Computing Homomorphic Program Invariants

PhD Final Exam

11/16/2018

Ben Holland

A special thank you to my committee

I would not be here without each and everyone of you…

Background at Iowa State University

- 2005 – 2010
    - B.S. in Computer Engineering
    - Internship: Wabtec Railway Electronics, Ames Lab, Rockwell Collins
- 2010 – 2011
    - B.S. in Computer Science
    - Internship: Rockwell Collins
- 2010 – 2012
    - M.S. in Computer Engineering (Co-major Information Assurance)
    - Internship: MITRE
    - Thesis: Enabling Open Source Intelligence (OSINT) in private social networks
- 2012 – 2015
    - Research Associate → Assistant Scientist
    - DARPA’s APAC and STAC programs
        - Demands impactful and practical software solutions for open security problems
        - Fast-paced, high-stakes, adversarial engagement challenges
- 2015 – Present
    - Ph.D. in Computer Engineering
    - Graduate College’s Teaching Excellence Award (2016)
    - Graduate College’s Research Excellence Award (yesterday)
- December 2018
    - Apogee Research

Background at Iowa State University

- 2005 – 2010
  - B.S. in Computer Engineering
  - Internship: Wabtec Railway Electronics, Ames Lab, Rockwell Collins
- 2010 – 2011
  - B.S. in Computer Science
  - Internship: Rockwell Collins
- 2010 – 2012
  - M.S. in Computer Engineering (Co-major Information Assurance)
  - Internship: MITRE
  - Thesis: Enabling Open Source Intelligence (OSINT) in private social networks
- *2012 – 2015*
  - *Research Associate → Assistant Scientist*
  - *DARPA's APAC and STAC programs*
    - *Demands impactful and practical software solutions for open security problems*
    - *Fast-paced, high-stakes, adversarial engagement challenges*
- *2015 – Present*
  - *Ph.D. in Computer Engineering*
- December 2018
  - Apogee Research

An Ambitious Goal

- Seven years spread over two multi-million dollar DARPA projects has taken me on a journey seeking an answer to one over arching question…
    - *How can we find software security anomalies in practice?*

- This has been a team effort…
    - Too big of a problem for a single person
    - There are many aspects to this problem
    - The research presented today has grown organically

My Research Related Activities To Date

A: First Author Papers (4)

B: Coauthor Papers (5)

C: Papers in Review (1)

D: Book Chapters (1)

E: Invited Talks (5)

F: Invited Tutorials and Seminars (11)

G: Program Analysis Tools (12)

H: Courses (1)

A: Papers (First Author)

A1: **Benjamin Holland**, Tom Deering, Suresh Kothari, Jon Mathews, Nikhil Ranade. **Security Toolbox for Detecting Novel and Sophisticated Android Malware.** *The 37th International Conference on Software Engineering (ICSE 2015)*, Firenze, Italy, May 2015.

A2: **Benjamin Holland**, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. **Statically-informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities.** *The 16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2016)*, Raleigh, North Carolina, October 2016.

A3: **Benjamin Holland**, Ganesh Ram Santhanam, Suresh Kothari. **Transferring State-of-the-art Immutability Analyses: Experimentation Toolbox and Accuracy Benchmark.** *The 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*, Tokyo, Japan, March 2017.

A4: **Benjamin Holland**, Payas Awadhutkar, Suraj Kothari, Ahmed Tamrawi and Jon Mathews. **COMB: Computing Relevant Program Behaviors**. *The 40th International Conference on Software Engineering (ICSE 2018),* Gothenburg, Sweden, May 2018.

B: Papers (Coauthor)

B1: Suresh Kothari, Akshay Deepak, Ahmed Tamrawi, **Benjamin Holland**, Sandeep Krishnan. **A "Human-in-the-loop" Approach for Resolving Complex Software Anomalies.** *The 2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2014)*, San Diego, California, October 2014.

B2: Ganesh Ram Santhanam, **Benjamin Holland**, Suresh Kothari, Jon Mathews. **Interactive Visualization Toolbox to Detect Sophisticated Android Malware**. *The 14th IEEE Symposium on Visualization for Cyber Security (VizSec 2017)*, Phoenix, Arizona, October 2017.

B3: Payas Awadhutkar, Ganesh Ram Santhanam, **Benjamin Holland**, Suresh Kothari. **Intelligence Amplifying Loop Characterizations for Detecting Algorithmic Complexity Vulnerabilities**. *The 24th Asia-Pacific Software Engineering Conference (APSEC 2017)*, Nanjing, China, December 2017.

B4: Ganesh Ram Santhanam, **Benjamin Holland**, Suresh Kothari, Nikhil Ranade. **Human-on-the-loop Automation for Detecting Software Side-Channel Vulnerabilities.** *The 13th International Conference on Information System Security (ICISS 2017)*, Mumbai, India, December 2017.

B5: Ahmed Tamrawi, Sharwan Ram, Payas Awadhutkar, **Benjamin Holland**, Ganesh Ram Santhanam, Suresh Kothari. **DynaDoc: Automated On-Demand Context-Specific Documentation.** Third International Workshop on Dynamic Software Documentation (DySDoc3), Madrid, Spain, September 2018. ✩ *Winner of the 2018 DOCGEN challenge comprehensiveness category!*

C: Papers in Review

C1: Derrick Lockwood, **Benjamin Holland**, Suresh Kothari. **Mockingbird: A Framework for Enabling Targeted Dynamic Analysis of Java Programs.** *The 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019)*, Montreal, Canada, May 2019.

D: Book Chapters

D1: Suresh Kothari, Ganesh Santhanam, **Benjamin Holland**, Payas Awadhutkar, and Jon Mathews, Ahmed Tamrawi. **Catastrophic Cyber-Physical Malware.** *Springer Verlag Publishers*, April 2018.

E: Invited Talks

E1: **Benjamin Holland**, Suresh Kothari. **A Bug or Malware? Catastrophic consequences either way**. *Derbycon 4.0*, Louisville, Kentucky, September 2014.

E2: **Benjamin Holland**. **There's a hole in my bucket, dear Liza - Examining side channel leaks in web apps.** *OWASP Ames*, Ames, Iowa, August 2015.

E3: **Benjamin Holland**. **Developing Managed Code Rootkits for the Java Runtime Environment**. *DEFCON 24*, Las Vegas, Nevada, August 2016.

E4: **Benjamin Holland**. **Exploring the space in between bugs and malware**. *Iowa State University Cybersecurity Seminar Series*, Ames, Iowa, November 2016.

E5: **Benjamin Holland**. **JReFrameworker: One Year Later.** *Derbycon 7.0*, Louisville, Kentucky, September 2017.

E6: **Benjamin Holland**. **Recent Trends in Program Analysis for Bug Hunting and Exploitation**. *SecDSM*, Des Moines, Iowa, September 2018.

F: Invited Tutorials and Seminars

F1: Suresh Kothari, **Benjamin Holland**. **Practical Program Analysis for Discovering Android Malware**. *MILCOM 2015*, Tampa, Florida, October 2015.

F2: Suresh Kothari, **Benjamin Holland**. **Hard Problems at the Intersection of Cybersecurity and Software Reliability**. *The 26th IEEE International Symposium on Software Reliability Engineering (ISSRE 2015)*, NIST, Gaithersburg, Maryland, November 2015.

F3: Suresh Kothari, **Benjamin Holland**. **Computer-aided Collaborative Validation of Large Software**. *The 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, Lincoln, Nebraska, November 2015.

F4: Suresh Kothari, **Benjamin Holland**. **Learn to Build Automated Software Analysis Tools with Graph Paradigm and Interactive Visual Framework**. *The 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, Singapore, September 2016.

F5: Suresh Kothari, **Benjamin Holland**. **Managing Complexity, Security, and Safety of Large Software**. *Global Initiative of Academic Networks (GIAN)*, Jaipur, India, September 2016.

F6: Suresh Kothari, **Benjamin Holland**. **Discovering Information Leakage Using Visual Program Models**. *MILCOM 2016*, Baltimore, Maryland, November 2016.

F7: **Benjamin Holland**. **Program Analysis for Cybersecurity**. *US Cyber Challenge Summer Bootcamps (USCC 2017)*, Illinois, Delaware, and Utah, July 2017.

F8: Suresh Kothari, **Benjamin Holland**. **Learn to Analyze and Verify Large Software for Cybersecurity and Safety**. *MILCOM 2017*, Baltimore, Maryland, October 2017.

F9: **Benjamin Holland**. **Cyber Security Awareness and Cyber Security Challenge Competition**. Malaviya National Institute of Technology (MNIT), Jaipur, India, July 2018.

F10: **Benjamin Holland**. **Program Analysis for Cybersecurity**. *US Cyber Challenge Summer Bootcamps (USCC 2018)*, Illinois, Delaware, and Virginia, Nevada, July 2018.

F11: Suresh Kothari, **Benjamin Holland**. **Systematic Exploration of Critical Software for Catastrophic Cyber-Physical Malware.** *MILCOM 2018*, Los Angeles, California, October 2018.

G: Program Analysis Tools Developed

Tool	Release	Dates	Citations	Logical LoC	LoC (Commit) Contributions	Description
[E1] APAC Toolbox	Private	2012-2014	A1, B2	52,712	49%	DARPA APAC Toolchain
[E2] Android Essentials	Open Source	2014-present	A1, B2, E1, F1	5,755	100%	Android permission mappings, manifest parser, XML UI / language / resource indexer
[E3] WAR Binary Processing	Open Source	2014-present	STAC Proposal	720	100%	Java WAR (JSP webserver applications) frontend binary support for Atlas
[E4] Toolbox Commons	Open Source	2015-present	None	15,391	99%	Language agnostic analysis, with extensions for C/C++, Java, and JVM bytecode specific analysis
[E5] STAC (RULER)	Private	2015-present	B3, B4	23,652	89%	DARPA STAC Toolchain
[E6] JReFrameworker	Open Source	2015-present	E3, C5, F7	37,664	100%	Abstracted JVM bytecode manipulation framework with applications to JVM rootkits
[E7] Points-To Toolbox	Open Source	2016-present	A3, F5	2,217	100%	Precise points-to analysis, on-the-fly call graph resolution, support for primitives and arrays.

G: Program Analysis Tools Developed (Cont.)

Tool	Release	Dates	Citations	Logical LoC	LoC (Commit) Contributions	Description
[E8] Call Graph Toolbox	Open Source	2016-present	F5	2,535	100%	9 call graph construction algorithms (RA → CHA → RTA → RTA Variants → points-to), library callback analysis for partial program analysis
[E9] Slicing Toolbox	Open Source	2016-present	F5, F6, F7, F9, F10	1,136	100%	Classical program slicing with basic system dependence graph
[E10] SIDIS Toolbox	Open Source	2016-present	A2	2,476	100%	Atlas to SOOT graph correspondence, bytecode manipulation, extensible instrumentation logic
[E11] Immutability Toolbox	Open Source	2016-present	A3	13,571	99%	2 accuracy benchmarked algorithms to compute immutability, side effect analysis, and function purity
[E12] PCG Toolbox (COMB)	Open Source	2017-present	A4	3,843	99%	Implementation of Project Control Graphs with a CHA based IPCG implementation

Personally contributed approximately eighty-five thousand lines of code publicly released under MIT license…

