

Demystifying Software Security

A Euler's Method Approach for Analyzing Complex Software

Ahmed Tamrawi



atamrawi



atamrawi.github.io



ahmedtamrawi@gmail.com

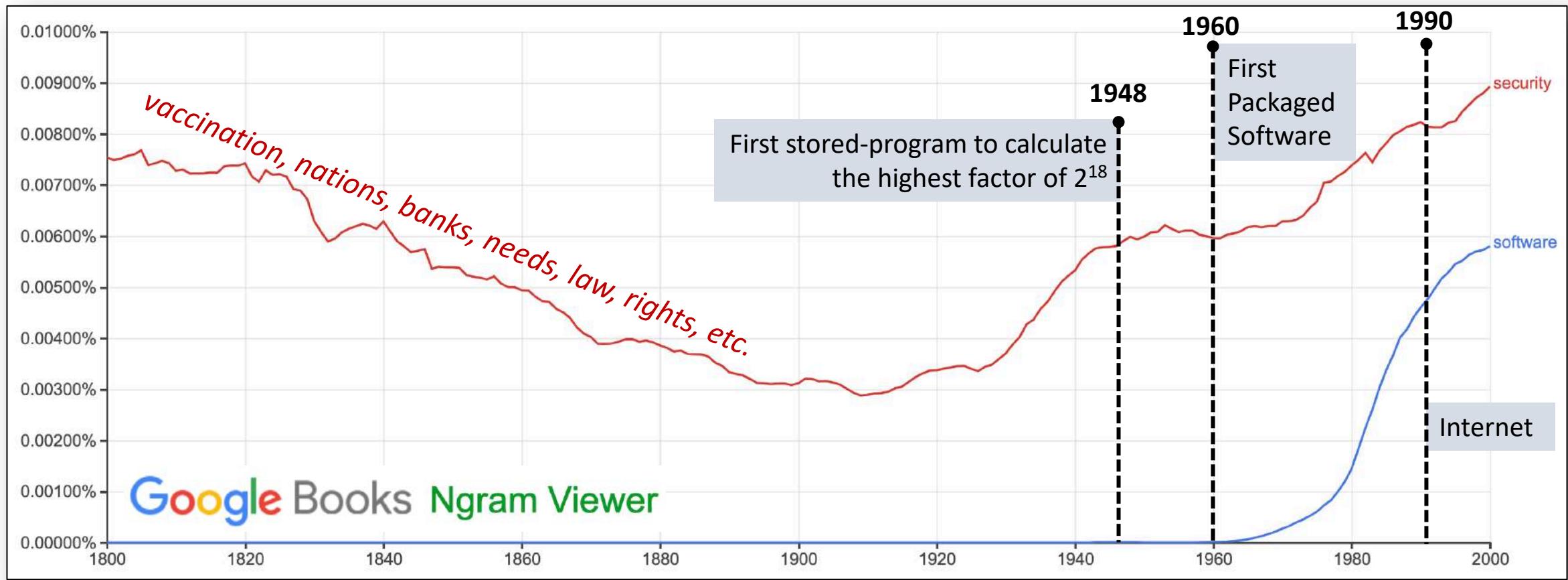
Acknowledgement: Team members at *Iowa State University* and *EnSoft, DARPA contracts FA8750- 12-2-0126 & FA8750-15-2-0080*

What is Software Security?

What is Software Security?

set of instructions, data or programs used to operate computers and execute specific tasks

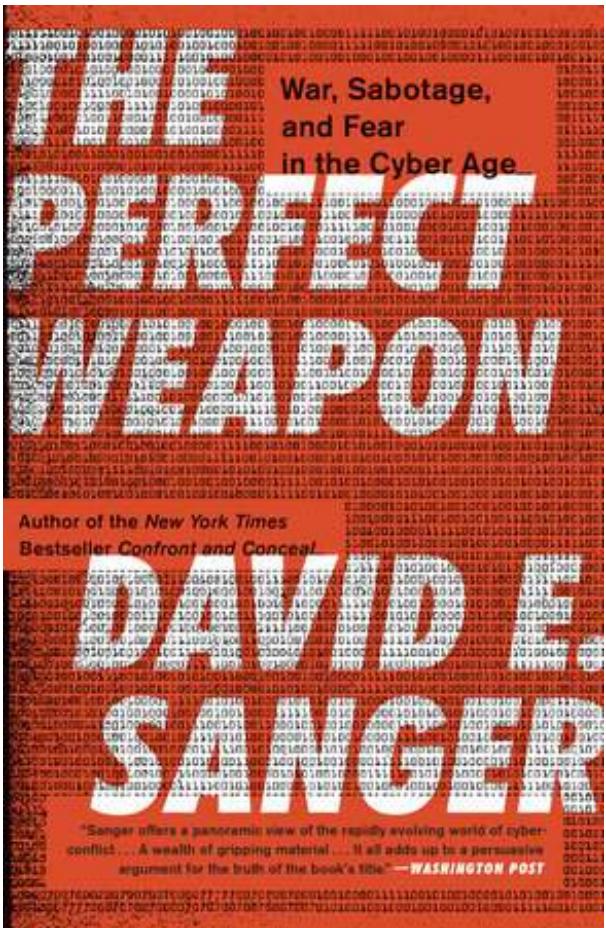
the state of being free from danger or threat



What is Software Security?

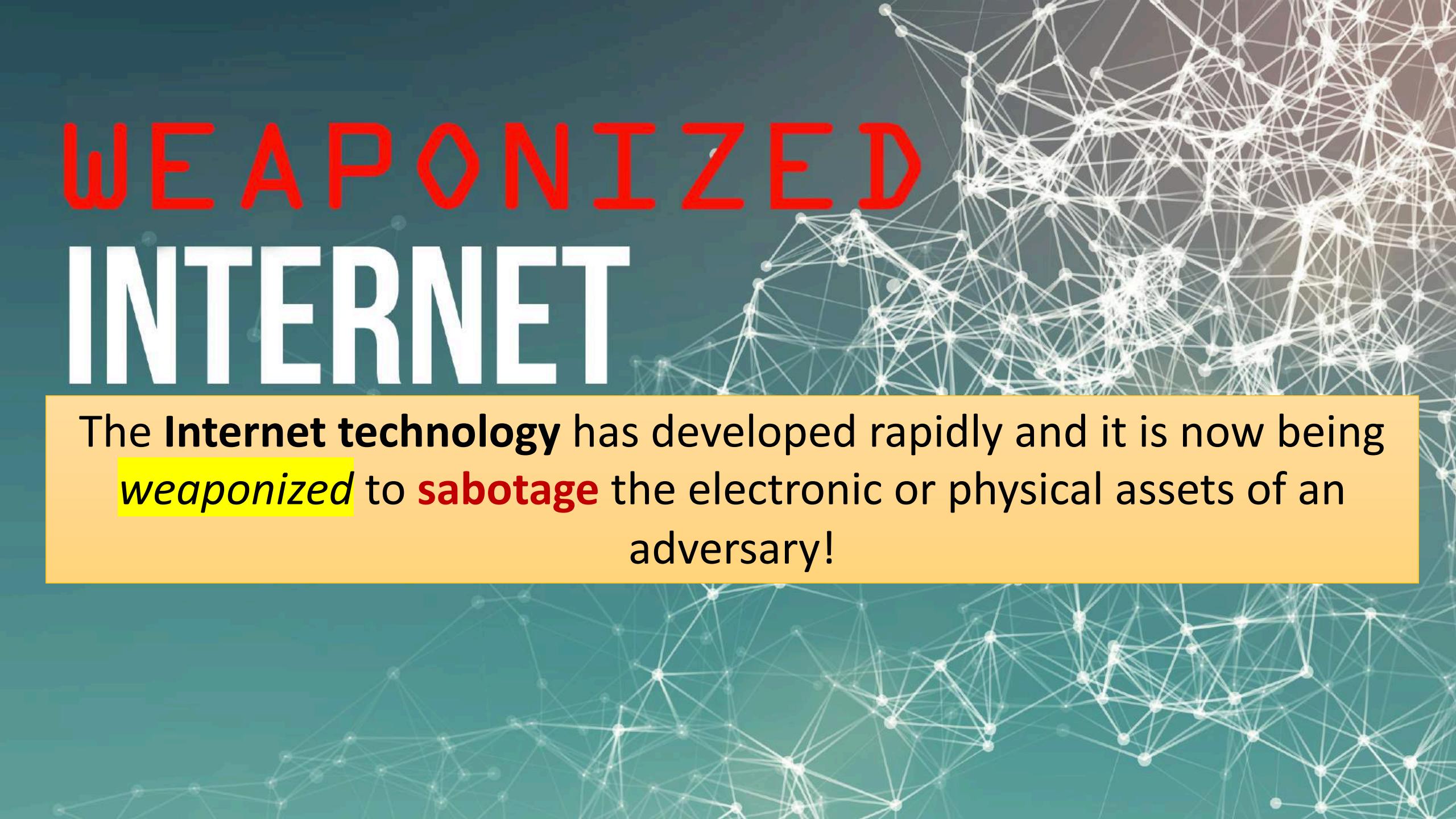
*is the umbrella term used to describe software that is engineered such that it **continues to function correctly under malicious attack***

Why do we need Software Security?

