 code comments. Style errors in Tarsnap code qualify here, but usually not style errors in upstream code (e.g., libarchive).

to our existing community outreach programs. Having these bounty programs provides a way to harness the collective intelligence and capabilities of security researchers to help further protect customers.

The following programs will launch on June 26, 2013:

1. **Mitigation Bypass Bounty.** Microsoft will pay up to $100,000 USD for truly novel exploitation techniques against protections built into the latest version of our operating system (Windows 8.1 Preview). Learning about new exploitation techniques earlier helps Microsoft improve security by leaps, instead of capturing one vulnerability at a time as a traditional bug bounty alone would. *TIMEFRAME: ONGOING*

2. **BlueHat Bonus for Defense.** Additionally, Microsoft will pay up to $50,000 USD for defensive ideas that accompany a qualifying Mitigation Bypass submission. Doing so highlights our continued support of defensive technologies and provides a way for the research community to help protect more than a billion computer systems worldwide. *TIMEFRAME: ONGOING (in conjunction with the Mitigation Bypass Bounty)*.

3. **Internet Explorer 11 Preview Bug Bounty.** Microsoft will pay up to $11,000 USD for critical vulnerabilities that affect Internet Explorer 11 Preview on the latest version of Windows (Windows 8.1 Preview). The entry period for this program will be the first 30 days of the Internet Explorer 11 beta period (June 26 to July 26, 2013). Learning about critical vulnerabilities in Internet Explorer as early as possible during the public preview will help Microsoft make the newest version of the browser more secure. *TIMEFRAME: 30 DAYS*

Want to know more?

A vision of the ~~future~~ present

Finding, fixing and ignoring bugs are all so expensive that it is **now** economical to pay untrusted strangers to submit candidate defect reports and patches.

A Modest Proposal

Automatically find and fix defects (rather than, or in addition to, paying strangers).

Outline

- Automated Program Repair
- The State of the Art
  - Scalability and Recent Growth
  - Recent GenProg Advances
- GenProg Lessons Learned (the fun part)
- Challenges & Opportunities

Historical Context

“We are moving to a new era where software systems are open, evolving and not owned by a single organization. Self-* systems are not just a nice new way to deal with software, but a necessity for the coming systems. The big new challenge of self-healing systems is to guarantee stability and convergence: we need to be able to master our systems even without knowing in advance what will happen to them.”

– Mauro Pezzè, Milano Bicocca / Lugano

Historical Context

- <= 1975 “Software fault tolerance”
  - Respond with minimal disruption to an unexpected software failure. Often uses isolation, mirrored fail-over, transaction logging, etc.
- ~1998: “Repairing one type of security bug”
  - [Cowan, Pu, Maier, Walpole, Bakke, Beattie, Grier, Wagle, Zhang, Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. USENIX Security 1998.]
- ~2002: “Self-healing (adaptive) systems”
  - Diversity, redundancy, system monitoring, models
  - [Garlan, Kramer, Wolf (eds). First Workshop on Self-Healing Systems, 2002.]

Why not just restart?

- Imagine two types of problems:
  - Non-deterministic (e.g., environmental): A network link goes down, send() raises an exception
  - Deterministic (e.g., algorithmic): The first line of main() dereferences a null pointer
- Failure-transparent or transactional approaches usually restart the same code
  - What if there is a deterministic bug in that code?

Checkpoint and Restart

[Lowell, Chandra, Chen: Exploring Failure Transparency and the Limits of Generic Recovery. OSDI 2000.]

Groundhog Day

[Lowell, Chandra, Chen: Exploring Failure Transparency and the Limits of Generic Recovery. OSDI 2000.]

Early “Proto” Program Repair Work

- 1999: Delta debugging [Zeller: Yesterday, My Program Worked. Today, It Does Not. Why? ESEC / FSE 1999.]
- 2001: Search-based software engineering [Harman, Jones. Search based software engineering. Information and Software Technology, 43(14) 2001]
- 2003: Data structure repair
    - Run-time approach based on constraints [Demsky, Rinard: Automatic detection and repair of errors in data structures. OOPSLA 2003.]
- 2006: Repairing safety policy violations
    - Static approach using formal FSM specifications [Weimer: Patches as better bug reports. GPCE 2006.]
- 2008: Genetic programming proposal [Arcuri: On the automation of fixing software bugs. ICSE Companion 2008.]

General Automated Program Repair

- Given a program …
  - Source code, assembly code, binary code
- … and evidence of a bug …
  - Passing and failing test cases, implicit specifications and crashes, preconditions and invariants, normal and anomalous runs
- … fix that bug.
  - A textual patch, a dynamic jump to new code, run-time modifications to variables

How could that work?

- Many faults can be localized to a small area
    - [Jones, Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. ASE 2005.]
    - [Qi, Mao, Lei, Wang. Using Automated Program Repair for Evaluating the Effectiveness of Fault Localization Techniques. ISSTA 2013.]
- Many defects can be fixed with small changes
    - [Park, Kim, Ray, Bae: An empirical study of supplementary bug fixes. MSR 2012.]
- Programs can be robust to such changes
    - "Only attackers and bugs care about unspecified, untested behavior."
    - [Schulte, Fry, Fast, Weimer, Forrest: Software Mutational Robustness. J. GPEM 2013.]