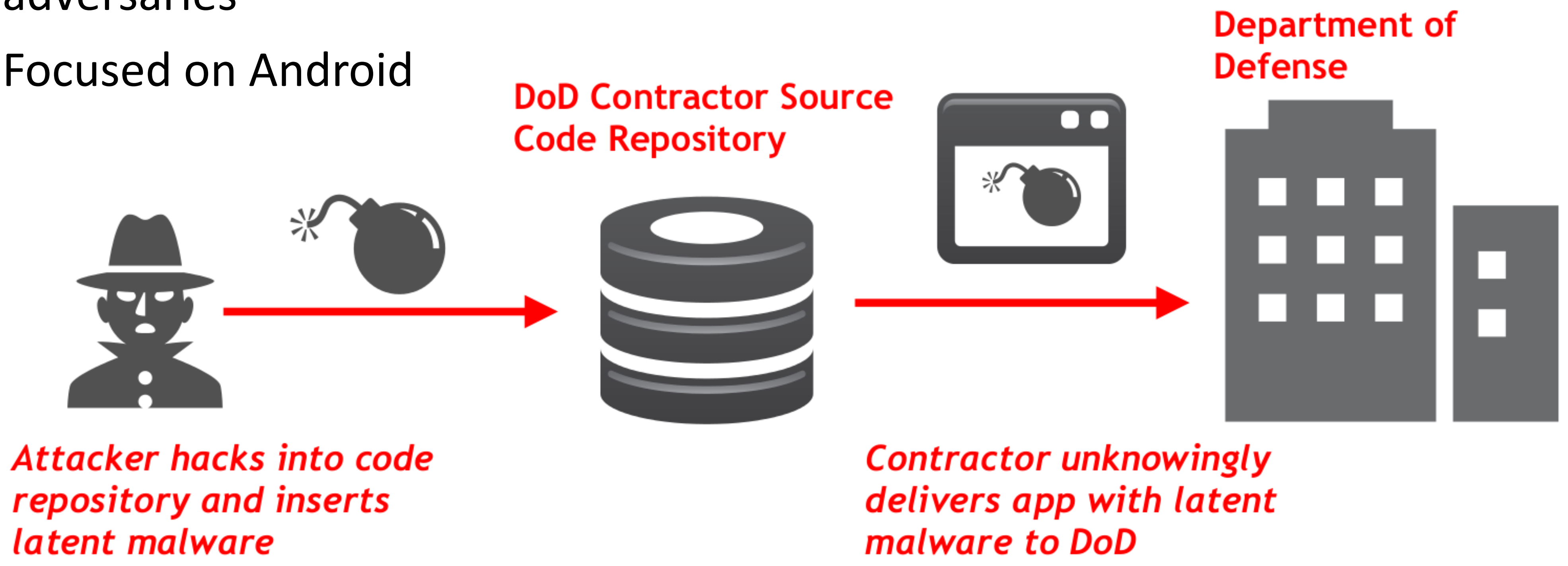
H: Courses

H1: **Benjamin Holland**. **SE 421: Software Analysis and Verification for Safety and Security.** *Iowa State University*, Fall 2018. (introduced Spring 2018)

Previously TA'd or Co-taught 7 other semester courses

DARPA’s APAC Program (2012 – 2014)

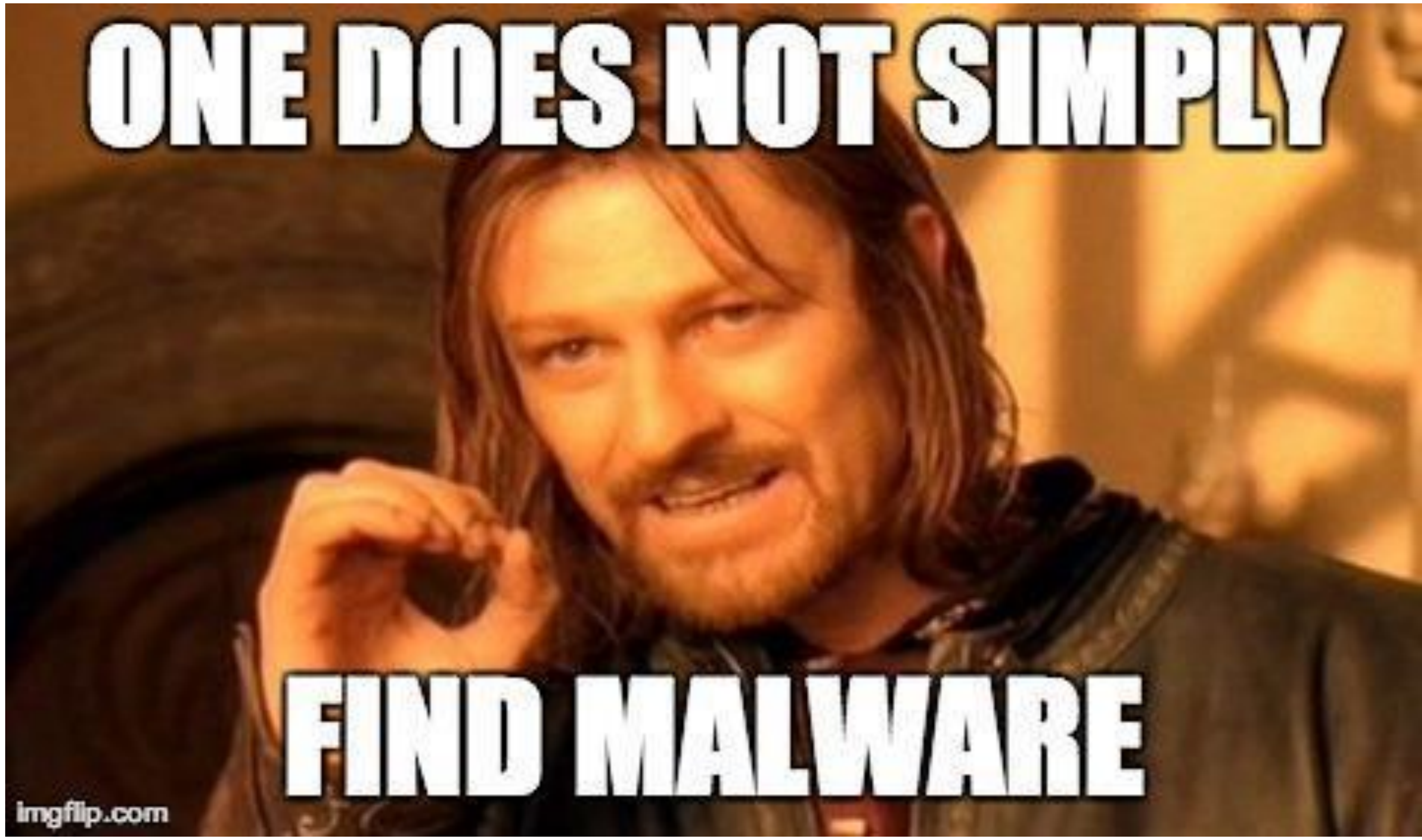
- Automated Program Analysis For Cybersecurity (APAC)
- Scenario: Hardened devices, internal app store, untrusted contractors, expert adversaries
- Focused on Android


DoD Contractor Source Code Repository
Department of Defense
Attacker hacks into code repository and inserts latent malware
Contractor unknowingly delivers app with latent malware to DoD

https://www.darpa.mil/program/automated-program-analysis-for-cybersecurity

Hard Lessons for a Younger Researcher

- Is a bug malware? What if its planted intentionally? We know bug finding is hard…
- Bugs have plausible deniability and malicious intent cannot be determined from code.
- Novel attacks have escaped previous threat models.
- Need precision tools to detected **novel** and **sophisticated** malware in advance!
- *Augmented human reasoning is incredibly powerful and extremely difficult to match through automation alone.*



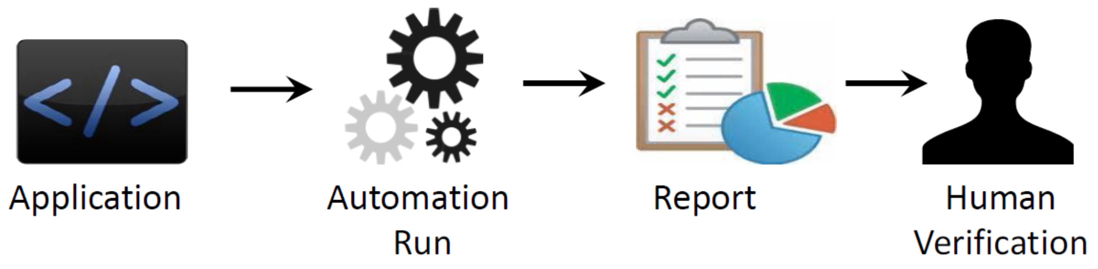
Human-in-the-loop Approach to Detecting Software Security Anomalies

Amplified Reasoning Technique (ART)

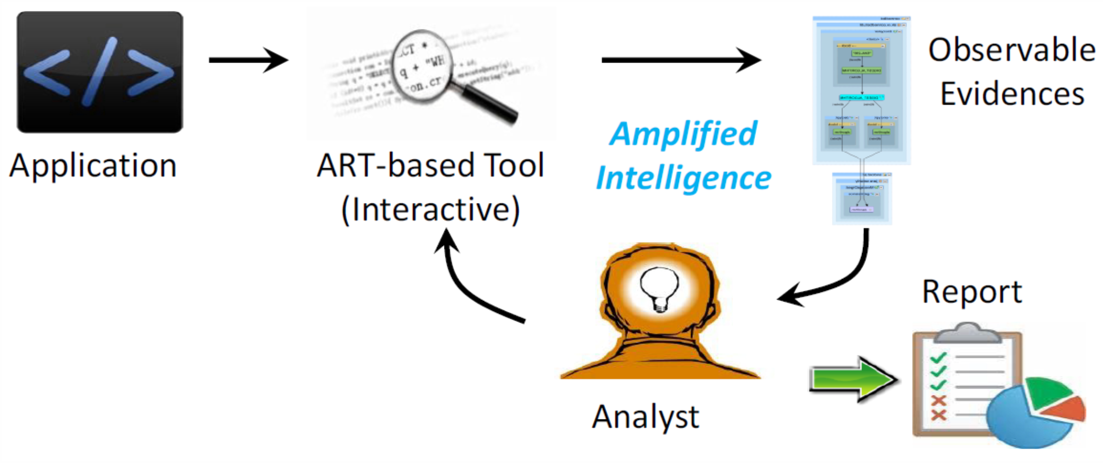
1. Hypothesize anomalous behaviors
2. Interactive static analysis to refine hypothesis
3. Dynamic experimentation to validate or refute hypothesis
4. Rinse and repeat

Papers:

- [A1] Security Toolbox for Detecting Novel and Sophisticated Android Malware
- [B1] A "Human-in-the-loop" Approach for Resolving Complex Software Anomalies
- [B2] Interactive Visualization Toolbox to Detect Sophisticated Android Malware



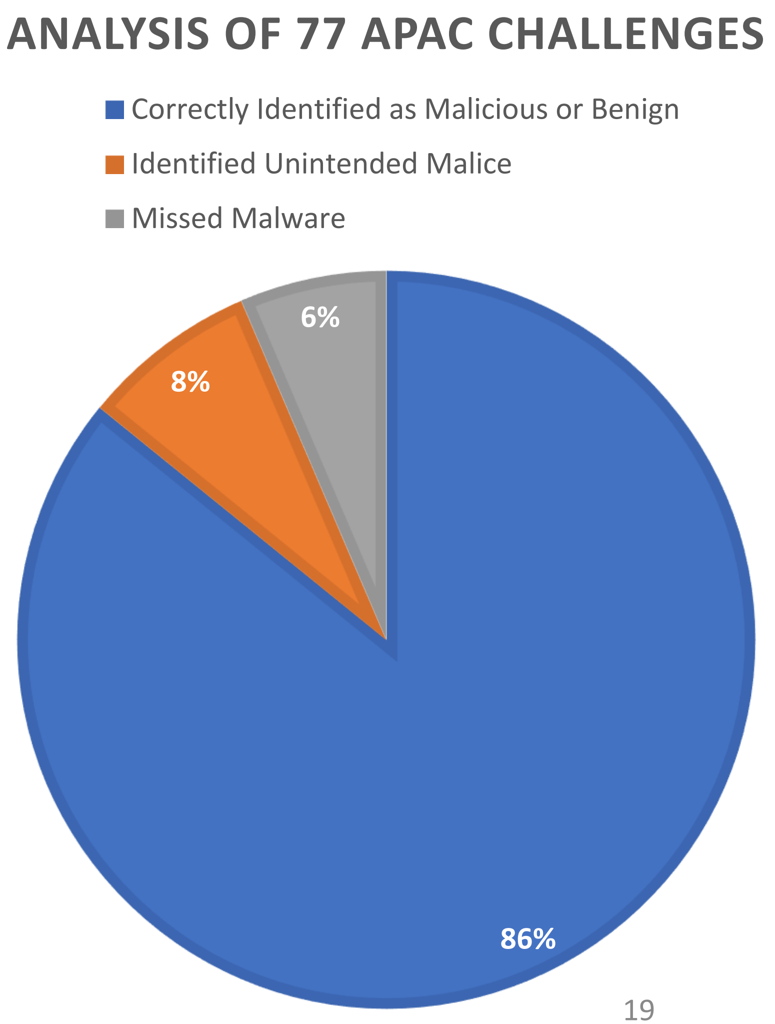
(a) Traditional automation



(b) ART-based approach

APAC Results (2012 - 2014)