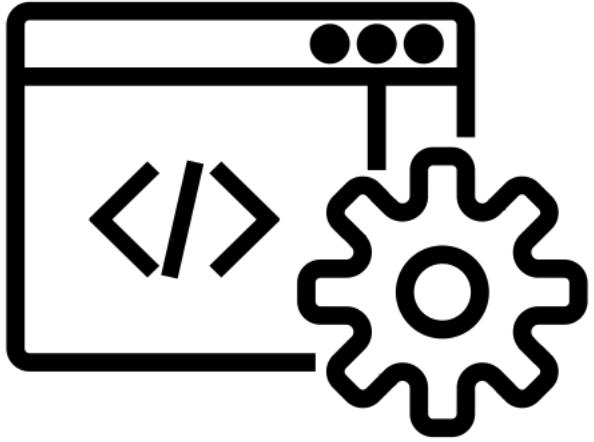


*"Each technology goes through a cycle of development and weaponization, **followed only later** by the formulation of doctrine and occasionally by efforts to control the weapon's use."*

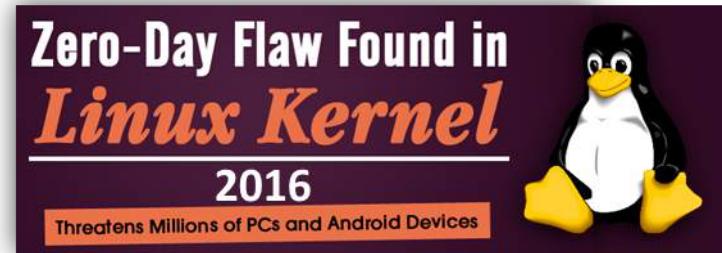
WEAPONIZED INTERNET

A complex network graph composed of numerous small white dots (nodes) connected by thin white lines (edges). The graph is highly interconnected, forming a dense web-like structure that serves as a visual metaphor for the global internet or a complex system.

The **Internet technology** has developed rapidly and it is now being *weaponized* to **sabotage** the electronic or physical assets of an adversary!

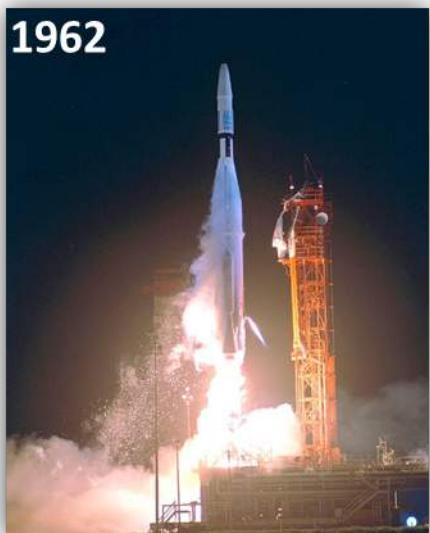


Software is an integral part of nearly all technology and almost all prominent attacks on cyber physical systems (CPS) have **exploited vulnerabilities** *rooted in the underlying systems software.*



Zero-Day Flaw Linux
Taking control and privacy

1962



NASA - Mariner 1
\$18 million



Android Lollipop

<https://threatpost.com/google-aware-of-memory-leakage-issue-in-android-5-1-fix-forthcoming/111640/>



Car Recalls - \$3 Billion



Knight Capital Trading
\$440 million

July 21, 2015



Jeep remotely hijacked

November 29, 2011



HP printers remotely set on fire

August 17, 2009



Destruction Sayano-Shushenskaya Hydroelectric Power Plant

Dec 2015 & Dec 2016



Ukraine power grid attacks

August 2003



Northeast Power Blackout

2018 COST OF CYBER CRIME

TOTAL COST

\$ **600 BILLION¹**

⌚ **\$1,138,888/minute**

🛡 **\$171,233/minute**
spend by business on information security²

🌐 Global, the cost of cybercrime on large business ranged from
11.7 MILLION/year³

⌚ Ranging from
\$222/minute

CYBERCRIME VICTIMS

⌚ **2.7 MILLION/day⁴**

⌚ **1,861/minute**



RANSOMWARE

costs to organizations

⌚ **\$8 BILLION/day⁵**

⌚ **\$15,221/minute⁵**

⌚ **1.5 organizations/minute** fall victim to ransomware attacks⁶

MALWARE

⌚ **1,274 new malware variants/minute⁷**

PHISHING EMAILS

⌚ **22.9 attacks/minute⁸**

RECORDS LEAKED

from publicly disclosed incidents

⌚ **2.9 BILLION/day⁹**

⌚ **5,518/minute**

Deployed in 2005, Identified in 2010



STUXnet Worm

August 2003



Davis-Besse Nuclear Power Plant



No need for bombs, *Plant Malware!*



*is investing billions of dollars
into Securing Software*

APAC

*Automated Program
Analysis for Cybersecurity*

VET

*Vetting Commodity IT
Software and Firmware*

HACMS

*High Assurance Cyber
Military Systems*

STAC

*Space/Time Analysis for
Cybersecurity*

CASE

*Cyber Assured Systems
Engineering*

CHESS

*Computers and Humans
Exploring Software Security*

ARCOS

*Automated Rapid
Certification Of Software*

WELL THEN

LET'S SECURE OUR SOFTWARE!



Block User Input!

Sanitize User Input!



Brought to you by **Dettol**,
kills all germs except 0.01%

How to tell if my Software *is* Secure?

Hire A Cybersecurity Engineer!

The image displays two side-by-side screenshots of job listings from professional networking sites.

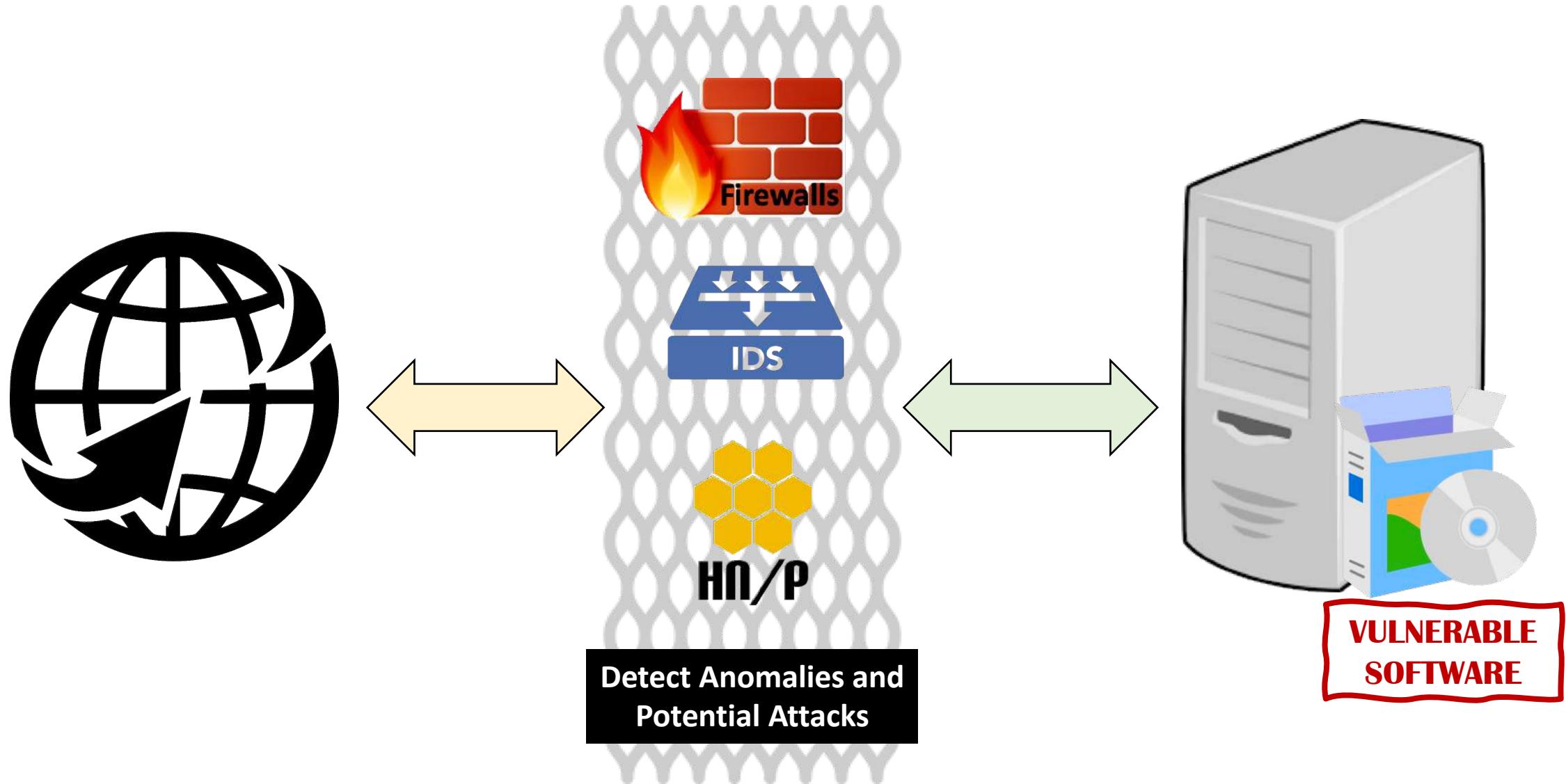
Top Listing: Ford Motor Company is hiring a Cyber Security Engineer in Dearborn, MI, US. The job requires an analytical mind and a detailed understanding of cybersecurity methodologies. Responsibilities include maintaining security walls, VPNs, Data, and upgrading security systems. The listing shows 48 applicants and 1000+ employees at the company. It was posted 2 years ago with 368 views. Buttons for 'Save' and 'Apply' are visible.