Scalability and Growth

2009: A Banner Year

GenProg

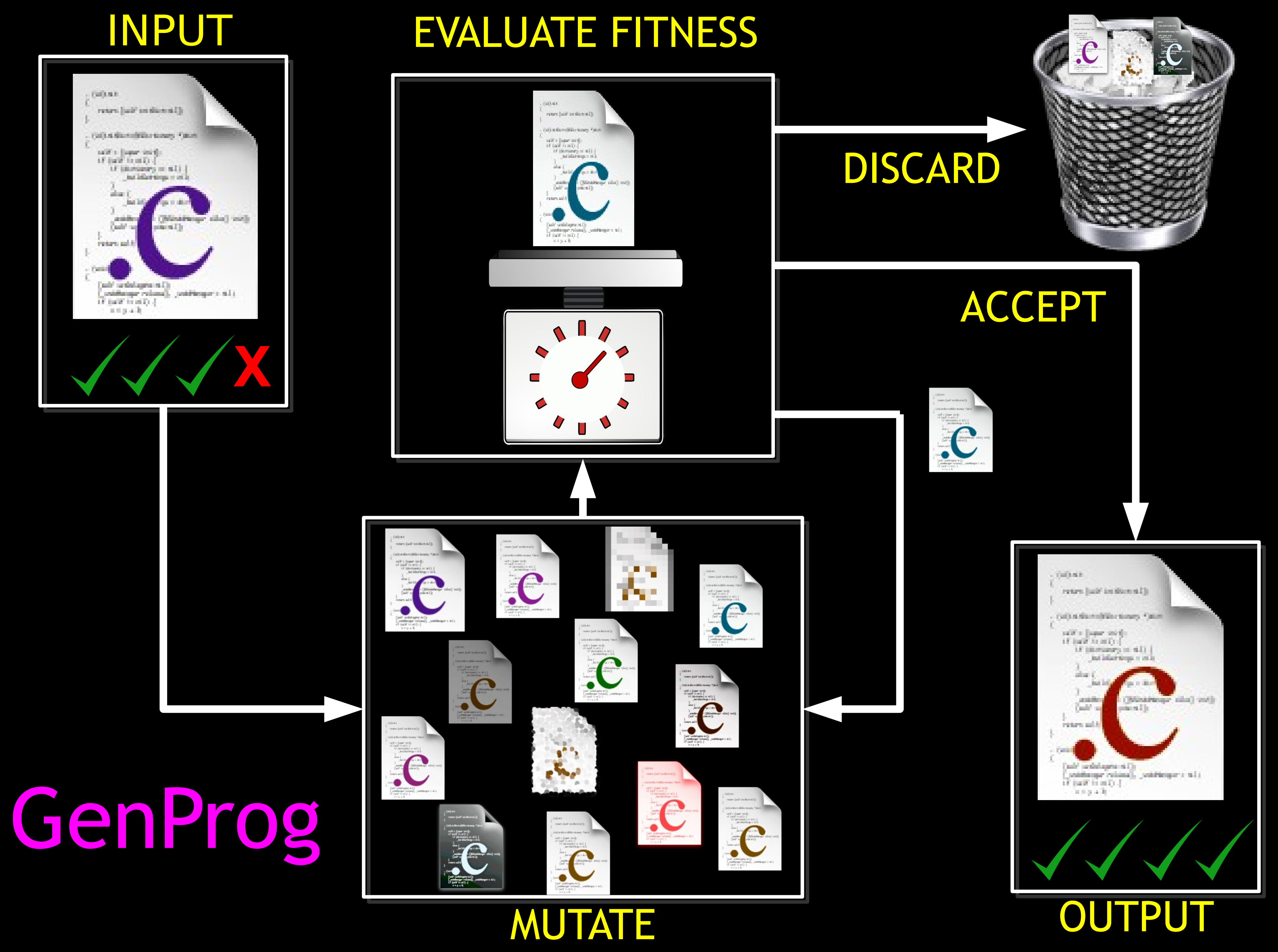
Genetic programming evolves source code until it passes the rest of a test suite. [Weimer, Nguyen, Le Goues, Forrest: Automatically finding patches using genetic programming. ICSE May 2009.]

ClearView

Detects normal workload invariants and anomalies, deploying binary repairs to restore invariants.
[Perkins, Kim, Larsen, Amarasinghe, Bachrach, Carbin, Pacheco, Sherwood, Sidiroglou, Sullivan, Wong, Zibin, Ernst, Rinard: Automatically patching errors in deployed software. SOSP Oct 2009.]

PACHIKA

Summarizes test executions to behavior models, generating fixes based on the differences. [Dallmeier, Zeller, Meyer: Generating Fixes from Object Behavior Anomalies. ASE Nov 2009.]


INPUT
EVALUATE FITNESS
DISCARD
ACCEPT
MUTATE
OUTPUT
GenProg

2009 In A Nutshell

- Given a program and tests (or a workload)
  - Normal observations: A B C *or* A B C D
- A problem is detected
  - Failing observations: A B X C
- The difference yields candidate repairs
  - { “Don't do X”, “Always do D” }
- One repair passes all tests
  - Report “Don't do X” as the patch

Two Broad Repair Approaches

- Single Repair or "Correct by Construction"
  - Careful consideration (constraint solving, invariant reasoning, lockset analysis, type systems, etc.) of the problem produces a single good repair.
- Generate-and-Validate
  - Various techniques (mutation, genetic programming, invariant reasoning, etc.) produce multiple candidate repairs.
  - Each candidate is evaluated and a valid repair is returned.

Name	Subjects	Tests	Bugs	Notes
AFix	2 Mloc	–	8	Concurrency, guarantees
ARC	–	–	–	Concurrency, SBSE
ARMOR	6 progs.	–	3 + –	Identifies workarounds
Axis	13 progs.	–	–	Concurrency, guarantees, Petri nets
AutoFix-E	21 Kloc	650	42	Contracts, guarantees
CASC	1 Kloc	–	5	Co-evolves tests and programs
ClearView	Firefox	57	9	Red Team quality evaluation
Coker Hafiz	15 Mloc	–	7 / –	Integer bugs only, guarantees
Debroy Wong	76 Kloc	22,500	135	Mutation, fault localization focus
Demsky *et al.*	3 progs.	–	–	Data struct consistency, Red Team
FINCH	13 tasks	–	–	Evolves unrestricted bytecode
GenProg	5 Mloc	10,000	105	Human-competitive, SBSE
Gopinath *et al.*	2 methods.	–	20	Heap specs, SAT
Jolt	5 progs.	–	8	Escape infinite loops at run-time
Juzi	7 progs.	–	20 + –	Data struct consistency, models
PACHIKA	110 Kloc	2,700	26	Differences in behavior models
PAR	480 Kloc	25,000	119	Human-based patches, quality study
SemFix	12 Kloc	250	90	Symex, constraints, synthesis
Sidiroglou *et al.*	17 progs.	–	17	Buffer overflows