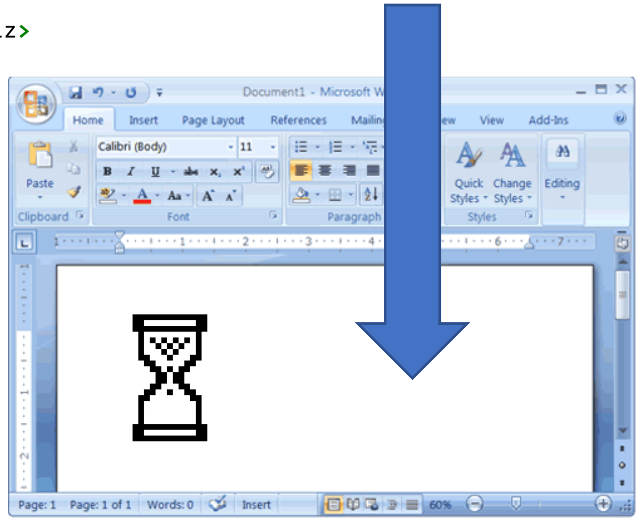
- Initially the only team to apply human-in-the-loop reasoning
- Completed Phase I of the DARPA APAC program as the top performing R&D team
- 62/77 Android source code applications developed by the Red team contained novel malware able to evade current automatic detection techniques
- APAC's Phase II we ranked closely among the top three performing R&D teams (difference of one challenge application between top 3), who had all adopted human-in-the-loop approaches



DARPA's STAC Program (2015 – present)

- Space/Time Analysis for Cybersecurity (STAC)
- Scenario: Detect algorithmic complexity (AC) and side channel (SC) vulnerabilities in a compiled bytecode applications
- Measured with respect to execution time or volatile/non-volatile memory space and an attacker input budget
  - Example: Send 1k byte request to cause 300 sec runtime execution
  - Example: Measure the response times of 100 requests to learn private key

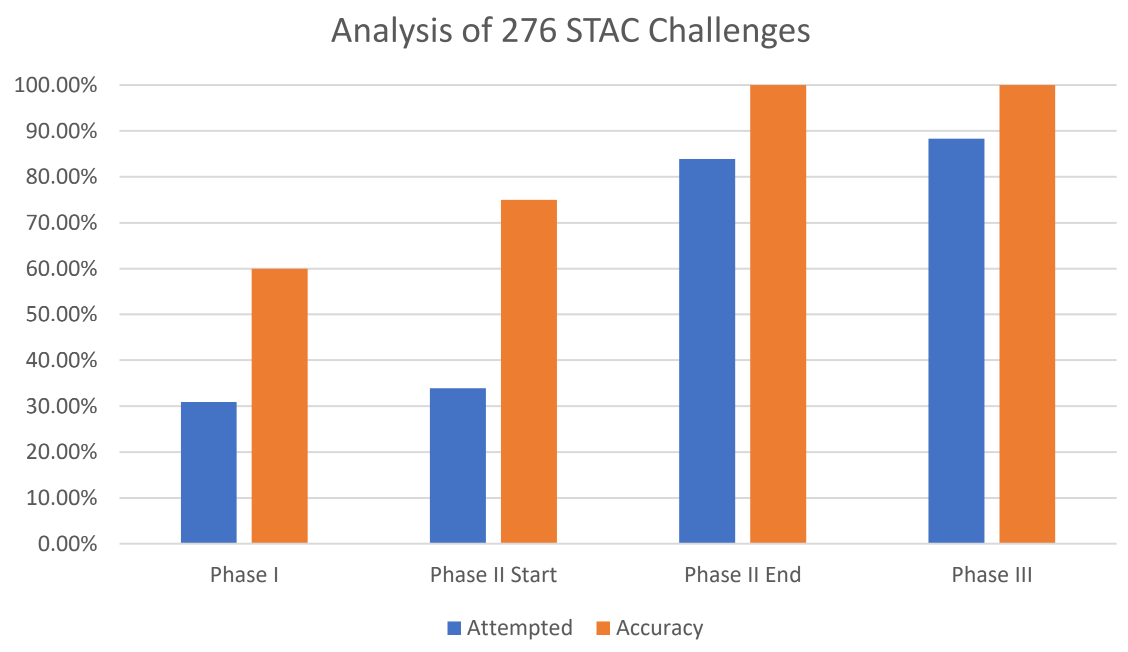
```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```



https://www.darpa.mil/program/space-time-analysis-for-cybersecurity

STAC Results: (2015 – present)

- Transitioned human-in-the-loop approach from APAC to STAC
  - [A3] Transferring State-of-the-art Immutability Analyses: Experimentation Toolbox and Accuracy Benchmark
  - [B3] Human-on-the-loop Automation for Detecting Software Side-Channel Vulnerabilities
  - [B4] Intelligence Amplifying Loop Characterizations for Detecting Algorithmic Complexity Vulnerabilities
- Remaining Challenges



Review: Program Invariants

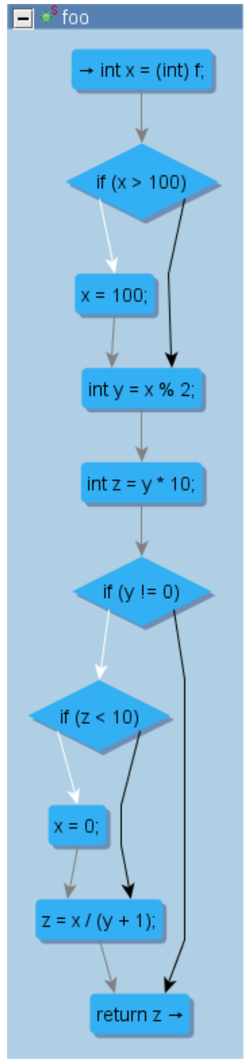
"Programmers have invariants in mind … when they write or otherwise manipulate programs: they have an idea of how the system works or is intended to work, how the data structures are laid out and related to one another, and the like. Regrettably, these notions are rarely written down..." ~ Michael Ernst

Program Invariant:

- "a property that is true at a particular program point or points" [Ernst 2000]
- "a property of a program that is always true for every possible runtime state of the program" [MIT OpenCourseWare 6.005]

Review: Control Flow Graph

```java
public int foo(float f) {
    int x = (int) f;
    if(x > 100) {
        // limit x to 100
        x = 100;
    }
    int y = x % 2;
    int z = y * 10;
    if(y != 0) { // if y is odd
        if(z < 10) {
            // round off to zero
            x = 0;
        }
        z = x / (y + 1);
    }
    return z;
}
```



Recap of Preliminary Exam (March 2018)

- Thesis: “Computing homomorphic program invariants is novel, useful, and practical.”
  - Novelty: Homomorphic program invariants – with respect to an equivalence class of control flow paths*
  - Usefulness: Homomorphic program invariants can lead to stricter assertions

*In 1979, Tarjan introduces graph homomorphisms based on equivalence classes of paths in a graph

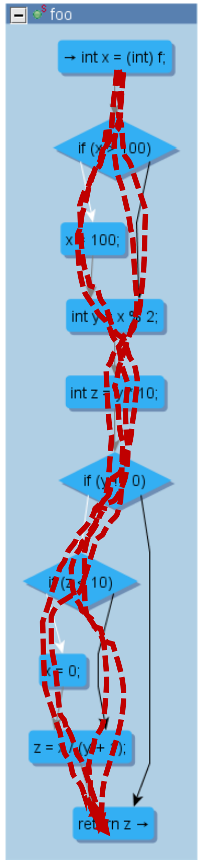
Tarjan, Robert Endre. *A Unified Approach to Path Problems*. No. STAN-CS-79-729. STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE, 1979.

Illustration of Stricter Assertions

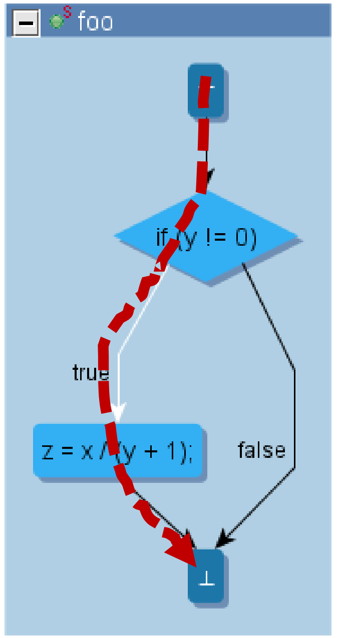
What are the values of y when the true path of the y != 0 branch is taken?

Recall: *y = x % 2;*
y ∈ {-1, 0, 1} for all paths
y ∈ {-1, 1} for the 4 relevant paths

… if y is -1 then division by zero will occur!



4 paths in the control flow graph are equivalent with respect to whether or not a division operation occurs

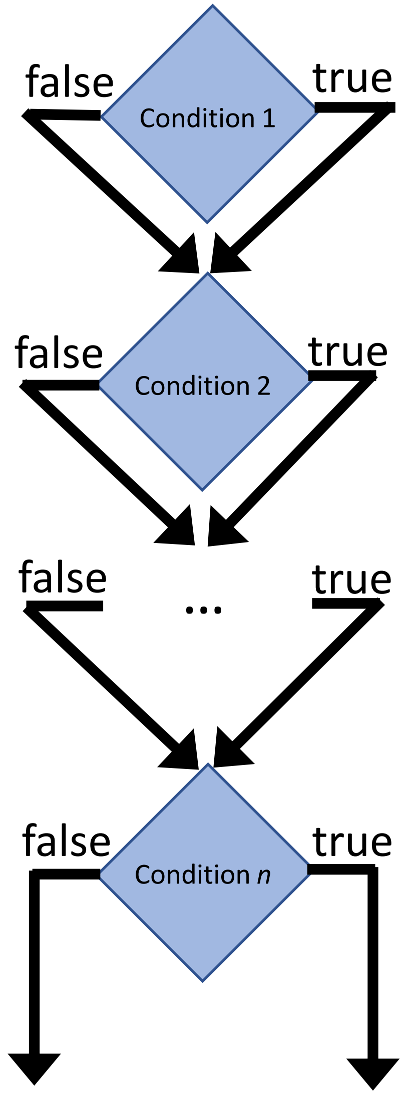


Challenge: Path Explosion Problem

- The number of paths through programs is astronomical
- A single function in the Linux kernel (*lustre_assert_wire_constants*) has 2^{656} paths!
    - Only 10^{80} atoms in the known universe…
    - $2^{656} \approx 10^{197}$

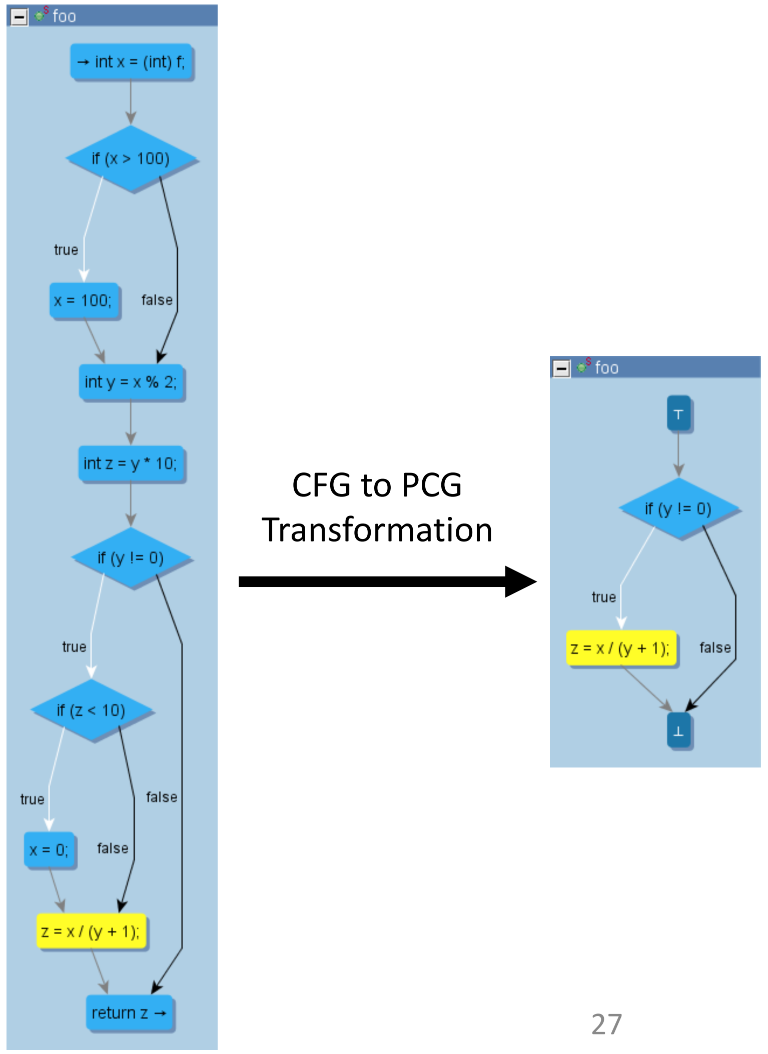
2^n paths!