Bottom Listing: Adobe is hiring a Security Software Engineer in Lehi, UT, US. The job involves providing technical cybersecurity expertise for mobility products, services, and applications. Responsibilities include developing and improving security products and performing assessments. The listing shows 4 applicants and 1000+ employees at the company. It was posted 1 day ago with 17 views. Buttons for 'Save' and 'Apply' are visible.

Cybersecurity engineers perform a number of functions including architecting, developing and fielding secure network solutions to protect against advanced persistent threats, developing/engineering trusted systems into secure systems, performing assessments and penetration testing, and managing security technology and audit/intrusion systems.

A typical description for cybersecurity engineering jobs!

Cybersecurity Engineering





Software Analysis

Find potential vulnerabilities in software that could result on unintended behavior (fatal error, denial of service, etc.)

WELL THEN

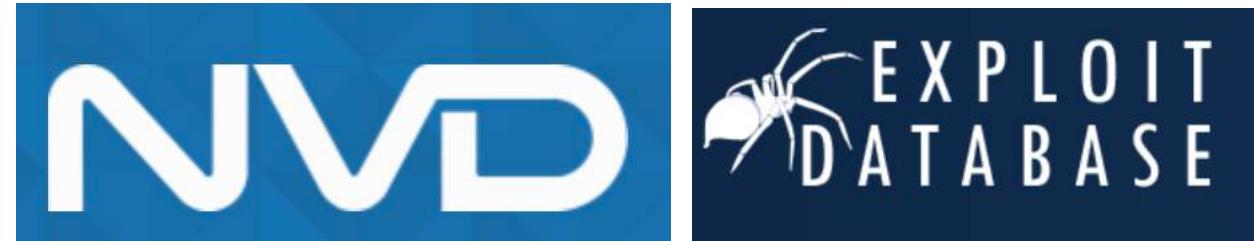
**LET'S ANALYZE
OUR SOFTWARE!**

Software Analysis

①

Choose the security property you want to prove its conformance or the security vulnerability you want to prove its absence.

Rank	ID	Name	Score
[1]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	75.56
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.69
[3]	CWE-20	Improper Input Validation	43.61
[4]	CWE-200	Information Exposure	32.12
[5]	CWE-125	Out-of-bounds Read	26.53
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	24.54
[7]	CWE-416	Use After Free	17.94
[8]	CWE-190	Integer Overflow or Wraparound	17.35
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	15.54
[10]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.10
[11]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.47
[12]	CWE-787	Out-of-bounds Write	11.08
2019 Top 25 Vulnerabilities			
[16]	CWE-434	Unrestricted Upload of File with Dangerous Type	5.50
[17]	CWE-611	Improper Restriction of XML External Entity Reference	5.48
[18]	CWE-94	Improper Control of Generation of Code ('Code Injection')	5.36
[19]	CWE-798	Use of Hard-coded Credentials	5.12
[20]	CWE-400	Uncontrolled Resource Consumption	5.04
[21]	CWE-772	Missing Release of Resource after Effective Lifetime	5.04
[22]	CWE-426	Untrusted Search Path	4.40
[23]	CWE-502	Deserialization of Untrusted Data	4.30
[24]	CWE-269	Improper Privilege Management	4.23
[25]	CWE-295	Improper Certificate Validation	4.06



CVE Details
The ultimate security vulnerability datasource

VULDB
THE CROWD-BASED VULNERABILITY DATABASE

Software Analysis

②

Pick a **software analysis strategy** or a combination of strategies to verify property conformance or vulnerability absence on each **feasible execution path**.

*What is a **feasible execution path**?*

What are software analysis strategies?

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Division-By-Zero (DBZ) Vulnerability?

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Division-By-Zero (DBZ) Vulnerability?

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Division-By-Zero (DBZ) Vulnerability?

No DBZ Vulnerability!

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

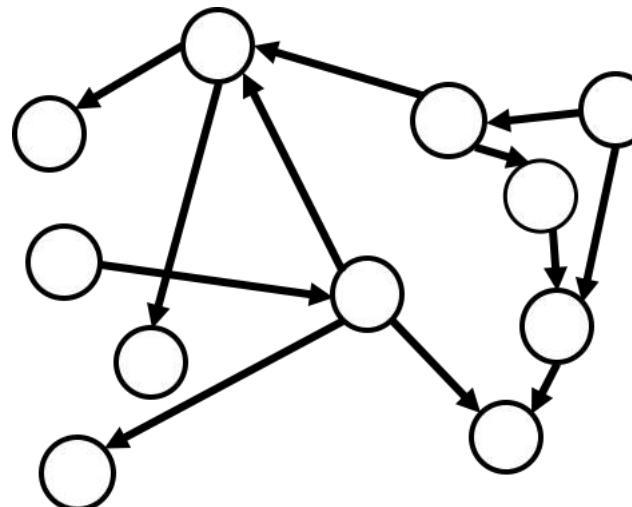
How to encode programs into
machine-comprehensible format to
enable software analysis?