Name	Subjects	Tests	Bugs	Notes
AFix	2 Mloc	–	8	Concurrency, guarantees
ARC	–	–	–	Concurrency, SBSE
ARMOR	6 progs.	–	3 + –	Identifies workarounds
Axis	13 progs.	–	–	Concurrency, guarantees, Petri nets
AutoFix-E	21 Kloc	650	42	Contracts, guarantees
CASC	1 Kloc	–	5	Co-evolves tests and programs
ClearView	Firefox	57	9	Red Team quality evaluation
Coker Hafiz	15 Mloc	–	7 / –	Integer bugs only, guarantees
Debroy Wong	76 Kloc	22,500	135	Mutation, fault localization focus
Demsky *et al.*	3 progs.	–	–	Data struct consistency, Red Team
FINCH	13 tasks	–	–	Evolves unrestricted bytecode
GenProg	5 Mloc	10,000	105	Human-competitive, SBSE
Gopinath *et al.*	2 methods.	–	20	Heap specs, SAT
Jolt	5 progs.	–	8	Escape infinite loops at run-time
Juzi	7 progs.	–	20 + –	Data struct consistency, models
PACHIKA	110 Kloc	2,700	26	Differences in behavior models
PAR	480 Kloc	25,000	119	Human-based patches, quality study
SemFix	12 Kloc	250	90	Symex, constraints, synthesis
Sidiroglou *et al.*	17 progs.	–	17	Buffer overflows

Name	Subjects	Tests	Bugs	Notes
AFix	2 Mloc	–	8	Concurrency, guarantees
ARC	–	–	–	Concurrency, SBSE
ARMOR	6 progs.	–	3 + –	Identifies workarounds
Axis	13 progs.	–	–	Concurrency, guarantees, Petri nets
AutoFix-E	21 Kloc	650	42	Contracts, guarantees
CASC	1 Kloc	–	5	Co-evolves tests and programs
ClearView	Firefox	57	9	Red Team quality evaluation
Coker Hafiz	15 Mloc	–	7 / –	Integer bugs only, guarantees
Debroy Wong	76 Kloc	22,500	135	Mutation, fault localization focus
Demsky *et al.*	3 progs.	–	–	Data struct consistency, Red Team
FINCH	13 tasks	–	–	Evolves unrestricted bytecode
GenProg	5 Mloc	10,000	105	Human-competitive, SBSE
Gopinath *et al.*	2 methods.	–	20	Heap specs, SAT
Jolt	5 progs.	–	8	Escape infinite loops at run-time
Juzi	7 progs.	–	20 + –	Data struct consistency, models
PACHIKA	110 Kloc	2,700	26	Differences in behavior models
PAR	480 Kloc	25,000	119	Human-based patches, quality study
SemFix	12 Kloc	250	90	Symex, constraints, synthesis
Sidiroglou *et al.*	17 progs.	–	17	Buffer overflows

Name	Subjects	Tests	Bugs	Notes
AFix	2 Mloc	–	8	Concurrency, guarantees
ARC	–	–	–	Concurrency, SBSE
ARMOR	6 progs.	–	3 + –	Identifies workarounds
Axis	13 progs.	–	–	Concurrency, guarantees, Petri nets
AutoFix-E	21 Kloc	650	42	Contracts, guarantees
CASC	1 Kloc	–	5	Co-evolves tests and programs
ClearView	Firefox	57	9	Red Team quality evaluation
Coker Hafiz	15 Mloc	–	7 / –	Integer bugs only, guarantees
Debroy Wong	76 Kloc	22,500	135	Mutation, fault localization focus
Demsky *et al.*	3 progs.	–	–	Data struct consistency, Red Team
FINCH	13 tasks	–	–	Evolves unrestricted bytecode
GenProg	5 Mloc	10,000	105	Human-competitive, SBSE
Gopinath *et al.*	2 methods.	–	20	Heap specs, SAT
Jolt	5 progs.	–	8	Escape infinite loops at run-time
Juzi	7 progs.	–	20 + –	Data struct consistency, models
PACHIKA	110 Kloc	2,700	26	Differences in behavior models
PAR	480 Kloc	25,000	119	Human-based patches, quality study
SemFix	12 Kloc	250	90	Symex, constraints, synthesis
Sidiroglou *et al.*	17 progs.	–	17	Buffer overflows

Name	Subjects	Tests	Bugs	Notes
AFix	2 Mloc	–	8	Concurrency, guarantees
ARC	–	–	–	Concurrency, SBSE
ARMOR	6 progs.	–	3 + –	Identifies workarounds
Axis	13 progs.	–	–	Concurrency, guarantees, Petri nets
AutoFix-E	21 Kloc	650	42	Contracts, guarantees
CASC	1 Kloc	–	5	Co-evolves tests and programs
ClearView	Firefox	57	9	Red Team quality evaluation
Coker Hafiz	15 Mloc	–	7 / –	Integer bugs only, guarantees
Debroy Wong	76 Kloc	22,500	135	Mutation, fault localization focus
Demsky *et al.*	3 progs.	–	–	Data struct consistency, Red Team
FINCH	13 tasks	–	–	Evolves unrestricted bytecode
GenProg	5 Mloc	10,000	105	Human-competitive, SBSE
Gopinath *et al.*	2 methods.	–	20	Heap specs, SAT
Jolt	5 progs.	–	8	Escape infinite loops at run-time
Juzi	7 progs.	–	20 + –	Data struct consistency, models
PACHIKA	110 Kloc	2,700	26	Differences in behavior models
PAR	480 Kloc	25,000	119	Human-based patches, quality study
SemFix	12 Kloc	250	90	Symex, constraints, synthesis
Sidiroglou *et al.*	17 progs.	–	17	Buffer overflows

State of the Art Woes

- GenProg uses test case results for guidance
  - But ~99% of candidates have identical test results
- Sampling tests improves GenProg performance
  - But GenProg cost models do not account for it
- Not all tests are equally important
  - But we could not learn a better weighting

Desired Solution

- Informative Cost Model
  - Captures observed behavior
- Efficient Algorithm
  - Exploits redundancy
- Theoretical Relationships
  - Explain potential successes

New Since The Papers You've Read

- Informative Cost Model
  - Highlights “two searches”, “redundancy”
- Efficient Algorithm
  - Exploits cost model, “adaptive equality”
- Theoretical Relationships
  - Duality with mutation testing

Cost Model

- GenProg at a high level:
  - "Pick a fault-y spot in the program, insert a fix-y statement there."
  - Dominating factor: cost of running tests.
- Search space of repairs = |Fault| x |Fix|
  - |Fix| can depend on |Fault|
    - Can only insert "x=1" if "x" is in scope, etc.
- Each repair must be validated, however
  - Run against |Suite| test cases
    - |Suite| can depend on repair (impact analysis, etc.)