```
if(condition_1){
   // code block 1
}
if(condition_2){
   // code block 2
}
if(condition_3){
   // code block 3
}
…
if(condition_n){
   // code block n
}
```



Projected Control Graph

- In 2016, Tamrawi proposed a PCG abstraction
  - Defined a graph homomorphism to efficiently group program behaviors into equivalence classes
  - Parameterized by control flow events of interest
  - Only relevant event statements and necessary conditions are retained



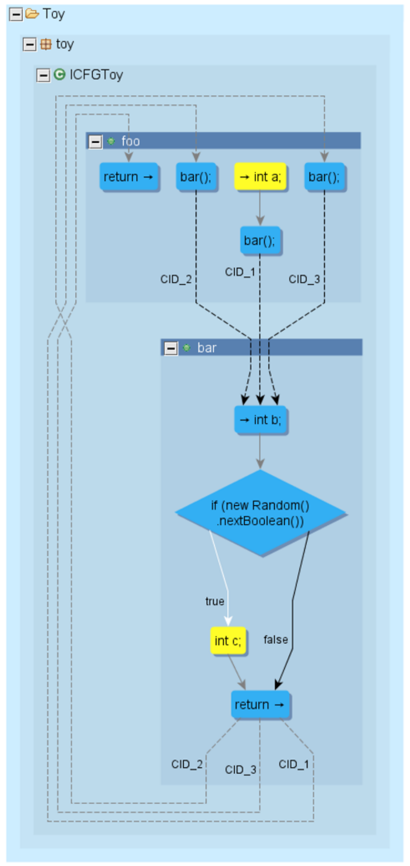
PCG Toolbox (COMB)

- COMB provided extensible infrastructure for applying PCG abstraction to solve different problems
- Key Contributions
  - Interactive User Interface
  - Inter-procedural analysis

A4: **Benjamin Holland**, Payas Awadhutkar, Suraj Kothari, Ahmed Tamrawi and Jon Mathews. **COMB: Computing Relevant Program Behaviors**. *The 40th International Conference on Software Engineering (ICSE 2018)*, Gothenburg, Sweden, May 2018.

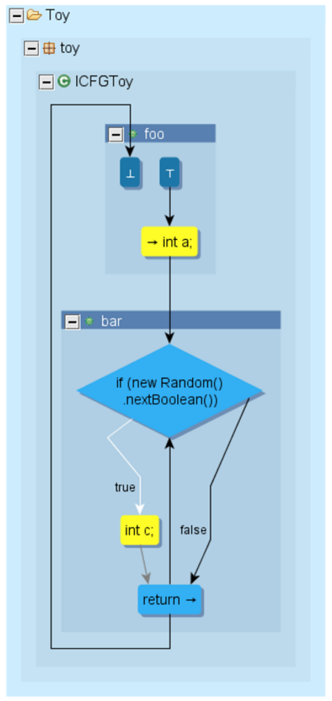
Inter-procedural Analysis with COMB

```java
public class ICFGToy {
  public void foo() {
    int a;
    bar();
    bar();
    bar();
    return;
  }
  public void bar() {
    int b;
    if(new Random().nextBoolean()) {
      int c;
    }
    return;
  }
}
```



Inter-procedural Control Flow Graph (ICFG)

PCG transformation with respect to selected events

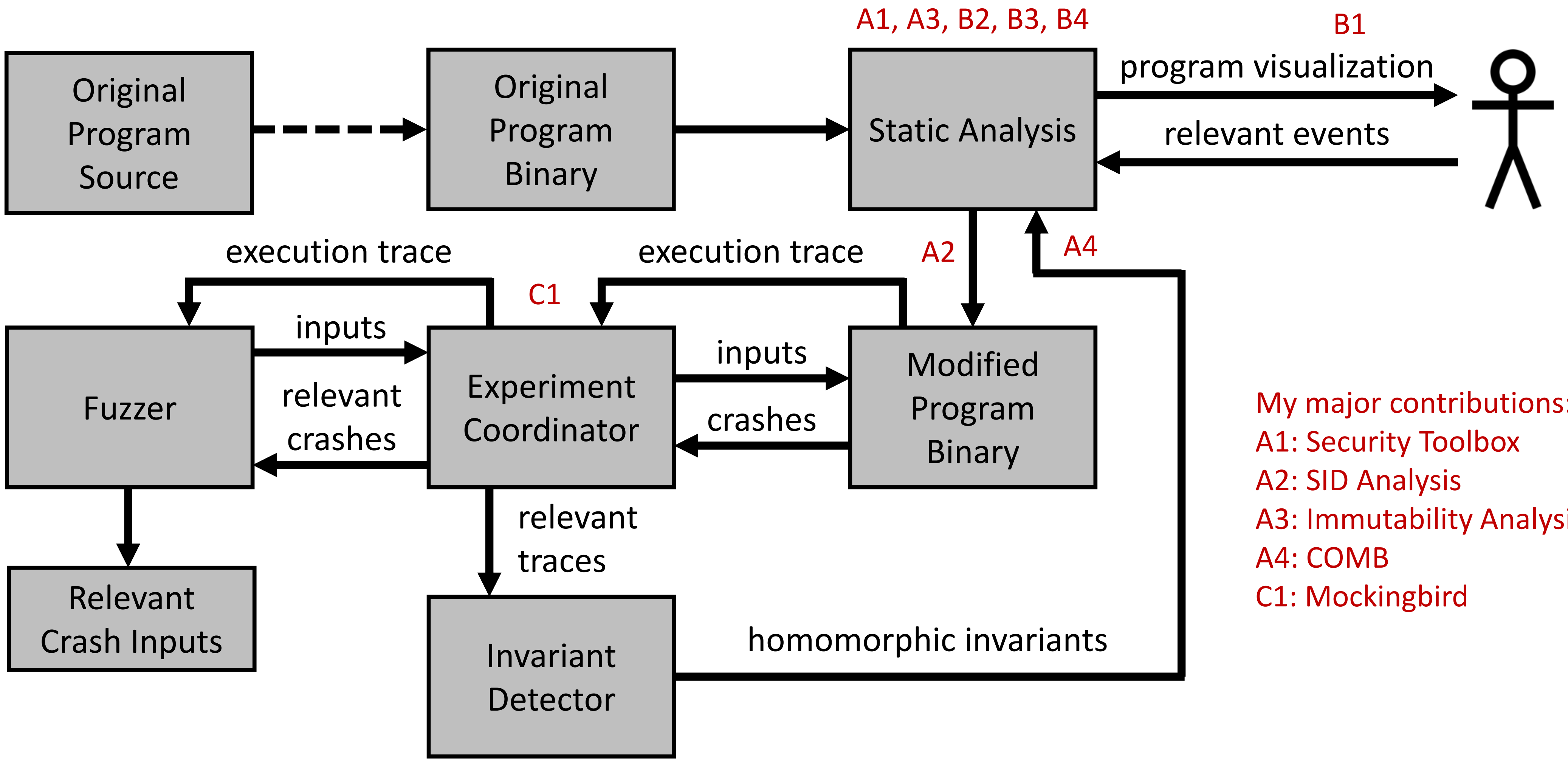


Inter-procedural Projected Control Graph (IPCG)

Computing Homomorphic Program Invariants

- Thesis: “Computing homomorphic program invariants is novel, useful, and practical.”
  - Practical: Homomorphic program invariants can be computed with standard hardware (e.g. personal computer)

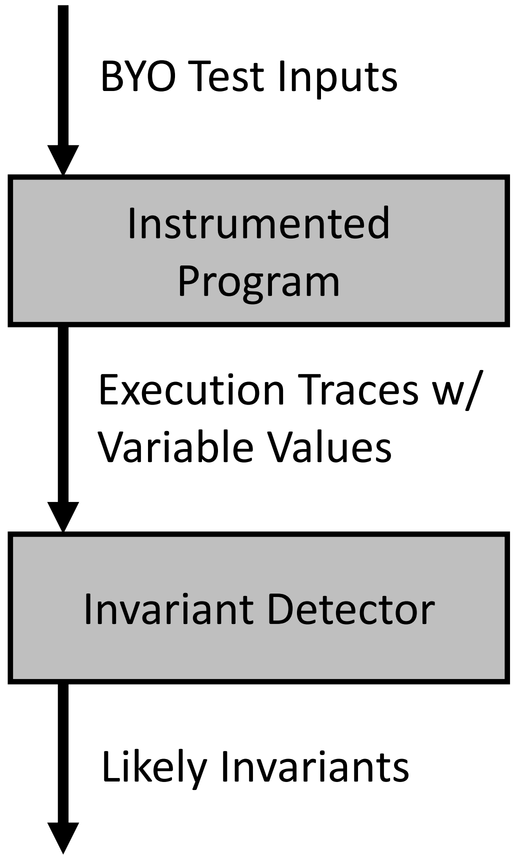
Computing Homomorphic Program Invariants



My major contributions:
A1: Security Toolbox
A2: SID Analysis
A3: Immutability Analysis
A4: COMB
C1: Mockingbird

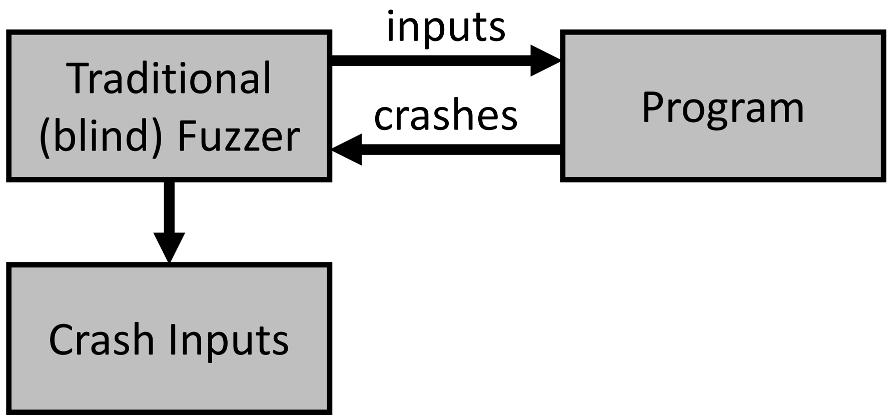
State-of-the-art: Dynamic Invariant Detection

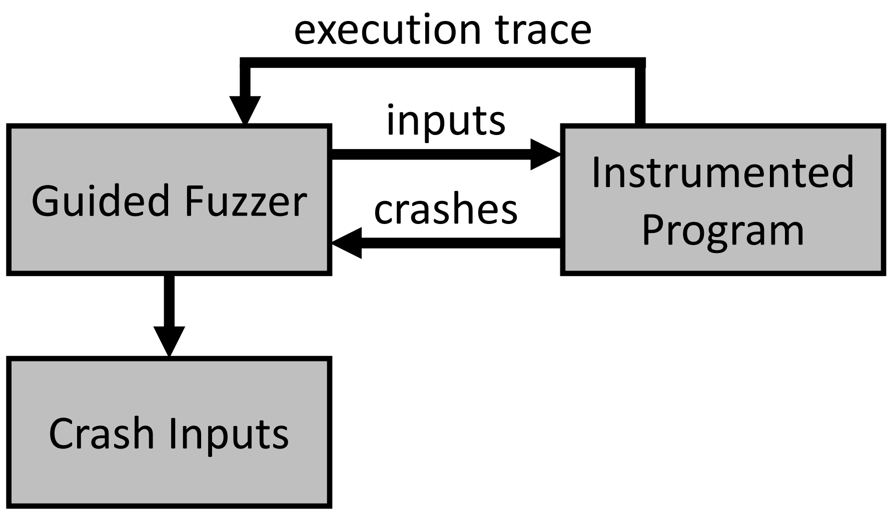
- Daikon: Dynamic Likely Invariant Detection
  - Dynamic analysis only observes feasible paths
  - Program variables are instrumented on all program paths
  - BYO test input strategy, typically used with unit tests or randomized testing
  - Large collection of program invariant patterns (ex: types)
  - Correctness is w.r.t. what was observed. Example: “x > 0” may only be true if negative values were never tested.
  - Can be expensive. Instrumentation adds overhead to execution time and invariant detection must employ many logic tricks in order to scale.



State-of-the-art: Guided Fuzzing

- American Fuzzy Lop (AFL): Guided Fuzzer
  - Effective mutation strategy to generate new inputs from initial test corpus
  - Lightweight instrumentation at branch points
  - Genetic algorithm promotes mutations of inputs that discover new branch edges
    - Aims to explore all code paths
  - Huge trophy case of bugs found in wild
    - 371+ reported bugs in 161 different programs as of March 2018
    - Responsible for a large majority of vulnerabilities found in DARPA CGC

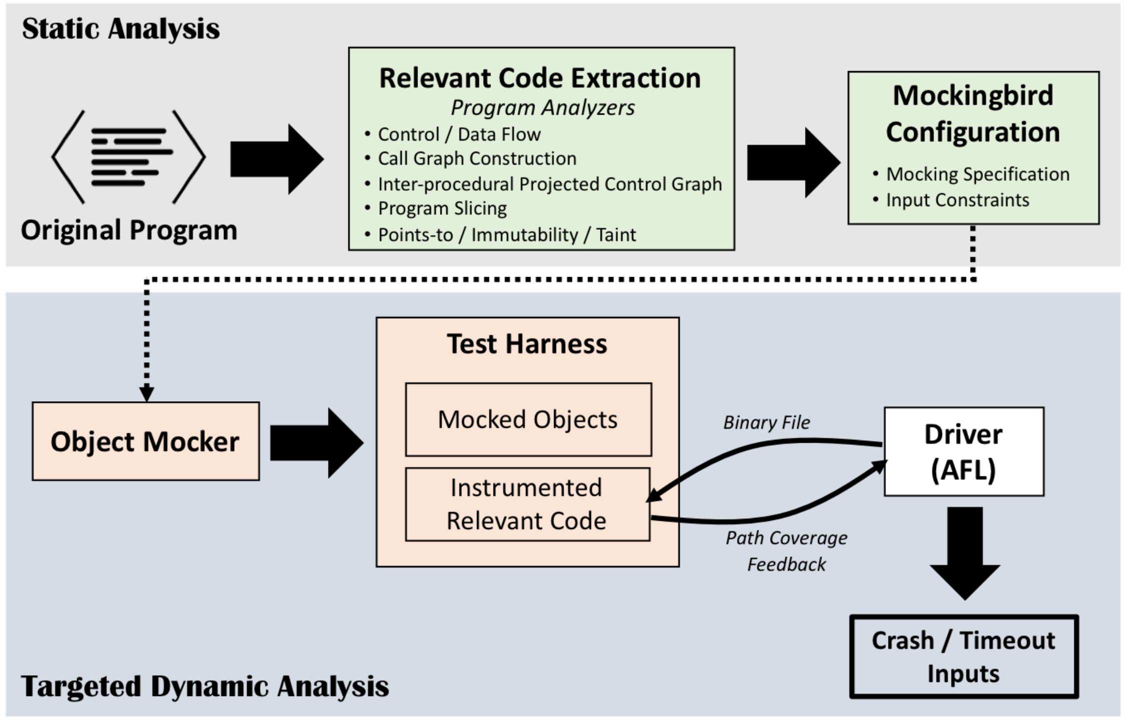




Targeted Dynamic Analysis

- Decouples target code from control and data dependencies by replacing objects with mocked objects
  - Global variables
  - Method parameters passed
  - Method return values

- Mocked objects have no dependencies
- Mocked object values can be programmatically stimulated



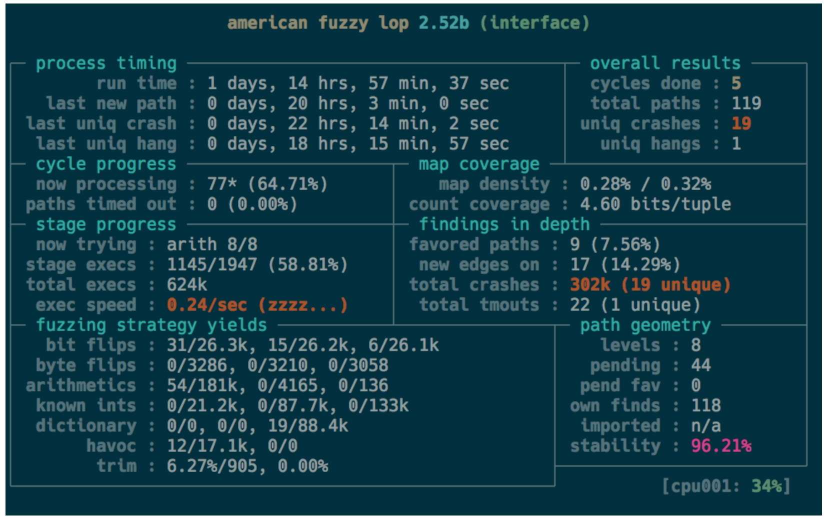
Paper: [C1] Mockingbird: A Framework for Enabling Targeted Dynamic Analysis of Java Programs.

Example Targeted Dynamic Analysis