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

*Think of software as a **Graph***



What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

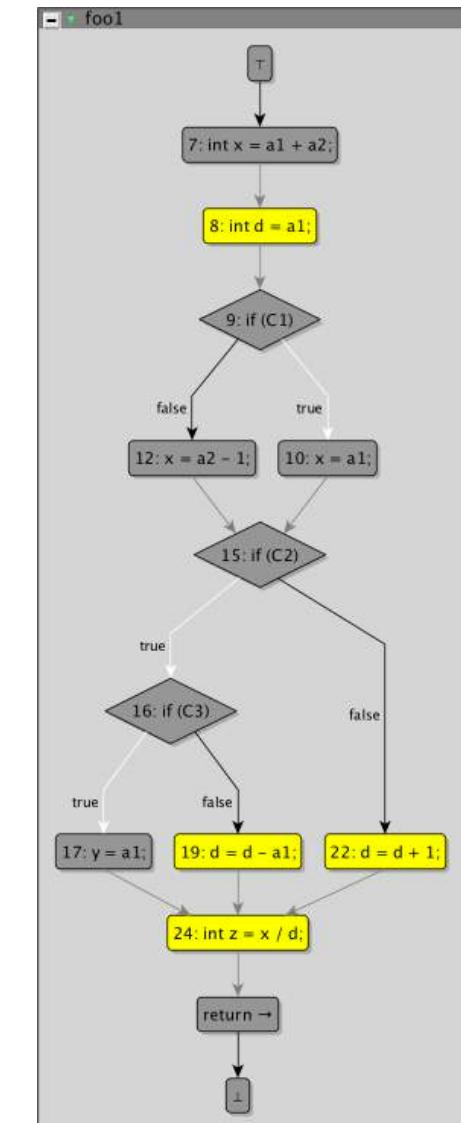
```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Transform into meaningful graph

A Node corresponds to a **Code Statement**

An Edge corresponds to the **control flow** from one statement to its successor in flow

Diamond Nodes are nodes corresponding to **conditional statements**



Function foo1

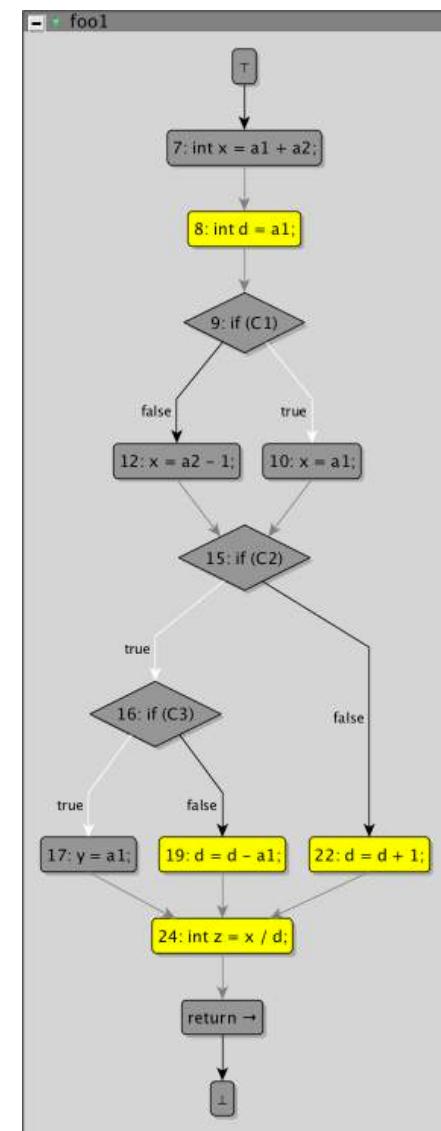
Control Flow Graph (CFG)

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Function foo1



Control Flow Graph (CFG)

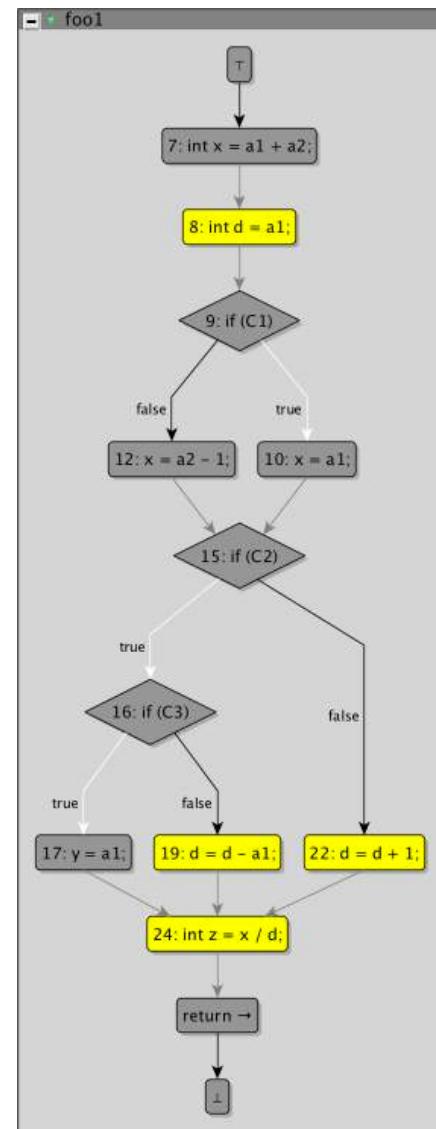
Division-By-Zero (DBZ) Vulnerability?

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Function foo1



Control Flow Graph (CFG)

Division-By-Zero (DBZ) Vulnerability?

Six Possible Execution Paths

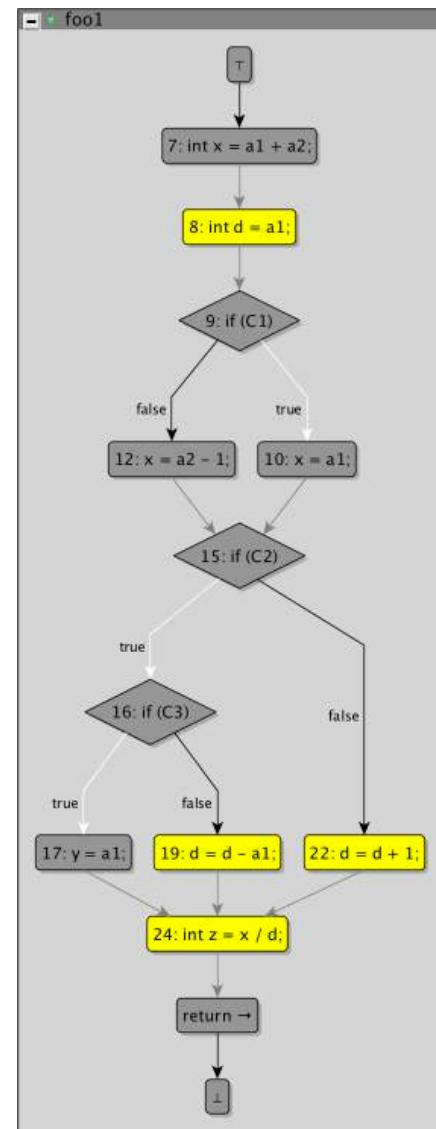
- | |
|----------------------------------------------------------------|
| $B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c}_3], 19, 24$ |
| $B_2 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[\bar{c}_3], 19, 24$ |
| $B_3 : 7, 8, 9[c_1], 10, 15[\bar{c}_2], 22, 24$ |
| $B_4 : 7, 8, 9[\bar{c}_1], 12, 15[\bar{c}_2], 22, 24$ |
| $B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$ |
| $B_6 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[c_3], 17, 24$ |

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Function foo1



Control Flow Graph (CFG)

Division-By-Zero (DBZ) Vulnerability?

Six Possible Execution Paths

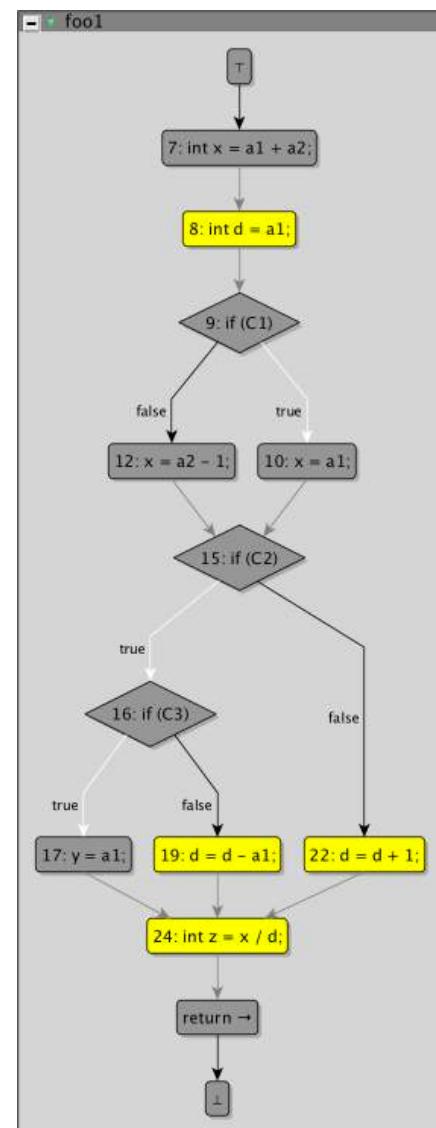
✗	$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c}_3], 19, 24$
✗	$B_2 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[\bar{c}_3], 19, 24$
✗	$B_3 : 7, 8, 9[c_1], 10, 15[\bar{c}_2], 22, 24$
✗	$B_4 : 7, 8, 9[\bar{c}_1], 12, 15[\bar{c}_2], 22, 24$
✗	$B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$
✗	$B_6 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[c_3], 17, 24$

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Function foo1



Control Flow Graph (CFG)

Division-By-Zero (DBZ) Vulnerability?

Six Possible Execution Paths

✗	$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c}_3], 19, 24$
✗	$B_2 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[\bar{c}_3], 19, 24$
✗	$B_3 : 7, 8, 9[c_1], 10, 15[\bar{c}_2], 22, 24$
✗	$B_4 : 7, 8, 9[\bar{c}_1], 12, 15[\bar{c}_2], 22, 24$
✗	$B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$
✗	$B_6 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[c_3], 17, 24$

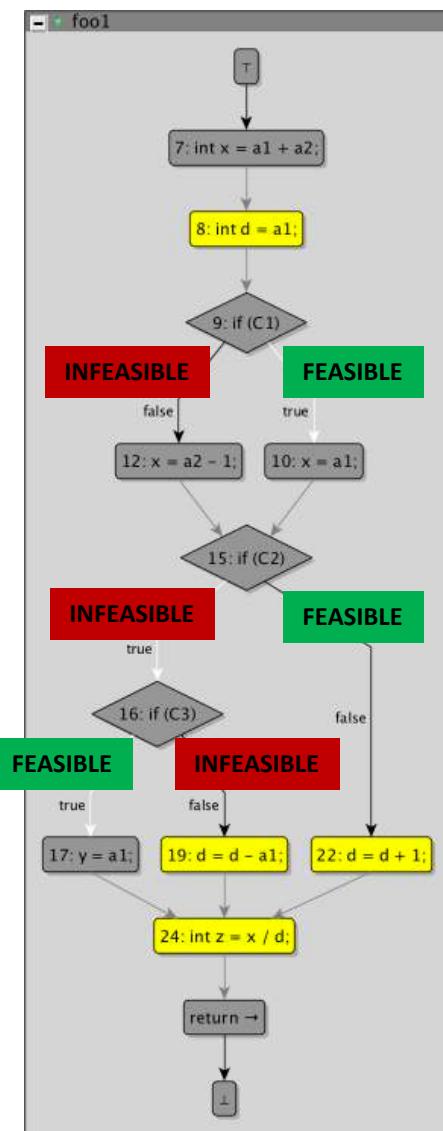
Are all **feasible**, in other words, are all could be executed at run-time?

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Function foo1