Cost Model Insights

- Suppose there are five candidate repairs.
  - Can stop when a valid repair is found.
  - Suppose three are invalid and two are valid:

$CR_1 \quad CR_2 \quad CR_3 \quad CR_4 \quad CR_5$

- The **order** of repair consideration matters.
  - Worst case: |Fault| x |Fix| x |Suite| x (4/5)
  - Best case: |Fault| x |Fix| x |Suite| x (1/5)
- Let |R-Order| represent this cost factor

Cost Model Insights (2)

- Suppose we have a candidate repair.
  - If it is valid, we must run all |Suite| tests.
  - If it is invalid, it fails at least one test.
  - Suppose there are four tests and it fails one:

$T_1 \quad T_2 \quad T_3 \quad T_4$

- The **order** of test consideration matters:
  - Best case: |Fault| x |Fix| x |Suite| x (1/4)
  - Worst case: |Fault| x |Fix| x |Suite| x (4/4)
- Let |T-Order| represent this cost factor.

Cost Model

|Fault| x |Fix| x |Suite| x |R-Order| x |T-Order|

- Fault localization
- Fix localization
- Size of validating test Suite
- Order (Strategy) for considering Repairs
- Order (Strategy) for considering Tests
  - Each factor depends on all previous factors.

Induced Algorithm

- The cost model induces a direct nested search algorithm:

For every repair, in order

    For every test, in order

        Run the repair on the test

        Stop inner loop early if a test fails

    Stop outer loop early if a repair validates

Induced Algorithm

- The cost model induces a direct nested search algorithm:

Order can vary *adaptively* based on observations.

For every repair, in order

  For every test, in order

    Run the repair on the test

    Stop inner loop early if a test fails

  Stop outer loop early if a repair validates

Algorithm: Can We Avoid Testing?

- If P1 and P2 are semantically equivalent they must have the same test case behavior.

Algorithm: Can We Avoid Testing?

- If P1 and P2 are semantically equivalent they must have the same test case behavior.
- Consider this insertion:

```
C=99;
```

Algorithm: Can We Avoid Testing?

- If P1 and P2 are semantically equivalent they must have the same test case behavior.
- Consider this insertion:

```
C=99;       A=1;
            B=2;
            C=3;
            D=4;
            print A,B,C,D
```

Algorithm: Can We Avoid Testing?

- If P1 and P2 are semantically equivalent they must have the same test case behavior.
- Consider this insertion:

C=99;

```
A=1;
B=2;
C=3;
D=4;
print A,B,C,D
```

Algorithm: Can We Avoid Testing?

- If P1 and P2 are semantically equivalent they must have the same test case behavior.
- Consider this insertion:

C=99;

```
A=1;
B=2;
C=3;
D=4;
print A,B,C,D
```

Formal Equality Idea

- Quotient the space of possible patches with respect to a conservative approximation of program equivalence
  - Conservative: P ≈ Q implies P is equivalent to Q
  - "Quotient" means "make equivalence classes"
- Only test one representative of each class
- Wins if computing P ≈ Q is cheaper than tests
  - Oh audience, how might we ***decide*** this?
  - Formal semantics (dead code, instruction sched.)

Adaptive Equality Algorithm

For every repair, ordered by observations

Adaptive Equality Algorithm

For every repair, ordered by observations

  Skip repair if equivalent to older repair

Adaptive Equality Algorithm

For every repair, ordered by observations

Skip repair if equivalent to older repair

For every test, ordered by observations

Adaptive Equality Algorithm

For every repair, ordered by observations

  Skip repair if equivalent to older repair

For every test, ordered by observations

  Run the repair on the test, update obs.

Adaptive Equality Algorithm

For every repair, ordered by observations
- Skip repair if equivalent to older repair

For every test, ordered by observations
- Run the repair on the test, update obs.
- Stop inner loop early if a test fails

Adaptive Equality Algorithm

For every repair, ordered by observations

Skip repair if equivalent to older repair

For every test, ordered by observations

Run the repair on the test, update obs.

Stop inner loop early if a test fails

Stop outer loop early if a repair validates

Adaptive Equality Algorithm

Test Cases or Invariants +
Bug Example +
Fault Localization +
Formal Semantics +
AST Substitutions +
Machine Learning
=
Automated Program Repair

op outer loop early if a repair validates

Theoretical Relationship

- The generate-and-validate program repair problem is a dual of mutation testing
  - This suggests avenues for cross-fertilization and helps explain some of the successes and failures of program repair.

- Very informally:
  - PR   Exists M in Mut. Forall T in Tests.   M(T)
  - MT   Forall M in Mut. Exists T in Tests. Not M(T)

Idealized Formulation

Ideally, mutation testing takes a program that **passes** its test suite and requires that **all** mutants based on human **mistakes** from the **entire** program that are not equivalent **fail** at least one test.

By contrast, program repair takes a program that **fails** its test suite and requires that **one** mutant based on human **repairs** from the fault **localization** only be found that **passes** all tests.

Idealized Formulation

Ideally, mutation testing takes a program that **passes** its test suite and requires that **all** mutants based on human **mistakes** from the **entire** program that are not equivalent **fail** at least one test.

By c... ...gram r...

pas...

For mutation testing, the Equivalent Mutant Problem is an issue of *correctness* (or the adequacy score is not meaningful).

For program repair, it is purely an issue of *performance*.

GenProg Improvement Results

- Evaluated on 105 defects in 5 MLOC guarded by over 10,000 tests
- Adaptive Equality reduces GenProg's test case evaluations by **10x** and monetary cost by **3x**
  - Adaptive T-Order is within 6% of optimal
  - “GenProg – GP ≥ GenProg” ?
- Cost Model (expressive)
- Efficient Algorithm (adaptive equality)
- Theoretical Relationships (mutation testing)

State of the Art

- 2009: 15 papers on auto program repair
  - (Manual search/review of ACM Digital Library)
- 2011: Dagstuhl on Self-Repairing Programs
- 2012: 30 papers on auto program repair
  - At least 20+ different approaches, 3+ best paper awards, etc.
- 2013: ICSE has a “Program Repair” session
- So now let's talk about the seamy underbelly.