```java
public class Example {
  public static boolean isVowel(char c) {
    return c == 'a' || c == 'e' || c == 'i'
        || c == 'o' || c == 'u' || c == 'y';
  }
  class Pet {
    private String name;
    public Pet(String name) {
      this.name = name;
      sleep(5000);
    }
    public String getName() {
      return name;
    }
    public double getVowelRatio() {
      double vowels = 0;
      String name = getName().toLowerCase();
      for(char c : name.toCharArray()) {
        if(isVowel(c)) {
          vowels++;
        }
      }
      return vowels / (name.length() - vowels);
    }
  }
}
```

Mock Specification

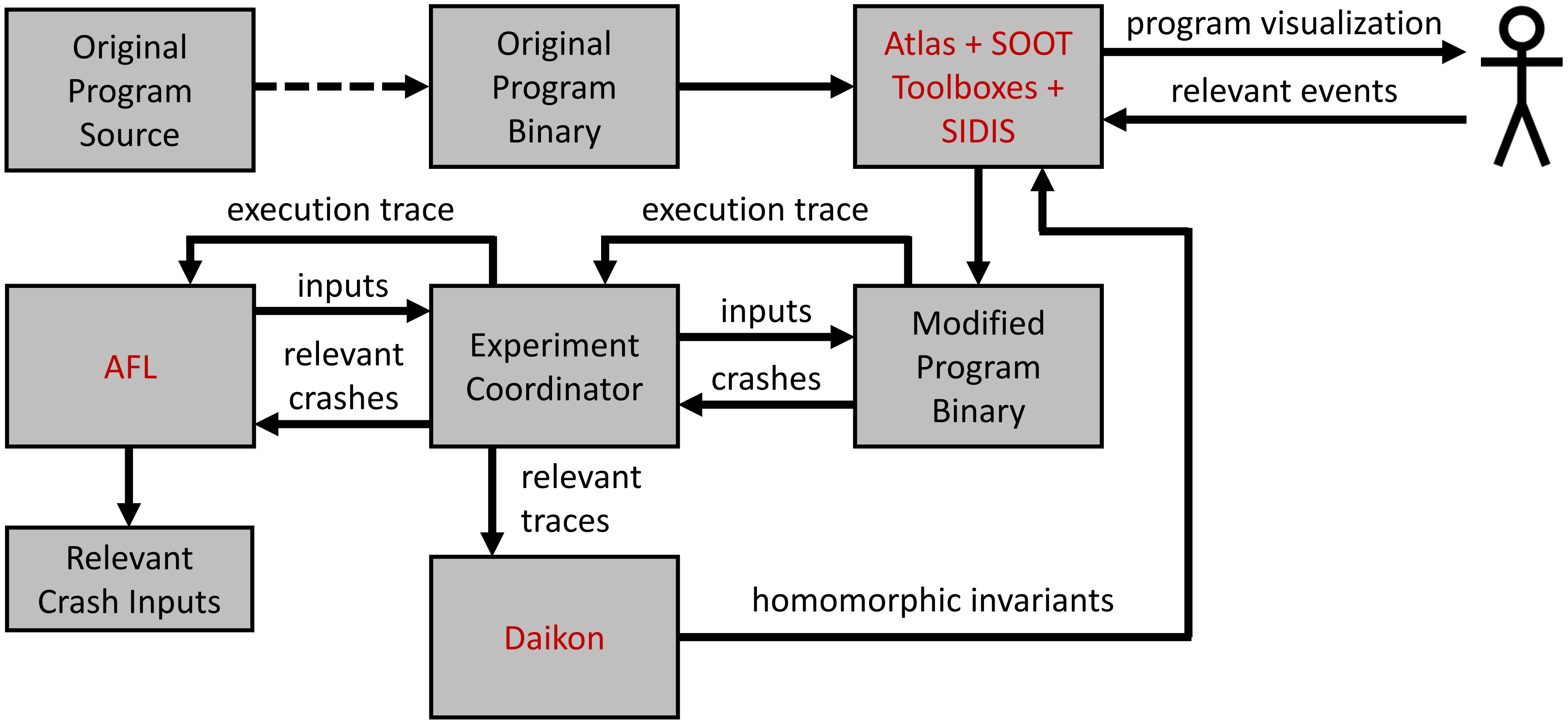
```json
{
  "definition": {
    "class": "Example$Pet",
    "method": "getVowelRatio",
    "instance_variables": [
        {
                "name": "name"
        }
    ]
  },
  "config": {
    "timeout": 1000
  }
}
```



```
                    american fuzzy lop 2.52b (interface)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 14 hrs, 57 min, 37 sec      │  cycles done : 5      │
│   last new path : 0 days, 20 hrs, 3 min, 0 sec        │  total paths : 119    │
│ last uniq crash : 0 days, 22 hrs, 14 min, 2 sec       │ uniq crashes : 19     │
│  last uniq hang : 0 days, 18 hrs, 15 min, 57 sec      │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 77* (64.71%)      │    map density : 0.28% / 0.32%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 4.60 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 8/8             │ favored paths : 9 (7.56%)              │
│ stage execs : 1145/1947 (58.81%)    │  new edges on : 17 (14.29%)            │
│ total execs : 624k                  │ total crashes : 302k (19 unique)       │
│  exec speed : 0.24/sec (zzzz...)    │  total tmouts : 22 (1 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 31/26.3k, 15/26.2k, 6/26.1k           │    levels : 8          │
│  byte flips : 0/3286, 0/3210, 0/3058                │   pending : 44         │
│ arithmetics : 54/181k, 0/4165, 0/136                │  pend fav : 0          │
│  known ints : 0/21.2k, 0/87.7k, 0/133k              │ own finds : 118        │
│  dictionary : 0/0, 0/0, 19/88.4k                    │  imported : n/a        │
│       havoc : 12/17.1k, 0/0                         │ stability : 96.21%     │
│        trim : 6.27%/905, 0.00%                      ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu001: 34%]
```

SIDIS Framework Architecture



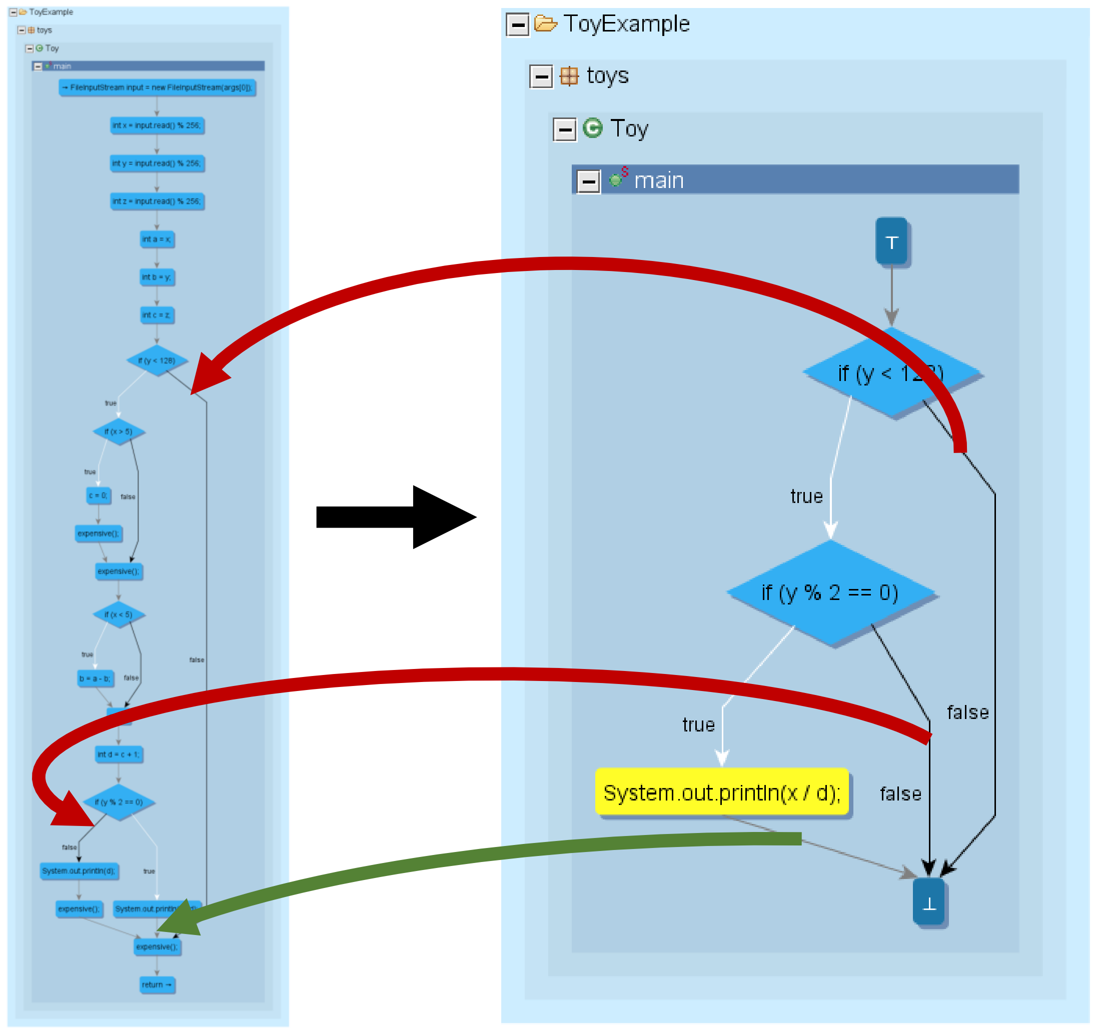
Motivating Example

- Is this program bug free?
- Could a division by zero error occur on line 24?
- What conditions are relevant to verifying the program?
- If the program is buggy, what inputs are required to produce the error?
- What input constraints must be satisfied to produce the error?
- Is there a family of buggy inputs?