Control Flow Graph (CFG)

Division-By-Zero (DBZ) Vulnerability?

Six Possible Execution Paths

INFEASIBLE	$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c}_3], 19, 24$	X
INFEASIBLE	$B_2 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[\bar{c}_3], 19, 24$	X
FEASIBLE	$B_3 : 7, 8, 9[c_1], 10, 15[\bar{c}_2], 22, 24$	X
INFEASIBLE	$B_4 : 7, 8, 9[\bar{c}_1], 12, 15[\bar{c}_2], 22, 24$	X
INFEASIBLE	$B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$	X
INFEASIBLE	$B_6 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[c_3], 17, 24$	X

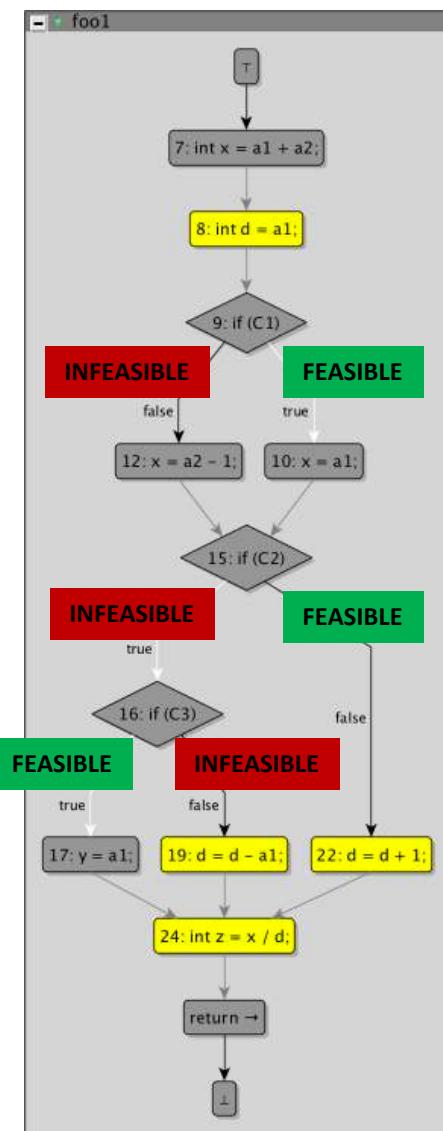
Are all **feasible**, in other words, are all could be executed at run-time?

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Function foo1



Control Flow Graph (CFG)

Division-By-Zero (DBZ) Vulnerability?

Six Possible Execution Paths

INFEASIBLE	$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c}_3], 19, 24$	Safe
INFEASIBLE	$B_2 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[\bar{c}_3], 19, 24$	Safe
FEASIBLE	$B_3 : 7, 8, 9[c_1], 10, 15[\bar{c}_2], 22, 24$	✗
INFEASIBLE	$B_4 : 7, 8, 9[\bar{c}_1], 12, 15[\bar{c}_2], 22, 24$	Safe
INFEASIBLE	$B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$	Safe
INFEASIBLE	$B_6 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[c_3], 17, 24$	Safe

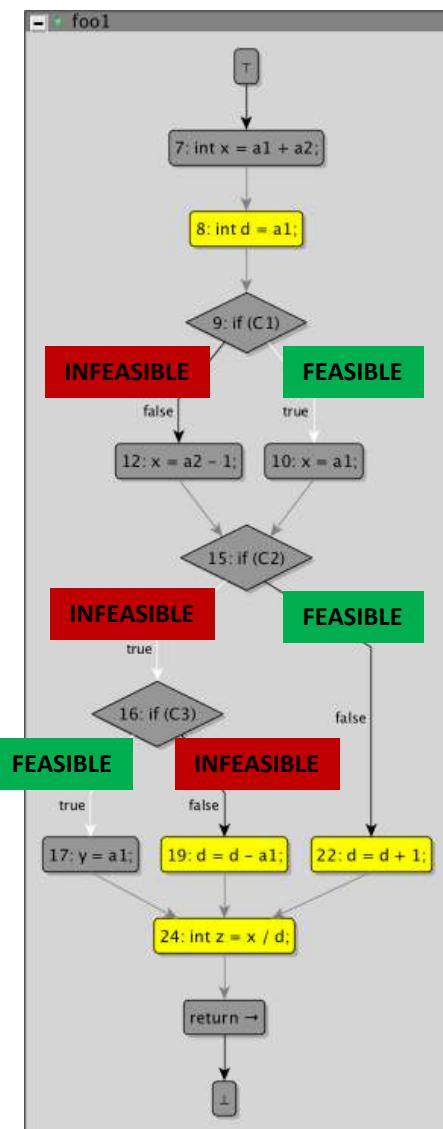
Check if values propagated on feasible
vulnerable path can result on DBZ?

What is a feasible execution path?

A path of statements in software that can be taken on an actual run of the software

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Function foo1



Control Flow Graph (CFG)

Division-By-Zero (DBZ) Vulnerability?

No DBZ Vulnerability!

Six Possible Execution Paths

INFEASIBLE	$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c}_3], 19, 24$	Safe
INFEASIBLE	$B_2 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[\bar{c}_3], 19, 24$	Safe
FEASIBLE	$B_3 : 7, 8, 9[c_1], 10, 15[\bar{c}_2], 22, 24$	Safe
INFEASIBLE	$B_4 : 7, 8, 9[\bar{c}_1], 12, 15[\bar{c}_2], 22, 24$	Safe
INFEASIBLE	$B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$	Safe
INFEASIBLE	$B_6 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[c_3], 17, 24$	Safe

Software Analysis

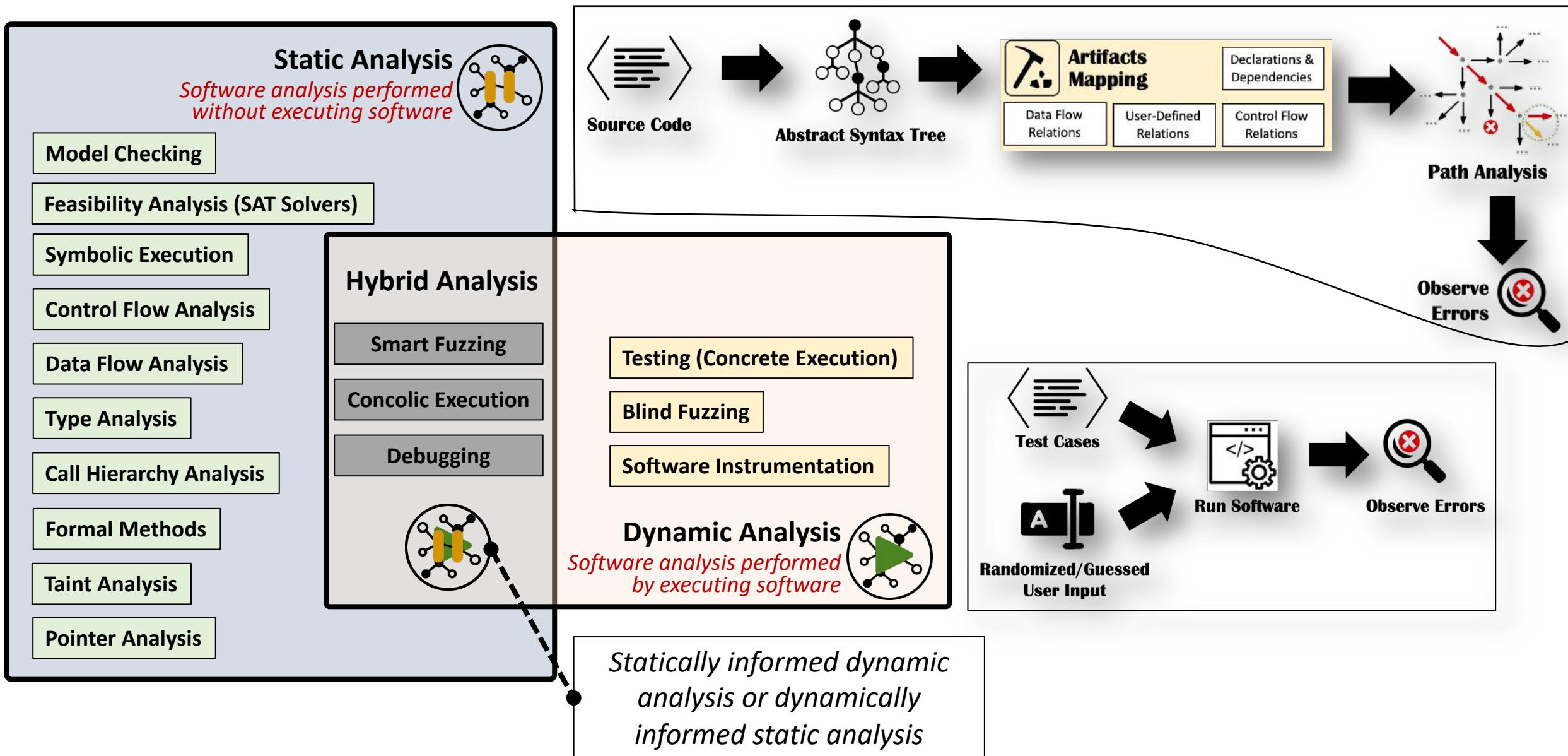
②

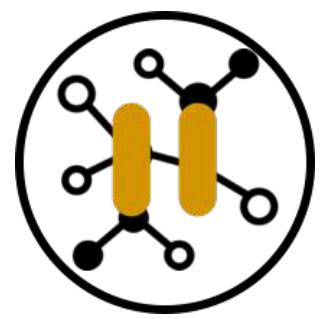
Pick a **software analysis strategy** or a combination of strategies to verify property conformance or vulnerability absence on each **feasible execution path**.

*What is a **feasible execution path**?*

What are software analysis strategies?

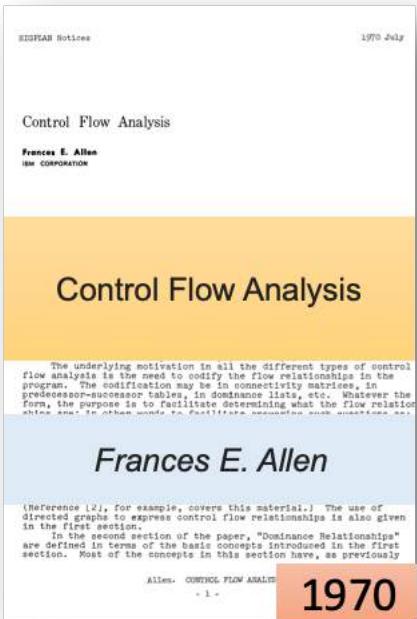
Software Analysis Strategies





Static Analysis: Control Flow Analysis

determines the order of program statements in a given source code, and predict and specify the set of execution traces