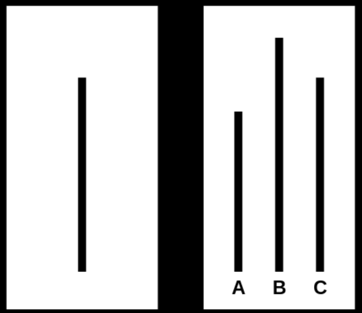
Computer Scientists

- Often dubbed "the first programmer", this English mathematician is known for work involving the early general-purpose computer known as the Analytical Engine. The first such published algorithm (lecture notes for an 1842 seminar at Turin) was designed to compute Bernoulli Numbers:

  B0 = 1, B1 = ±1/2, B2 = 1/6, B3 = 0, B4 = −1/30, B5 = 0, B6 = 1/42, B7 = 0, B8 = −1/30, etc.

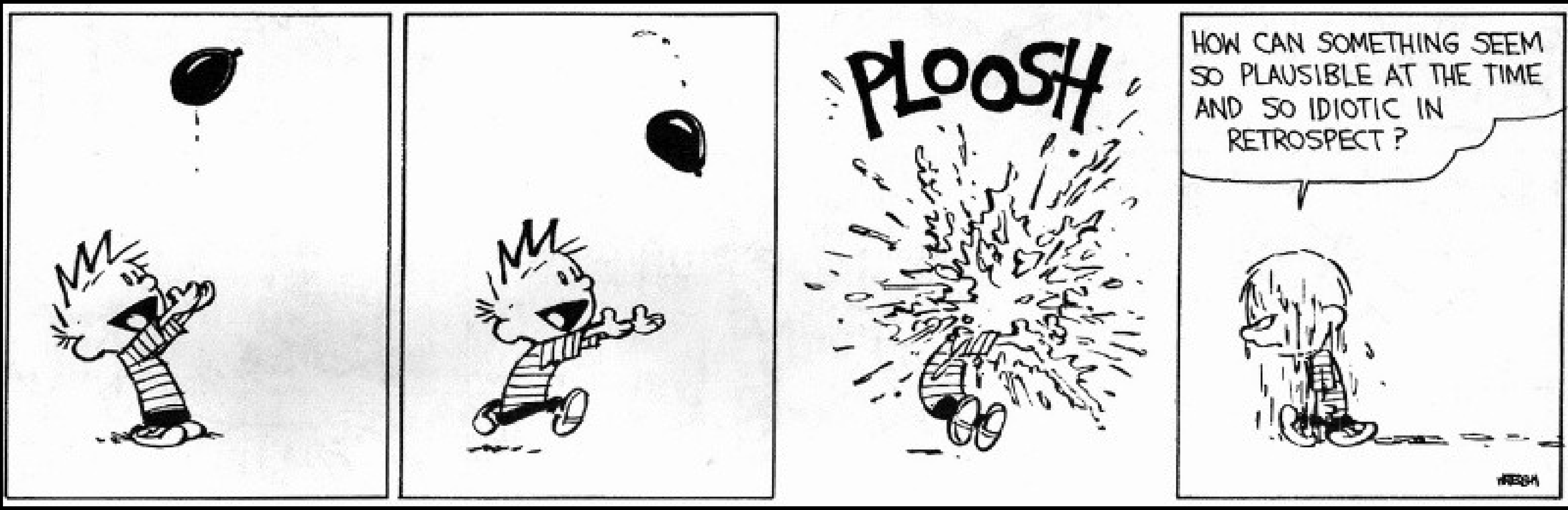
"[The Analytical Engine] might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine..."

Social Psychology



- Each participant was placed with seven "confederates". Participants were shown a card with a line on it, followed by a card with three lines on it. Participants were then asked to say aloud which line matched first line in length. Confederates unanimously gave the correct response or unanimously gave the incorrect response. For the first two trials the confederates gave the obvious, correct answer. On the third trial, the confederates would all give the same wrong answer, placing the participant in a dilemma.
- In the control group, with no pressure to conform to confederates, the error rate was less than 1%. An examination of all critical trials in the experimental group revealed that one-third of all responses were incorrect. These incorrect responses often matched the incorrect response of the majority group (i.e., confederates). Overall, in the experimental group, 75% of the participants gave an incorrect answer to at least one question.

Lessons Learned



Lessons Learned: Test Quality

- Automated program repair is a whiny child:
  - “You only said I had *get into* the bathtub, you didn't say I had to wash.”

Lessons Learned: Test Quality

- Automated program repair is a whiny child:
  - “You only said I had *get into* the bathtub, you didn't say I had to wash.”
- GenProg Day 1: gcd, nullhttpd
  - 5 tests for nullhttpd (GET index.html, etc.)
  - 1 bug (POST → remote exploit)
  - GenProg's fix: remove POST functionality
  - (Adding a 6th test yields a high-quality repair.)

Lessons Learned: Test Quality (2)

- MIT Lincoln Labs test of GenProg: sort
  - Tests: “the output of sort is in sorted order”
  - GenProg's fix: “always output the empty set”
  - (More tests yield a higher quality repair.)

Lessons Learned: Test Framework

- GenProg: binary / assembly repairs
  - Tests: “compare your-output.txt to trusted-output.txt”
  - GenProg's fix: “delete trusted-output.txt, output nothing”

- “Garbage In, Garbage Out”

Lessons Learned: Integration

- Integrating GenProg with a real program's test suite is non-trivial
- Example: spawning a child process
  - system("run test cmd 1 ..."); wait();
- wait() returns the error status
  - Can fail because the OS ran out of memory or because the child process ran out of memory
  - Unix answer: bit shifting and masking!

Lessons Learned: Integration (2)

- We had instances where PHP's test harness and GenProg's test harness wrapper disagreed on this bit shifting
  - GenProg's fix: “always segfault, which will mistakenly register as 'test passed' due to mis-communicated bit shifting”
- Think of deployment at a company:
  - Whose “fault” or “responsibility” is this?