```
 1 public class Toy {
 2    public static void main(String[] args) throws Exception {
 3       FileInputStream input = new FileInputStream(args[0]);
 4       int x = input.read() % 256; // x = -1 to 255
 5       int y = input.read() % 256; // y = -1 to 255
 6       int z = input.read() % 256; // z = -1 to 255
 7
 8       int a = x;
 9       int b = y;
10       int c = z;
11
12       if(y < 128) {
13          if(x > 5) {
14             c = 0;
15             expensive();
16          }
17          expensive();
18          if(x < 5) {
19             b = a - b;
20          }
21          c = b;
22          int d = c + 1;
23          if(y % 2 == 0) {
24             System.out.println(x / d);
25          } else {
26             System.out.println(d);
27             expensive();
28          }
29       }
30       expensive();
31    }
32 }
```

Program Modifications (1)

- Technique 1 - Aborting Irrelevant Path Execution
  - Only modification needed to compute behavior-relevant invariants
  - Inject an abort signal at the start of an irrelevant path
  - Insert an *abort-irrelevant* signal before any statement in the CFG that is a successor of a branch reachable from a reverse step in the PCG from the ⊥, omitting events
  - Optionally, insert an *abort-relevant* signal after events reachable in a reverse step of the PCG from the ⊥

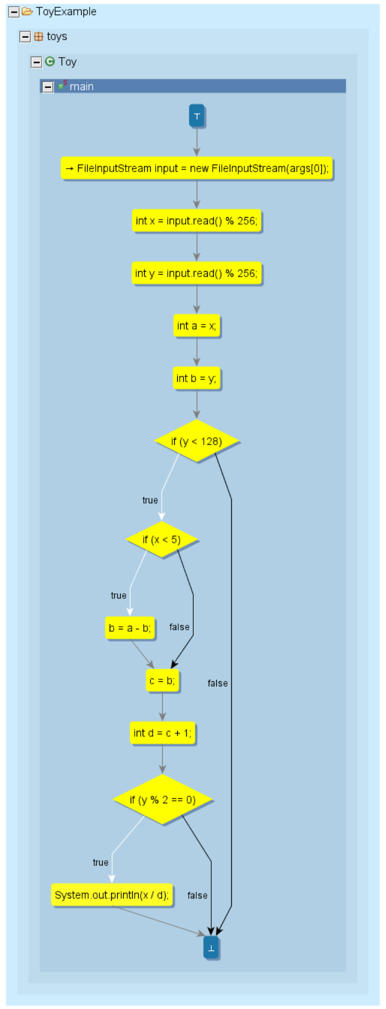


CFG

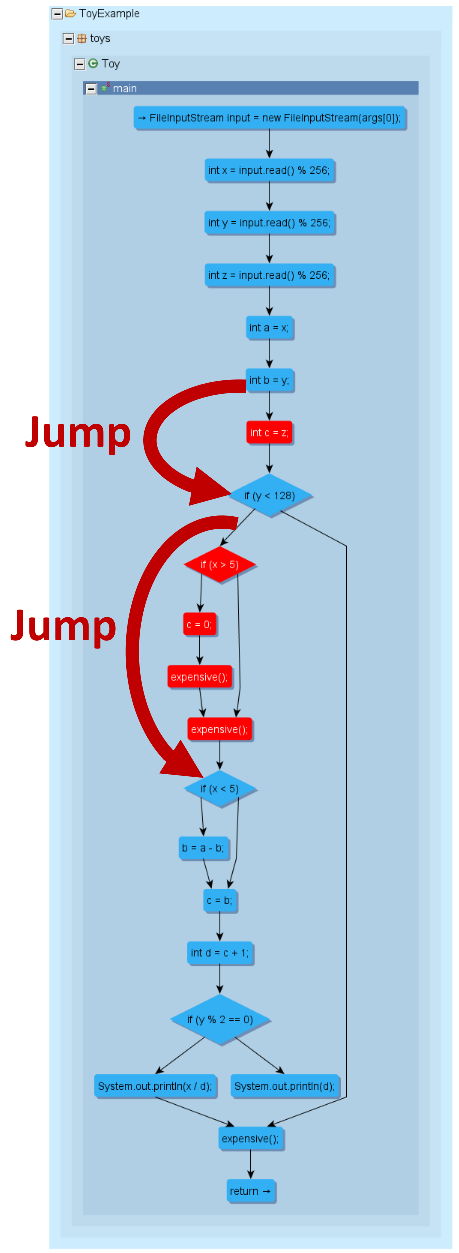
PCG(E), E=Division Statement

Program Modifications (2)

- Technique 2 - Eliding Irrelevant Statements
  - *Not strictly necessary*
  - Improves fuzzing speed
  - Program slice computes relevant control and data flow events
  - Elide irrelevant statements by injecting a pair of goto and label statements.
  - Specifically, for any edge in the PCG that is not in the CFG add a label before the successor node and a goto label statement after the predecessor node.



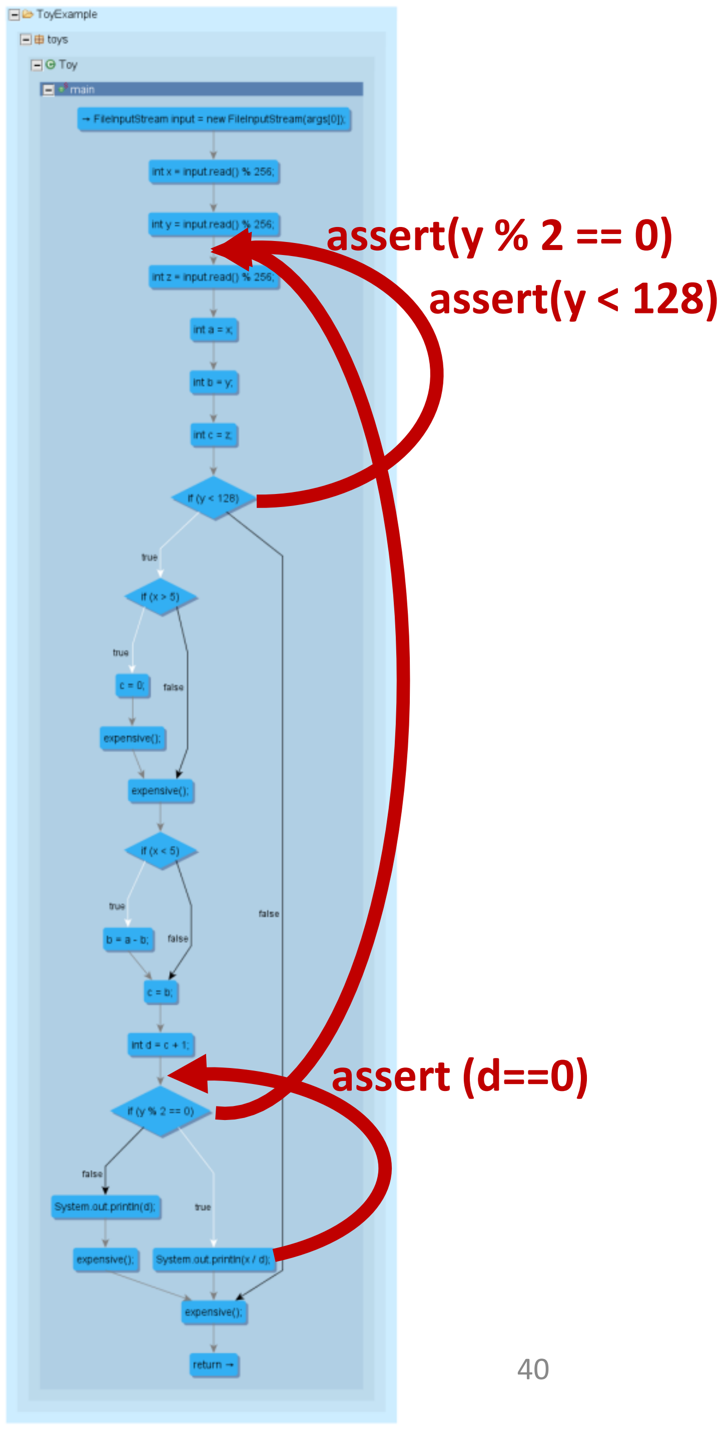
PCG of Relevant Statements



CFG with Irrelevant Statements Marked

Program Modifications (3)

- Technique 3 – Injecting Fail Early Assertions
  - *Not strictly necessary*
    - Can be used to further restrict relevance to a value at a statement. Example assert(d!=0) before the statement print(x / d).
  - Can also be used to improve fuzzing speed by preventing execution of relevant statements.
  - Specifically, for each condition in the PCG, insert an *assert-relevant(condition)* statement at the location of the last reaching definition of the condition variables.



```java
public class Toy {
   public static void main(String[] args) throws Exception {
      FileInputStream input = new FileInputStream(args[0]);
      int x = input.read() % 256; // x = -1 to 255
      int y = input.read() % 256; // y = -1 to 255
      int z = input.read() % 256; // z = -1 to 255

      int a = x;
      int b = y;
      int c = z;

      if(y < 128) {
         if(x > 5) {
            c = 0;
            expensive();
         }
         expensive();
         if(x < 5) {
            b = a - b;
         }
         c = b;
         int d = c + 1;
         if(y % 2 == 0) {
            System.out.println(x / d);
         } else {
            System.out.println(d);
            expensive();
         }
      }
      expensive();
   }
}
```

$\longrightarrow$

```java
public class Toy {
   public static void main(String[] args) throws Exception {
      FileInputStream input = new FileInputStream(args[0]);
      int x = input.read() % 256; // x = -1 to 255
      int y = input.read() % 256; // y = -1 to 255
      assert_relevant(y < 128 && y % 2 == 0);
      int z = input.read() % 256; // z = -1 to 255

      int a = x;
      int b = y;
      goto label_1;
      int c = z;

      label_1:
      if(y < 128) {
         goto label_2;
         if(x > 5) {
            c = 0;
            expensive();
         }
         expensive();
         label_2:
         if(x < 5) {
            b = a - b;
         }
         c = b;
         int d = c + 1;
         assert_relevant(d == 0);
         if(y % 2 == 0) {
            System.out.println(x / d);
            abort_relevant();
         } else {
            abort_irrelevant();
            System.out.println(d);
            expensive();
         }
      }
      abort_irrelevant();
      expensive();
   }
}
```

x	y	z	d	Division Executed	Outcome
1	2	*	0	Yes	Crash - Division by Zero
3	4	*	0	Yes	Crash - Division by Zero
5	6	*	7	Yes	No Crash - Failed Data Flow Constraint
5	-	-	0	No	No Crash - Failed Control Flow Constraint
-	-	-	1	Yes	No Crash - Failed Data Flow Constraint

Note: The lack of a file byte is denoted with a "–". A wildcard value is a "$*$" symbol.

Program	Fuzzing Time	Fuzzing Speed	Crashes
Original Program	15 minutes	2.93 executions/second	1
Modified Program	15 minutes	9.66 executions/second	2

Program	Original Program	Modified Program
Restrictions	None (all behaviors)	Homomorphic Behaviors
Detection Time	~1 hour (2218 traces)	< 1 second (2 traces)
Detected Invariants	• `x >= 0` • `y >= -1` • `z >= -1` • `x == a` • `b != d`	• $x \in \{1, 3\}$ • $y \in \{2, 4\}$ • `x <= y` • `x == a` • `y >= b` • `b == -1` • `c == -1` • `x >= d` • `y >= d` • `d == 0`