```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Function foo1

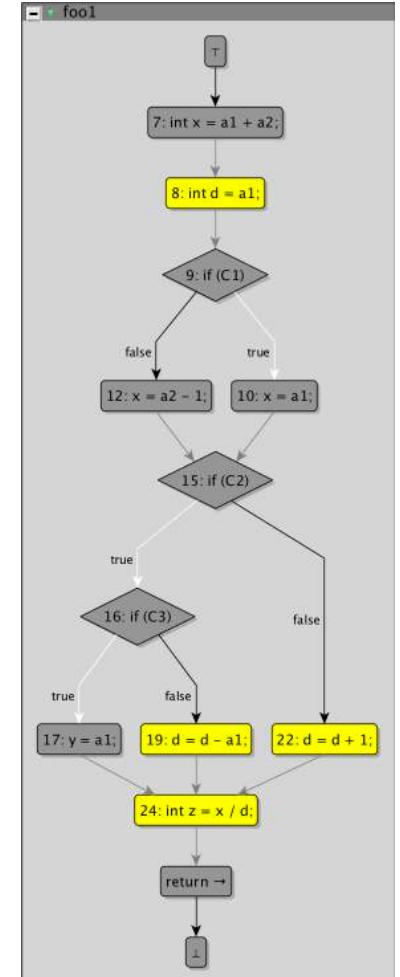


A Node corresponds to a Code Statement

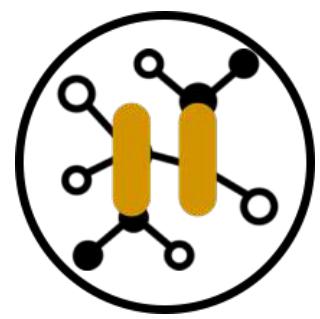
An Edge corresponds to the control flow
from one statement to its successor

Diamond Nodes are nodes corresponding
to conditional statements

It is not straightforward process!



Control Flow Graph (CFG)

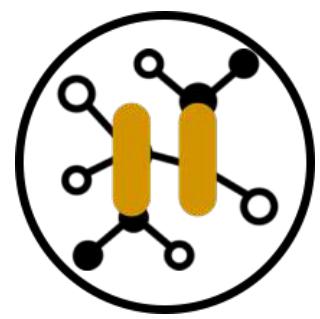


Static Analysis: Data Flow Analysis

gathers information about the possible set of values calculated at various points in a computer program

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d; X
25 }
```

How did we identify the statements at lines 8, 19, and 22 to be relevant to for the analysis of **the potential DBZ at line 24?**

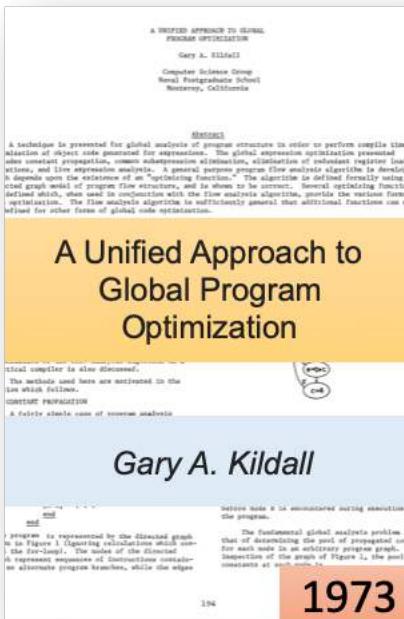


Static Analysis: Data Flow Analysis

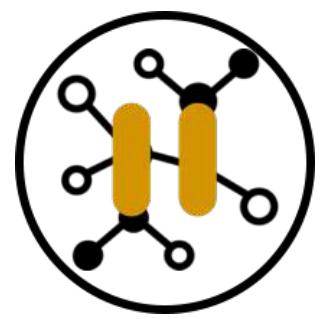
gathers information about the possible set of values calculated at various points in a computer program

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d; X
25 }
```

How did we identify the statements at lines 8, 19, and 22 to be relevant to for the analysis of **the potential DBZ at line 24?**



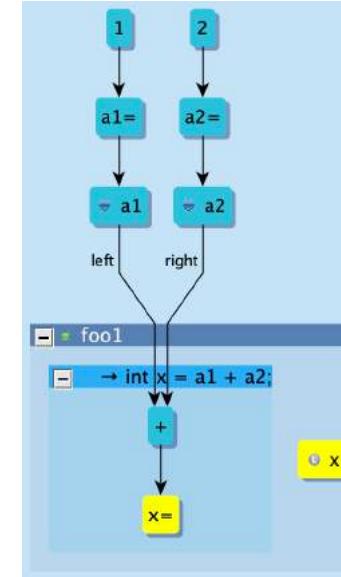
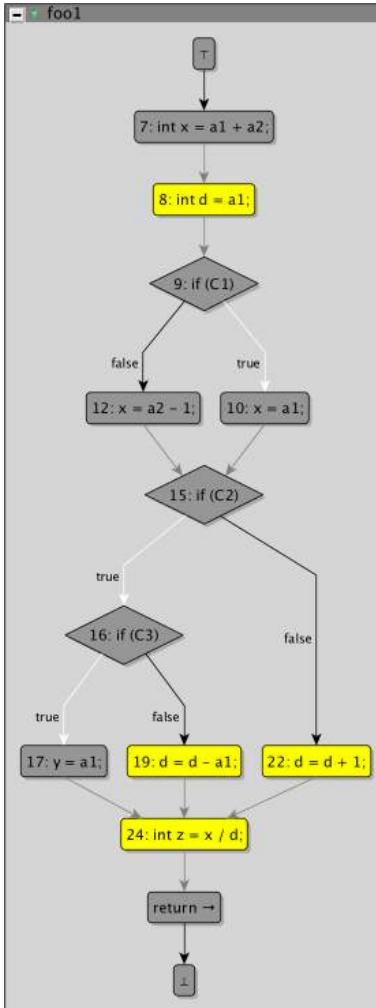
*set up **data-flow equations** for each node of the **control flow graph** and solve them by repeatedly calculating the output from the input locally at each node until the whole system/program stabilizes (reaches a fixpoint)*



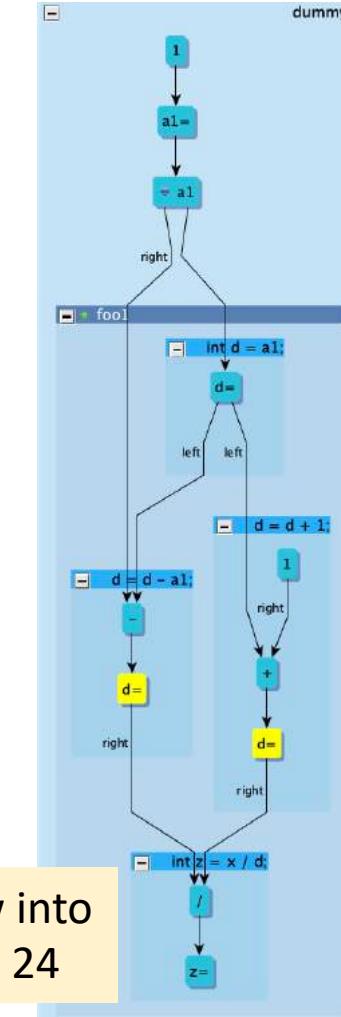
Static Analysis: Data Flow Analysis

gathers information about the possible set of values calculated at various points in a computer program

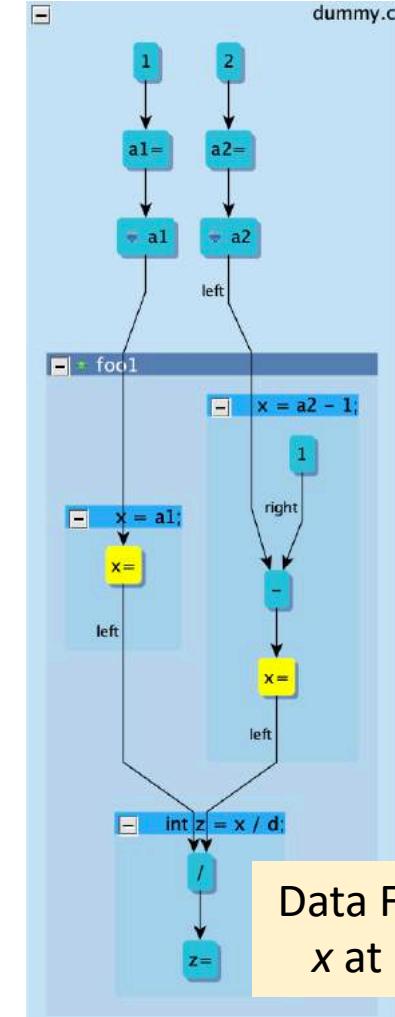
```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d; X
25 }
```



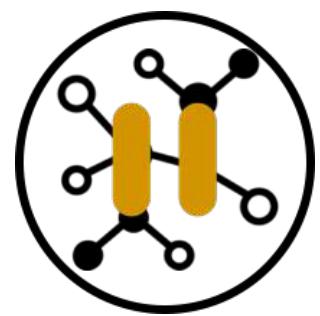
Data Flow into
x at line 7



Data Flow into
d at line 24



Data Flow into
x at line 24

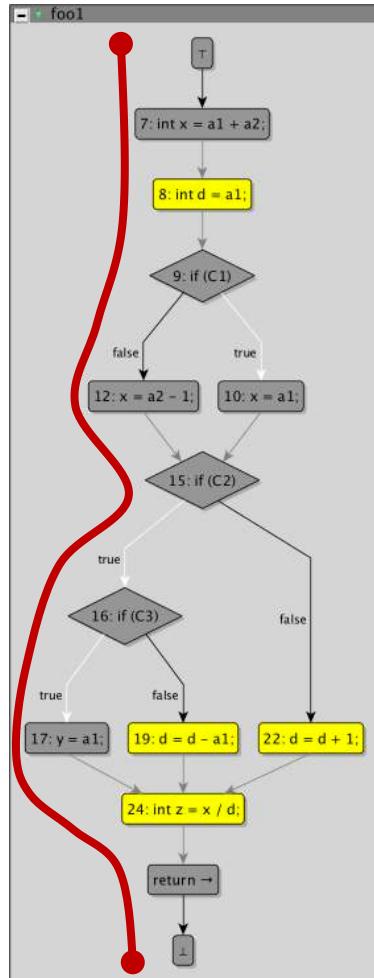


Static Analysis: Feasibility Analysis

*determines whether a given path is feasible (could be taken on actual run)
based on the associated Boolean path formula*