Lessons Learned: Integration (3)

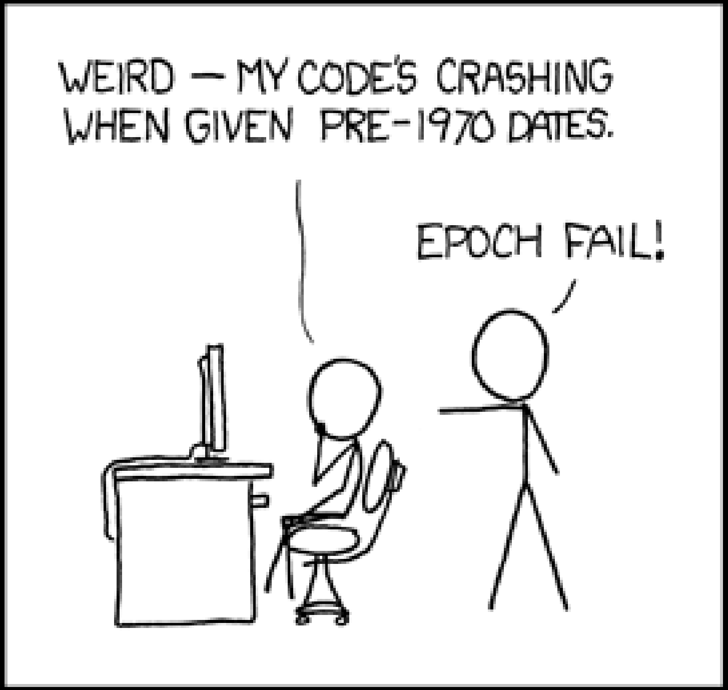
- GenProg has to be able to compile candidate patches
  - Just run “make”, right?
- Some programs, such as language interpreters, bootstrap or self-host.
  - We expected and handled infinite loops in tests
  - We did not expect infinite loops in compilation

Lessons Learned: Sandboxing

- GenProg has created …
  - Programs that kill the parent shell
  - Programs that “sleep forever” to avoid CPU-usage tests for infinite loops
  - Programs that allocate memory in an infinite loop, causing the Linux OOM killer to randomly kill GenProg
  - Programs that email developers so often that Amazon EC2 gave us the “we think you're a spammer” warning

Lessons Learned: Poor Tests

- Large open source programs have tests like:
  - Pass if today is less than December 31, 2012



Lessons Learned: Poor Tests

- Large open source programs have tests like:
  - Pass if today is less than December 31, 2012
  - Check that the modification times of files in this directory are equal to my hard-coded values
  - Generate a random ID with prefix “999”, check to see if result starts with “9996” (dev typo)

Lessons Learned: Sanity

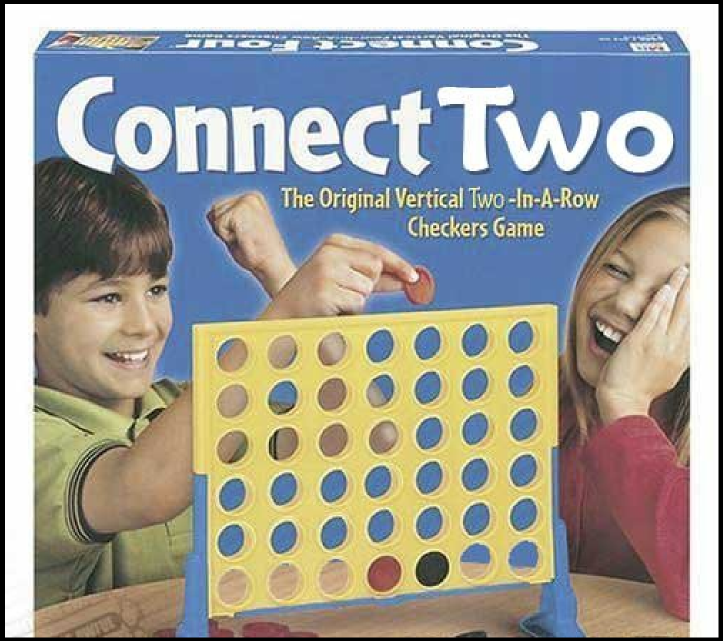
- Our earliest concession to reality was the addition of a “sanity check” to GenProg:
  - Does the program actually compile? Pass all non-bug tests? Fail all bug tests?
- A large fraction of our early reproduction difficulties were caught at this stage.

Challenges and Opportunities

- Test Suite Quality & Oracles
- Repair Quality

Challenge:

Test Suite Quality and Oracles



"A generated repair is the ultimate diagnosis in automated debugging – it tells the programmer where to fix the bug, what to fix, and how to fix it as to minimize the risk of new errors. A good repair depends on a good specification, though; and maybe the advent of good repair tools will entice programmers in improving their specifications in the first place."

– Andreas Zeller, Saarland University

Test Suite Quality & Oracles

- Repair_Quality = min(Technique, Test Suite)
- Currently, we trust the test suppliers
- What if we spent time on writing good specifications instead of on debugging?
- Charge: measure the suites we are using or generate high-quality suites to use
- Analogy: Formal Verification
  - Difficulty depends on more than program size

Test Data Generation

- We have all agreed to believe that we can create high-coverage test inputs

Test Data Generation

- We have all agreed to believe that we can create high-coverage test inputs
  - DART, CREST, CUTE, KLEE, AUSTIN, SAGE, PEX …
  - Randomized, search-based, constraint-based, concrete and symbolic execution, …
  - [Cadar, Sen: Symbolic execution for software testing: three decades later. Commun. ACM 56(2), 2013.]

Test Data Generation

- We have all agreed to believe that we can create high-coverage test inputs
  - DART, CREST, CUTE, KLEE, AUSTIN, SAGE, PEX …
  - Randomized, search-based, constraint-based, concrete and symbolic execution, ...
  - [Cadar, Sen: Symbolic execution for software testing: three decades later. Commun. ACM 56(2), 2013.]

- “And if it crashes on that input, that's bad.”