```java
public class Toy {
   public static void main(String[] args) throws Exception {
      FileInputStream input = new FileInputStream(args[0]);
      int x = input.read() % 256; // x = -1 to 255
      int y = input.read() % 256; // y = -1 to 255
      int z = input.read() % 256; // z = -1 to 255

      int a = x;
      int b = y;
      int c = z;

      if(y < 128) {
         if(x > 5) {
            c = 0;
            expensive();
         }
         expensive();
         if(x < 5) {
            b = a - b;
         }
         c = b;
         int d = c + 1;
         if(y % 2 == 0) {
            System.out.println(x / d);
         } else {
            System.out.println(d);
            expensive();
         }
      }
      expensive();
   }
}
```

x	y	z	d	Division Executed	Outcome
1	2	*	0	Yes	Crash - Division by Zero
3	4	*	0	Yes	Crash - Division by Zero
5	6	*	7	Yes	No Crash - Failed Data Flow Constraint
5	-	-	0	No	No Crash - Failed Control Flow Constraint
-	-	-	1	Yes	No Crash - Failed Data Flow Constraint

Note: The lack of a file byte is denoted with a "–". A wildcard value is a "$*$" symbol.

Program	Fuzzing Time	Fuzzing Speed	Crashes
Original Program	15 minutes	2.93 executions/second	1
Modified Program	15 minutes	9.66 executions/second	2

Program	Original Program	Modified Program
Restrictions	None (all behaviors)	Homomorphic Behaviors
Detection Time	~1 hour (2218 traces)	< 1 second (2 traces)
Detected Invariants	• x >= 0 • y >= -1 • z >= -1 • x == a • b != d	• $x \in \{1, 3\}$ • $y \in \{2, 4\}$ • x <= y • x == a • y >= b • b == -1 • c == -1 • x >= d • y >= d • d == 0

```java
public class Toy {
  public static void main(String[] args) throws Exception {
    FileInputStream input = new FileInputStream(args[0]);
    int x = input.read() % 256; // x = -1 to 255
    int y = input.read() % 256; // y = -1 to 255
    assert_relevant(y < 128 && y % 2 == 0);
    int z = input.read() % 256; // z = -1 to 255

    int a = x;
    int b = y;
    goto label_1;
    int c = z;

    label_1:
    if(y < 128) {
      goto label_2;
      if(x > 5) {
        c = 0;
        expensive();
      }
      expensive();
      label_2:
      if(x < 5) {
        b = a - b;
      }
      c = b;
      int d = c + 1;
      assert_relevant(d == 0);
      if(y % 2 == 0) {
        System.out.println(x / d);
        abort_relevant();
      } else {
        abort_irrelevant();
        System.out.println(d);
        expensive();
      }
    }
    abort_irrelevant();
    expensive();
  }
}
```

Case Study: BraidIt DARPA Challenge App

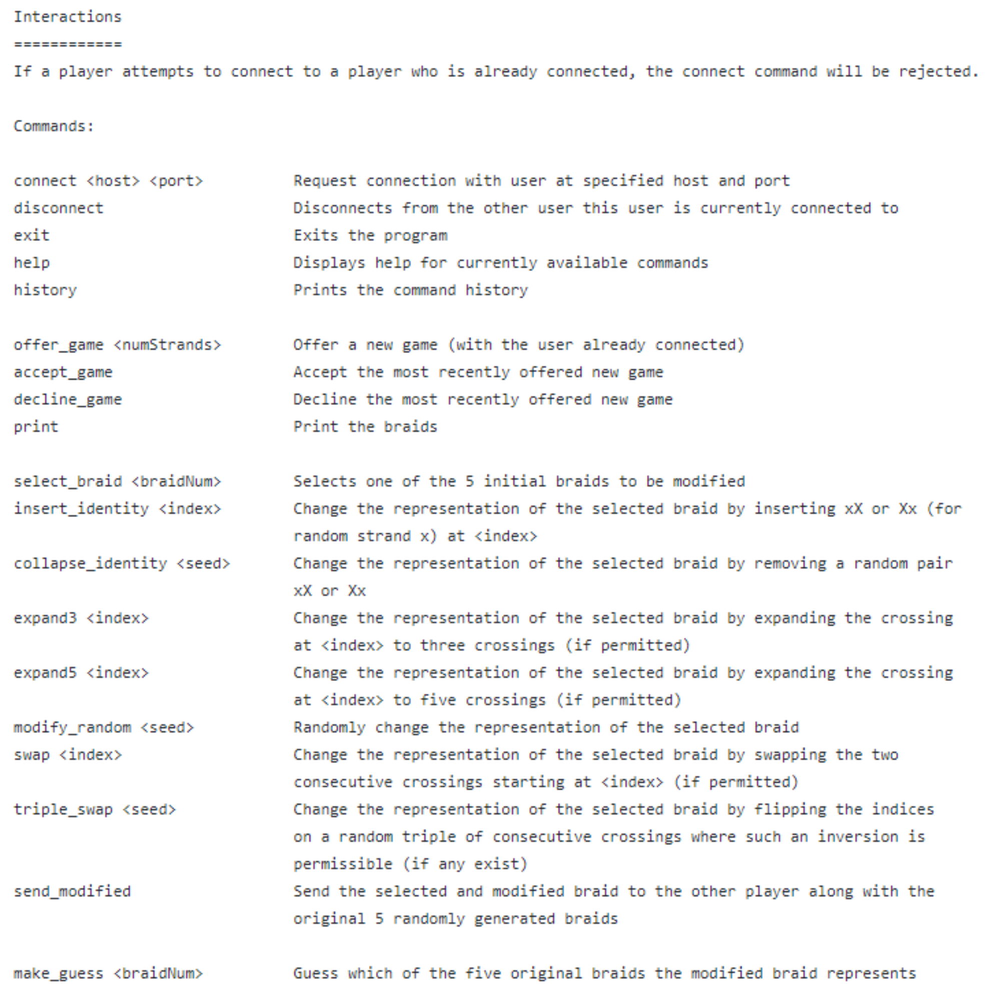
- A start to finish audit…

Case Study: BraidIt DARPA Challenge App

- Not *supposed* to be vulnerable…
  - But we didn’t know that…

Case Study: BraidIt DARPA Challenge App

- BraidIt is a peer-to-peer 2-player game that tests the players' ability to recognize topologically equivalent braids.
- BraidIt is based on the word equivalence problem in the Artin braid group. The application does all the dirty work, so users need not understand the theory and can treat it as a fun guessing game.



```
Interactions
============
If a player attempts to connect to a player who is already connected, the connect command will be rejected.

Commands:

connect <host> <port>         Request connection with user at specified host and port
disconnect                    Disconnects from the other user this user is currently connected to
exit                          Exits the program
help                          Displays help for currently available commands
history                       Prints the command history

offer_game <numStrands>       Offer a new game (with the user already connected)
accept_game                   Accept the most recently offered new game
decline_game                  Decline the most recently offered new game
print                         Print the braids

select_braid <braidNum>       Selects one of the 5 initial braids to be modified
insert_identity <index>       Change the representation of the selected braid by inserting xX or Xx (for
                              random strand x) at <index>
collapse_identity <seed>      Change the representation of the selected braid by removing a random pair
                              xX or Xx
expand3 <index>               Change the representation of the selected braid by expanding the crossing
                              at <index> to three crossings (if permitted)
expand5 <index>               Change the representation of the selected braid by expanding the crossing
                              at <index> to five crossings (if permitted)
modify_random <seed>          Randomly change the representation of the selected braid
swap <index>                  Change the representation of the selected braid by swapping the two
                              consecutive crossings starting at <index> (if permitted)
triple_swap <seed>            Change the representation of the selected braid by flipping the indices
                              on a random triple of consecutive crossings where such an inversion is
                              permissible (if any exist)
send_modified                 Send the selected and modified braid to the other player along with the
                              original 5 randomly generated braids

make_guess <braidNum>         Guess which of the five original braids the modified braid represents
```

Case Study: BraidIt DARPA Challenge App

- Is there an algorithmic complexity vulnerability in space that would cause the challenge program to store files with combined logical sizes that exceed the resource usage limit given the input budget?
- Input Budget: Maximum sum of the PDU sizes of the application requests sent from the attacker to the server: 2 kB (measured via sum of the length fields in tcpdump)
- Resource Usage Limit: Available Logical Size: 25 MB (logical size of output file measured with 'stat’)
- Probability of Success: 99%

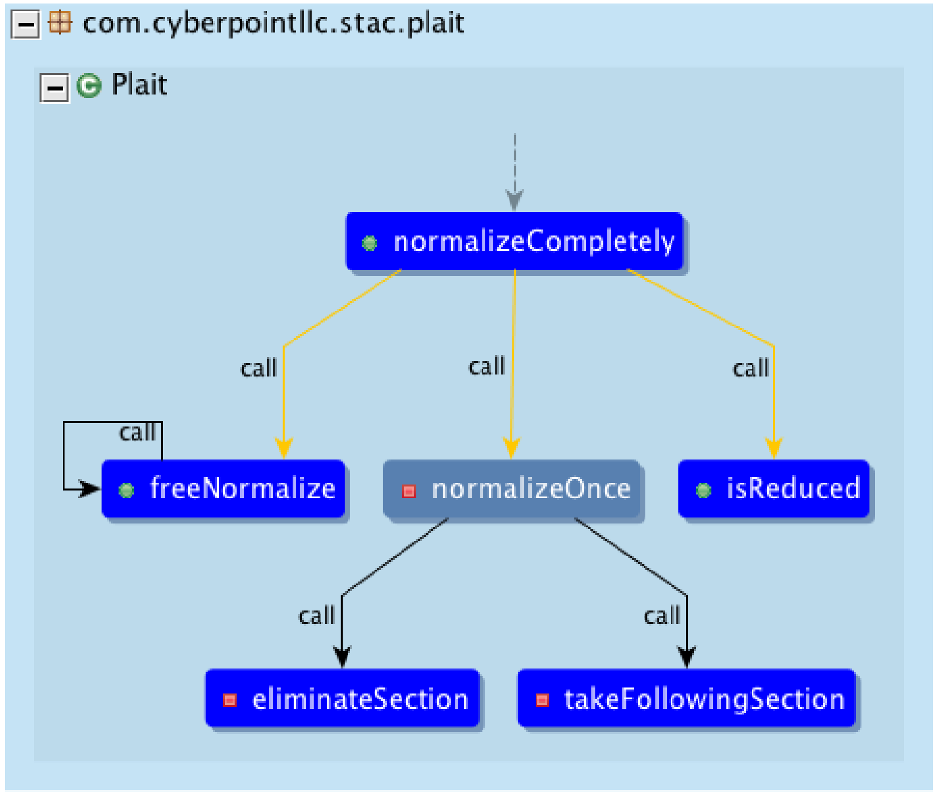
Example of Hardening

```
int i = 0;
while (i < charArray.length) {
    while (i < charArray.length && Math.random() < 0.4) {
```

Identification of an Interesting Loop

- *freeNormalize* and *normalizeOnce* each write to log file
- *normalizeCompletely* is an instance method that depends on existing program state
- The termination of *normalizeCompletely* depends on the result of *isReduced*
- Involves loop nesting and recursion