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Function foo1



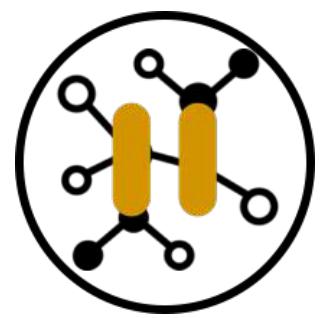
NOT-SATISFIABLE

Path Boolean Formula: $\overline{C1} \cap C2 \cap C3$

The **satisfiability modulo theories (SMT)** problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality.



For updated list of SMT solvers:
https://en.wikipedia.org/wiki/Satisfiability_modulo_theories#Solvers

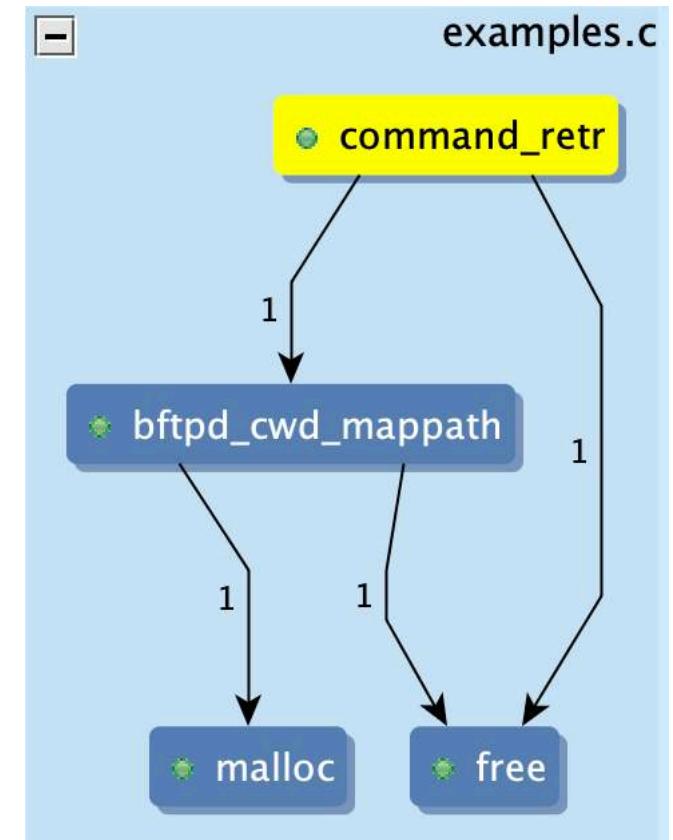


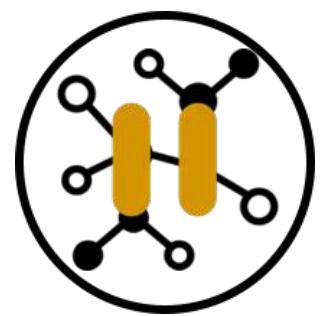
Static Analysis: Call Hierarchy Analysis

is a sub control flow analysis technique to mine call relations between functions.

The resultant graph is called the call graph

```
1 void bftpd_cwd_mappath() {  
2     char *result = malloc();  
3     if(!result) {  
4         return;  
5     }  
6     if(!path2) {  
7         free();  
8     }  
9 }  
10  
11 void command_retr() {  
12     bftpd_cwd_mappath();  
13     free();  
14 }
```





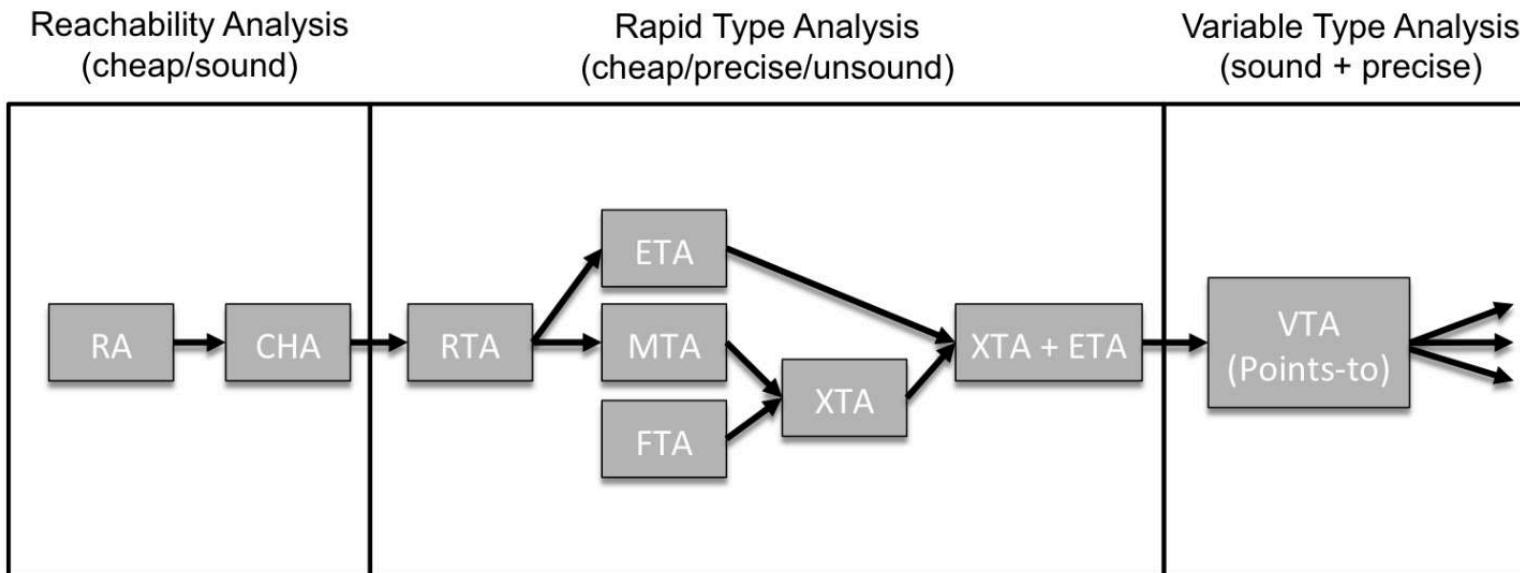
Static Analysis: Call Hierarchy Analysis

is a sub control flow analysis technique to mine call relations between functions.

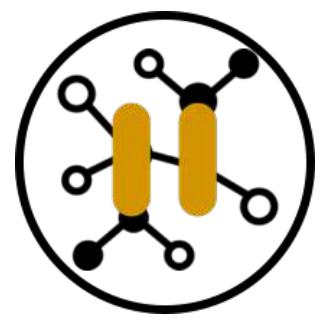
The resultant graph is called the call graph

Many Complications! think about *obscure control flows*:

- Event-driven in Web Frameworks.
- Dynamic Dispatch (e.g., Function pointer, polymorphism, overriding, etc.)



Check Ben Holland's blog explaining all the details:
<https://ben-holland.com/call-graph-construction-algorithms-explained/>

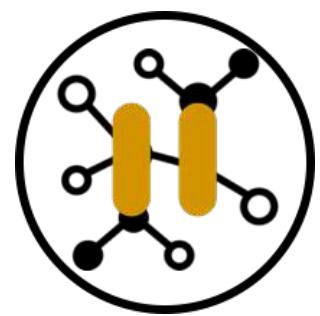


Static Analysis: Symbolic Execution

is a technique where an interpreter follows the program, assuming symbolic values for inputs, a case of abstract interpretation. Thus performing operations on symbolic values abstractly.