Test Oracle Generation

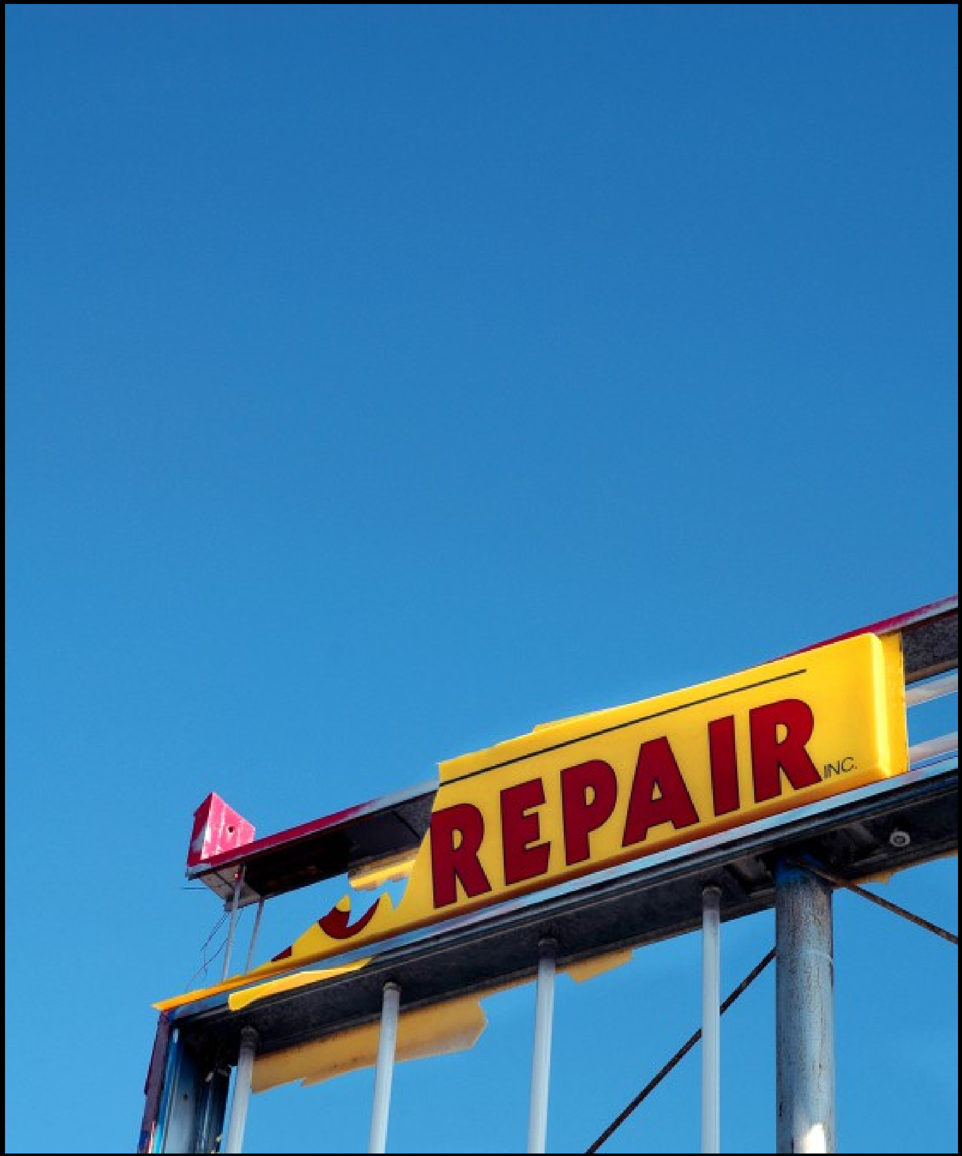
- What should the program be doing?
- μTEST [Fraser, Zeller: Mutation-Driven Generation of Unit Tests and Oracles. IEEE Trans. Software Eng. 38(2), 2012]
    - Great combination: Daikon + mutation analysis
    - Generate a set of candidate invariants
        - Running the program removes non-invariants
        - Retain only the useful ones: those killed by mutants
- [Staats, Gay, Heimdahl: Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. ICSE 2012.]
- [Nguyen, Kapur, Weimer, Forrest: Using dynamic analysis to discover polynomial and array invariants. ICSE 2012.]

Specification Mining

- Given a program (and possibly an indicative workload), generate partial-correctness specifications that describe proper behavior.
  [Ammons, Bodík, Larus: Mining specifications. POPL 2002.]
  - “Learn the rules of English grammar by reading student essays.”
- Problem: common behavior need not be correct behavior.
- Mining is most useful when the program deviates from the specification.

Challenge:

Repair Quality



Repair Quality

- Low-quality repairs may well be useless
- There are typically infinite ways to pass a test or implement a specification
- State of the art:
  - Report all repairs that meet the minimum requirements
- Charge:
  - Program repair papers should report on repair quality just as they report on quantity

A Pointed Fable

- [Das: Unification-based pointer analysis with directional assignments. PLDI 2000.]
  - “analyze a 1.4 MLOC program in two minutes”
- [Heintze, Tardieu: Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. PLDI 2001.]

A Pointed Fable

- [Das: Unification-based pointer analysis with directional assignments. PLDI 2000.]
  - “analyze a 1.4 MLOC program in two minutes”
- [Heintze, Tardieu: Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. PLDI 2001.]
- [Hind: Pointer analysis: haven't we solved this problem yet? PASTE 2001.]

A Pointed Fable

- [Das: Unification-based pointer analysis with directional assignments. PLDI 2000.]
  - "analyze a 1.4 MLOC program in two minutes"
- [Heintze, Tardieu: Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. PLDI 2001.]
- [Hind: Pointer analysis: haven't we solved this problem yet? PASTE 2001.]
- ??? [L. Regression: Analyzing 0.6 Million Lines of C Code in -119 Seconds. PLDI 2002.] ???

Pointer Analysis Lessons

- Common metrics:
  - Analyze X million lines of code
  - Analyze it in Y seconds
  - Answer's average "points-to set" size is Z

- Pushback:
  - "Points-to set size" is not a good metric.

You Can't Improve What You Can't Measure

- Cost to produce (time, money)
- Input required
- Functional Correctness
  - Addresses the “root of the problem”
  - Introduces no new defects
- Non-Functional Properties
  - Readable
  - Maintainable
  - Other?