```
public void normalizeCompletely() {
    this.freeNormalize();
    while (!this.isReduced()) {
        this.normalizeOnce();
        this.freeNormalize();
    }
}
```



Identification of an Interesting Loop

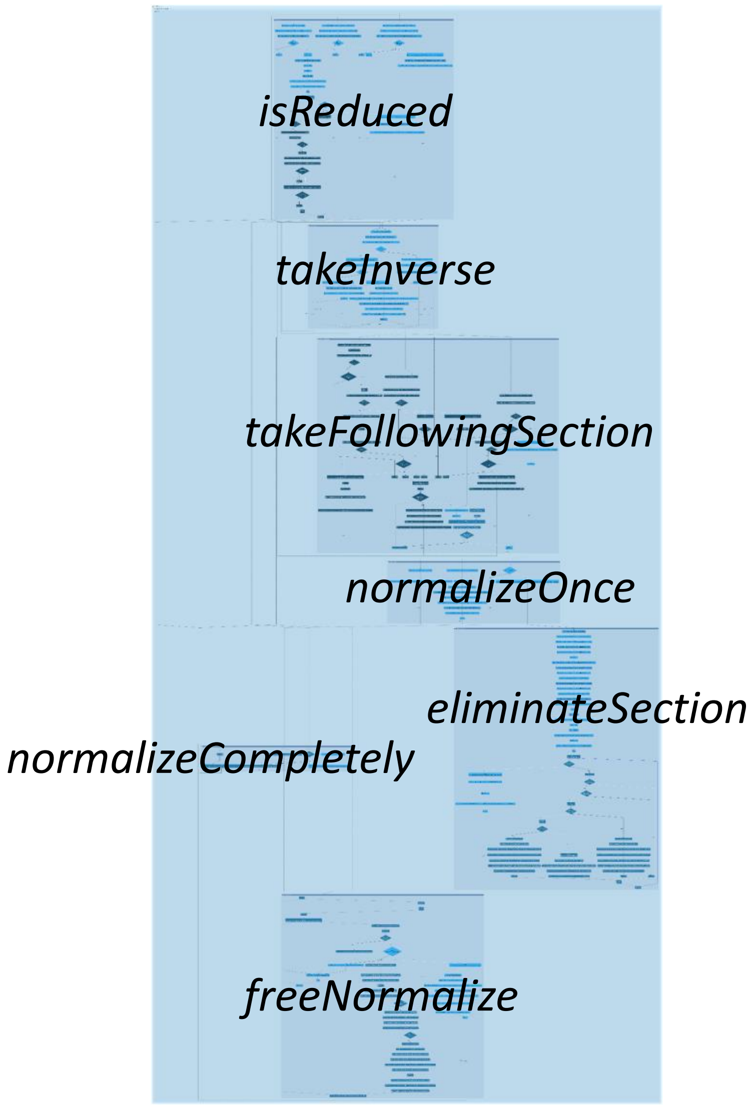
- *isReduced* reads a global variable called *intersections*
- *freeNormalize* and *normalizeOnce* update *intersections*
- *intersections* is a string variable that is initially attacker controlled

Hypothesis: *Can there be a string that has the property of being irreducible and therefore cause an infinite loop that writes to the file system?*

Complex String Operations

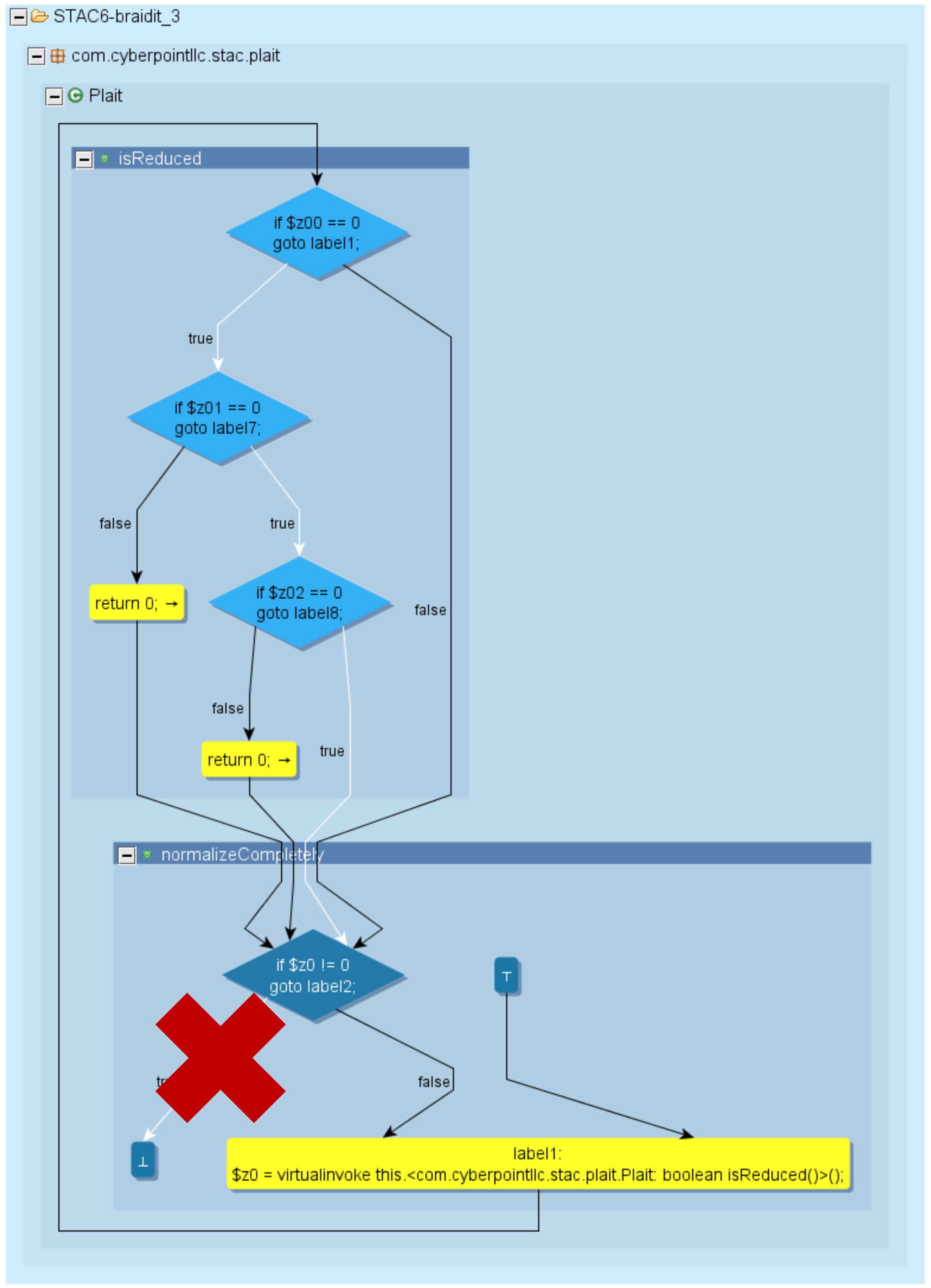
- 9 unique string operations in 62 locations
  - 20 of which are within loops
- 8 unique character level operations in 60 locations
  - 27 locations are within loops

Inter-procedural Control Flow Graph



Relevant Paths

- What do we care about?
  - The loop should not exit
  - isReduced() should always return false
  - If an input gets reduced then abort immediately


STAC6-braidit_3
com.cyberpointllc.stac.plait
Plait
isReduced
if $z00 == 0 goto label1;
true
if $z01 == 0 goto label7;
false
true
return 0;
if $z02 == 0 goto label8;
false
false
return 0;
true
normalizeCompletely
if $z0 != 0 goto label2;
⊤
false
⊥
label1:
$z0 = virtualinvoke this.<com.cyberpointllc.stac.plait.Plait: boolean isReduced()>();

Targeted Dynamic Analysis of *normalizeCompletely*

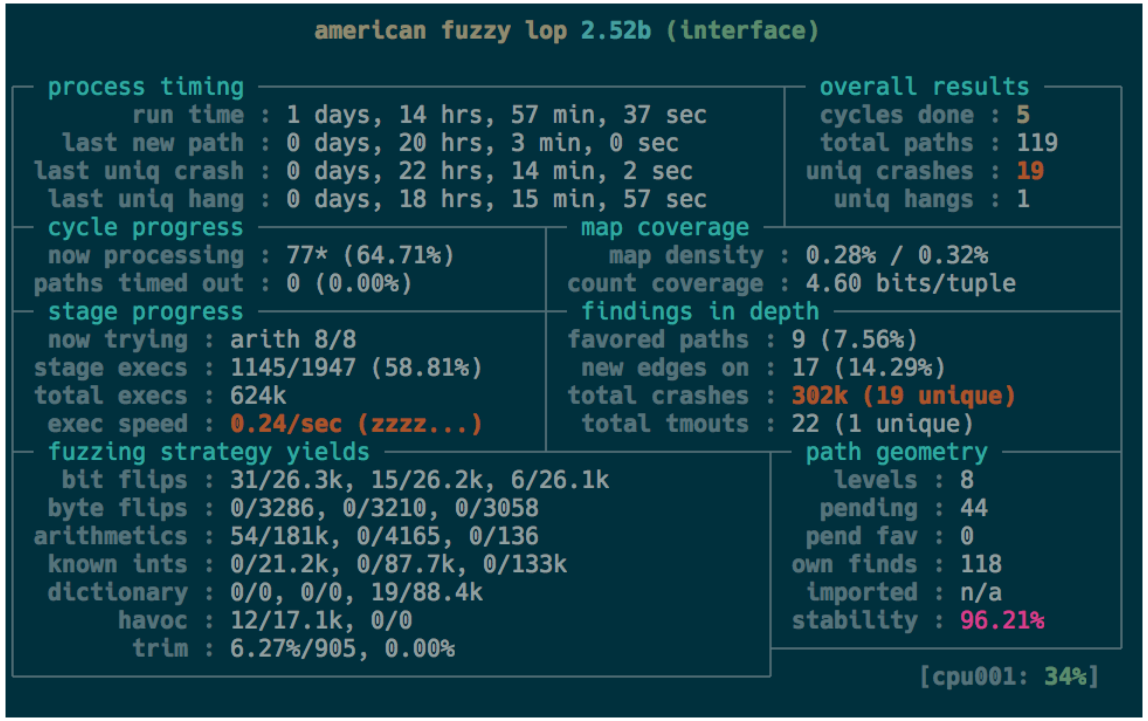
20 hours of directly fuzzing *normalizeCompletely…*

H. Invariant: *isReduced* is always false

H. Invariant: *intersections* is always a non-empty string

Input: “ËĎġčçęêďªã”

What does this mean?



```
                    american fuzzy lop 2.52b (interface)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 1 days, 14 hrs, 57 min, 37 sec      │  cycles done : 5      │
│   last new path : 0 days, 20 hrs, 3 min, 0 sec        │  total paths : 119    │
│ last uniq crash : 0 days, 22 hrs, 14 min, 2 sec       │ uniq crashes : 19     │
│  last uniq hang : 0 days, 18 hrs, 15 min, 57 sec      │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 77* (64.71%)      │    map density : 0.28% / 0.32%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 4.60 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 8/8             │ favored paths : 9 (7.56%)              │
│ stage execs : 1145/1947 (58.81%)    │  new edges on : 17 (14.29%)            │
│ total execs : 624k                  │ total crashes : 302k (19 unique)       │
│  exec speed : 0.24/sec (zzzz...)    │  total tmouts : 22 (1 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 31/26.3k, 15/26.2k, 6/26.1k            │    levels : 8          │
│  byte flips : 0/3286, 0/3210, 0/3058                 │   pending : 44         │
│ arithmetics : 54/181k, 0/4165, 0/136                 │  pend fav : 0          │
│  known ints : 0/21.2k, 0/87.7k, 0/133k               │ own finds : 118        │
│  dictionary : 0/0, 0/0, 19/88.4k                     │  imported : n/a        │
│       havoc : 12/17.1k, 0/0                          │ stability : 96.21%     │
│        trim : 6.27%/905, 0.00%                       ├────────────────────────┘
└──────────────────────────────────────────────────────┘          [cpu001: 34%]
```

Refined Experiment: Constrained Fuzzing

- Plait Constructor does some complex validation on *intersections*, which end with the following checks
  - Checks that each character is alphabetic
  - Checks that each character’s lowercase character is greater than 122 + *numStrands* + 2
  - *numStrands* is attacker controlled input between 8 and 27
- Experiment: Iterate over strings of the alphabet described by constructor
  - 20 minutes to find smallest malicious inputs +13 more…
- Minimal Input: “ªªª”

Refined Experiment: Homomorphic Invariants

H. Invariant: *isReduced* is always false

H. Invariant: *intersections* is always a non-empty string

H. Invariant: *intersections* contains a common subsequence of a single character ‘ª’

Refined Hypothesis: *A property of the character ‘ª’ can be used to create an irreducible string that causes an infinite loop that writes to the file system.*

Reasoning with Homomorphic Invariants

- Debug with the minimal input “ªªª” and pay attention character level operations
  - *freeNormalize* method removes a pair of case insensitive matching characters where one character is the first character in the string (leaving a single character ‘ª’ remaining)
  - *isReduced* method can return false if the string contains an uppercase character of a lowercase character
  - Uppercase(ª) == Lowercase(ª)
  - Fine scheme for ASCII, but Java Strings support Unicode UTF-16 standard
    - There are 395 UTF-16 characters that alphabetic and lowercase is their uppercase
    - Any could be used to craft an exploit

Conclusion

- Thesis: “Computing homomorphic program invariants is novel, useful, and practical.”
  - Novelty: Homomorphic program invariants – with respect to an equivalence class of control flow paths
  - Usefulness: Homomorphic program invariants can lead to stricter assertions that can be used to improve program comprehension
  - Practical: SIDIS framework demonstrates that homomorphic program invariants are computable
- Observation:
  - In a fuzzing based approach to computing homomorphic invariants the computational burden can be decreased by aborting on irrelevant paths

Questions?