Real-World Deployment (2017)

Fixing Bugs in Your Sleep: How Genetic Improvement Became an Overnight Success

Saemundur O. Haraldsson*
University of Stirling
Stirling, United Kingdom FK9 4LA
soh@cs.stir.ac.uk

John R. Woodward
University of Stirling
Stirling, United Kingdom FK9 4LA
jrw@cs.stir.ac.uk

Alexander E.I. Brownlee
University of Stirling
Stirling, United Kingdom FK9 4LA
sbr@cs.stir.ac.uk

Kristin Siggeirsdottir
Janus Rehabilitation Centre
Reykjavik, Iceland
kristin@janus.is

ABSTRACT

We present a bespoke live system in commercial use with self-improving capability. During daytime business hours it provides an overview and control for many specialists to simultaneously schedule and observe the rehabilitation process for multiple clients. However in the evening, after the last user logs out, it starts a self-analysis based on the day's recorded interactions. It generates test data from the recorded interactions for Genetic Improvement to fix any recorded bugs that have raised exceptions. The system has already been under test for over 6 months and has in that time identified, located, and fixed 22 bugs. No other bugs have been identified by other methods during that time. It demonstrates the

1 INTRODUCTION

Genetic Improvement (GI) [38] is a growing area within Search Based Software Engineering (SBSE) [23, 24] which uses computational search methods to improve existing software. Despite its growth within academic research the practical usage of GI has not yet followed. Like with many SBSE applications, the software industry needs an incubation period for new ideas where they come to trust in outcomes and see those ideas as cost effective solutions. GI is in the ideal position to shorten that period for the latter as it presents a considerable cost decrease for the software life cycle's often most expensive part: maintenance [18, 34]. There are examples of software improved by GI being used and publicly avail-

Real-World Deployment (2018)

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

Simon Urli
University of Lille & Inria Lille, France
simon.urli@inria.fr

Zhongxing Yu
University of Lille & Inria Lille, France
zhongxing.yu@inria.fr

Lionel Seinturier
University of Lille & Inria Lille, France
lionel.seinturier@inria.fr

Martin Monperrus
KTH Royal Institute of Technology, Sweden
martin.monperrus@csc.kth.se

ABSTRACT

Program repair research has made tremendous progress over the last few years, and software development bots are now being invented to help developers gain productivity. In this paper, we investigate the concept of a "program repair bot" and present Repairnator. The Repairnator bot is an autonomous agent that constantly monitors test failures, reproduces bugs, and runs program repair tools against each reproduced bug. If a patch is found, Repairnator bot reports it to the developers. At the time of writing, Repairnator uses three different program repair systems and has been operating since February 2017. In total, it has studied 11 523 test failures over 1 609 open-source software projects hosted on GitHub, and has generated patches for 15 different bugs. Over months, we hit a number of hard technical challenges and had to make various design and

in industry, it is desirable to study the design and implementation of an end-to-end repair toolchain that is amenable to the mainstream development practices.

For bridging this gap between research and industrial use, we investigate the concept of a "program repair bot" in this paper. To us, a program repair bot is an autonomous agent that constantly monitors test failures, reproduces bugs, and runs program repair tools against each reproduced bug. If a patch is found, the program repair bot reports it to the developers. We envision that in ten years from now there will be hundreds of program repair bots that will work in concert with developers to maintain large code bases. But today, to the best of our knowledge, nobody has ever reported on the design and operation of such a repair bot.

The Repairnator project is a project to design, implement and

Real-World Deployment (2019)

SapFix: Automated End-to-End Repair at Scale

A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, A. Scott
Facebook Inc.

***Abstract*—We report our experience with SAPFIX: the first deployment of automated end-to-end fault fixing, from test case design through to deployed repairs in production code[1]. We have used SAPFIX at Facebook to repair 6 production systems, each consisting of tens of millions of lines of code, and which are collectively used by hundreds of millions of people worldwide.**

INTRODUCTION

Automated program repair seeks to find small changes to software systems that patch known bugs [1], [2]. One widely studied approach uses software testing to guide the repair process, as typified by the GenProg approach to search-based program repair [3].

Recently, the automated test case design system, Sapienz [4], has been deployed at scale [5], [6]. The deployment of Sapienz allows us to find hundreds of crashes per month, before they even reach our internal human testers. Our software engineers have found fixes for approximately 75% of Sapienz-reported crashes [6], indicating a high signal-to-noise ratio [5] for Sapienz bug reports. Nevertheless, developers' time and expertise could undoubtedly be better spent on more creative programming tasks if we could automate some or all of the comparatively tedious and time-consuming repair process.

In order to deploy such a fully automated end-to-end detect-and-fix process we naturally needed to combine a number of different techniques. Nevertheless the SAPFIX core algorithm is a simple one. Specifically, it combines straightforward approaches to mutation testing [8], [9], search-based software testing [6], [10], [11], and fault localisation [12] as well as existing developer-designed test cases. We also needed to deploy many practical engineering techniques and develop new engineering solutions in order to ensure scalability.

SAPFIX combines a mutation-based technique, augmented by patterns inferred from previous human fixes, with a reversion-as-last resort strategy for high-firing crashes (that would otherwise block further testing, if not fixed or removed). This core fixing technology is combined with Sapienz automated test design, Infer's static analysis and the localisation infrastructure built specifically for Sapienz [6]. SAPFIX is deployed on top of the Facebook FBLearner Machine Learning infrastructure [13] into the Phabricator code review system, which supports the interactions with developers.

Because of its focus on deployment in a continuous integration environment, SAPFIX makes deliberate choices to sidestep some of the difficulties pointed out in the existing

Conclusion

Conclusion

- Industry is already paying untrusted strangers
- Automated Program Repair is a hot research area with rapid growth over a dozen years
  - (Lesson: “saying what you mean” is hard.)
- Challenges & Opportunities:
  - Test Suites and Oracles (spec mining)
  - Repair Quality (???)
- Real-world deployments have already started

Adaptive Equality Algorithm

For every repair, ordered by observations

Skip repair if equivalent to older repair

For every test, ordered by observations

Run the repair on the test, update obs.

Stop inner loop early if a test fails

Stop outer loop early if a repair validates