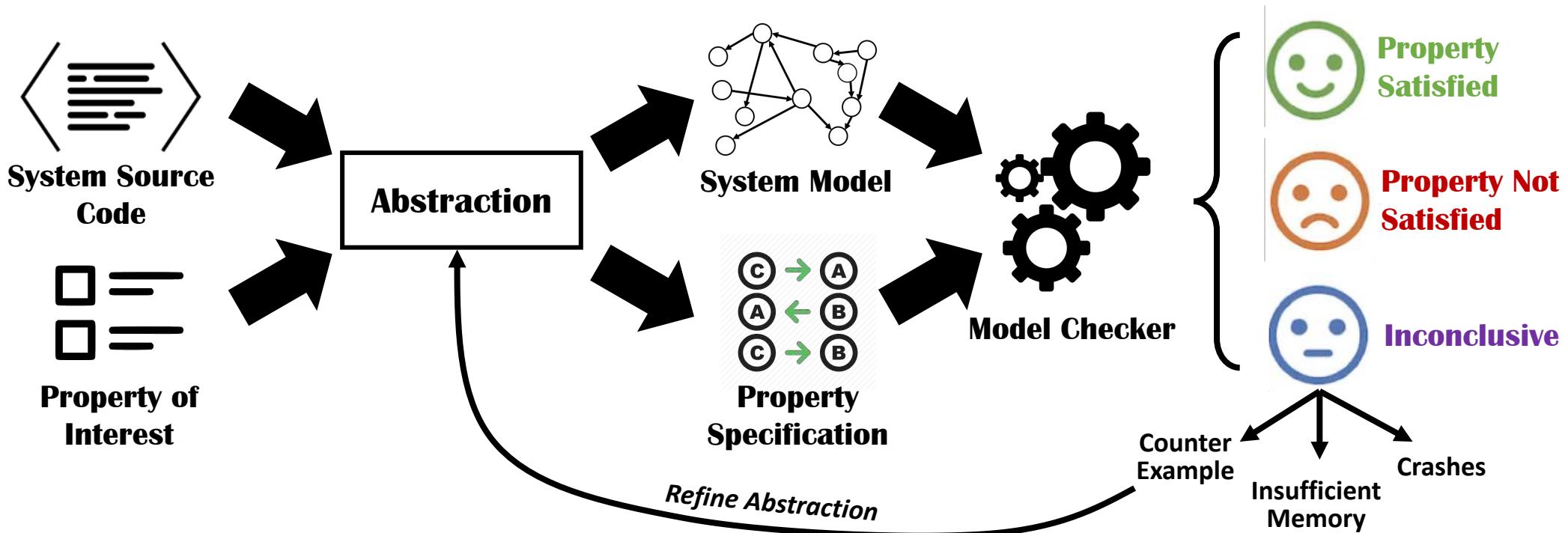
```
1 void foo() {  
2     int y = read(); ←  $y \in \mathbb{N}$   
3     if(y > 10 && y < -10) {  
4         return; ←  $y \in ]-\infty, -10[ \cup ]10, +\infty[$   
5     }  
6     int z = y * 2; ←  $y \in [-10, 9]$   
7     if(z == 12) {  
8         crash();  
9     } else {  
10        printf("OK");  
11    }  
12 }
```

Does this program crash?



Static Analysis: Model Checking

is an analysis technique where a given model of a system is exhaustively and automatically checked whether it meets a given specification. Both the model of the system and the specification are formulated in some precise mathematical language



For updated list of Model Checkers:

https://en.wikipedia.org/wiki/List_of_model_checking_tools



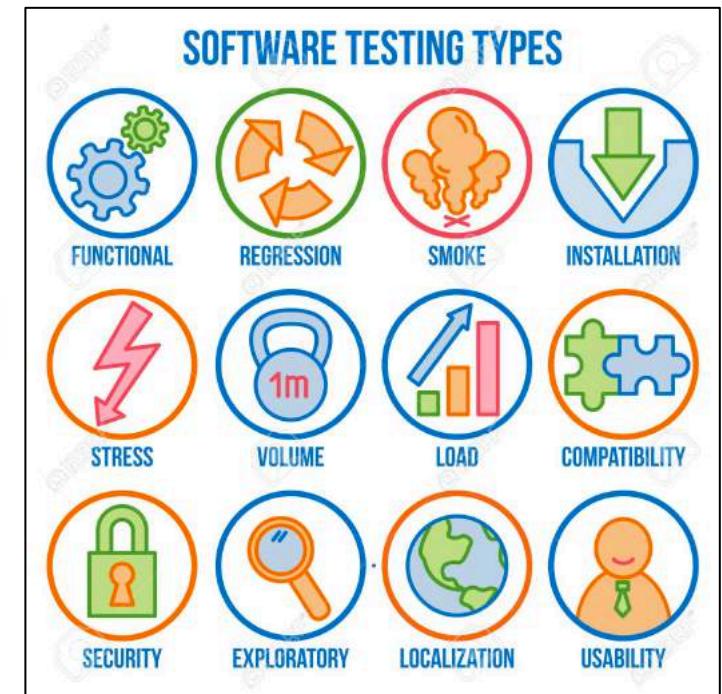
Dynamic Analysis: Testing

Software analysis performed by executing software against a pre-defined test cases.



Ultimate Goal is to achieve:

- Requirements Coverage
- Path Coverage

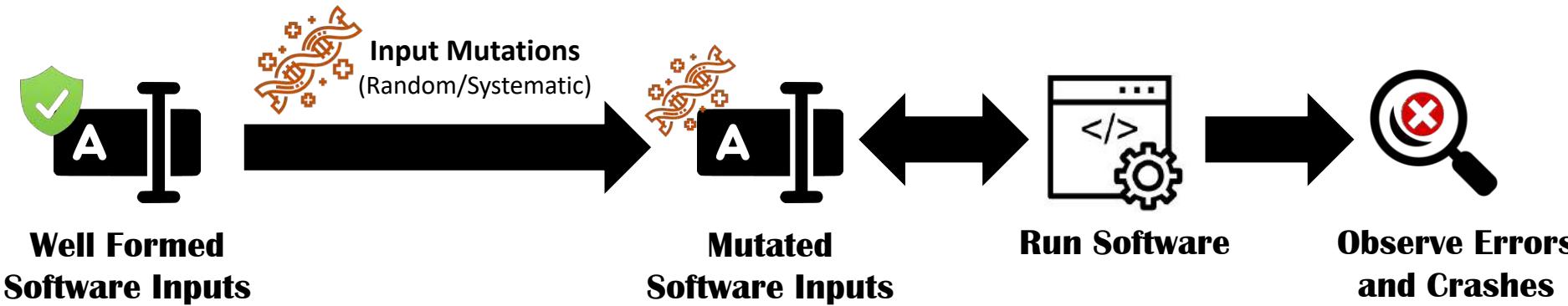


For more info:
<https://www.fuzzingbook.org/>



Dynamic Analysis: *Blind Fuzzing*

is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.



Ultimate Goal is to achieve:



```

american fuzzy lop 0.94b (unrtf)
----- overall results -----
Process timing
  run time: 8 days, 0 hrs, 0 min, 37 sec
  last new path: 8 days, 0 hrs, 0 min, 0 sec
  total paths: 268
  total crashes: 1
  last crash: none seen yet
  unit hangs: 0

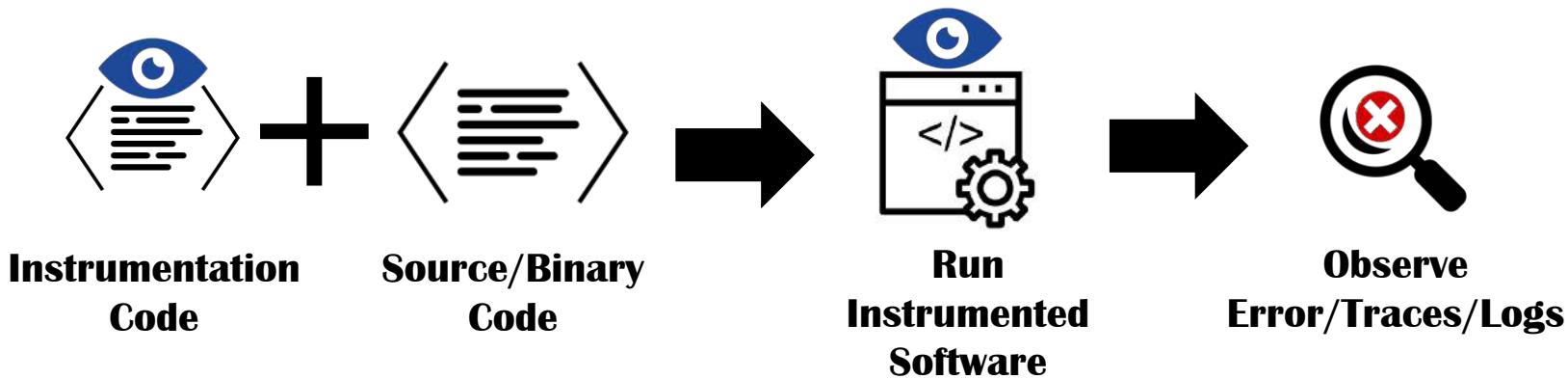
cycle progress
  new preexisting: 0 (0.80%)
  paths timed out: 0 (0.00%)
  stage progress
    new paths: 118/118 (100.00%)
    next stage: http://2/1
    stage execs: 7405/13.3K (55.57%)
    total execs: 24 2k
    exec speed: 645.5/sec
  fuzzing statistics
    total runs: 118/118 (100.00%)
    btrfs files: 0/0, 0/0, 0/0
    arithmetic: 0/0, 0/0, 0/0
    known ints: 0/0, 0/0, 0/0
    havoc: 0/0, 0/0
    trim: 4 B/828 (0.24% gain!)
```



Dynamic Analysis: Software Instrumentation

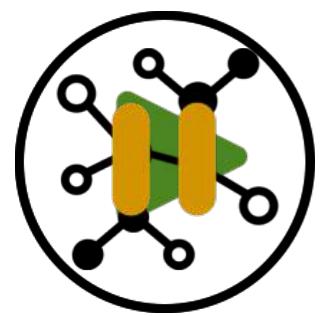
an ability to monitor software run to diagnose errors, and write trace information.

Programmers implement instrumentation in the form of code instructions that monitor specific components in a system



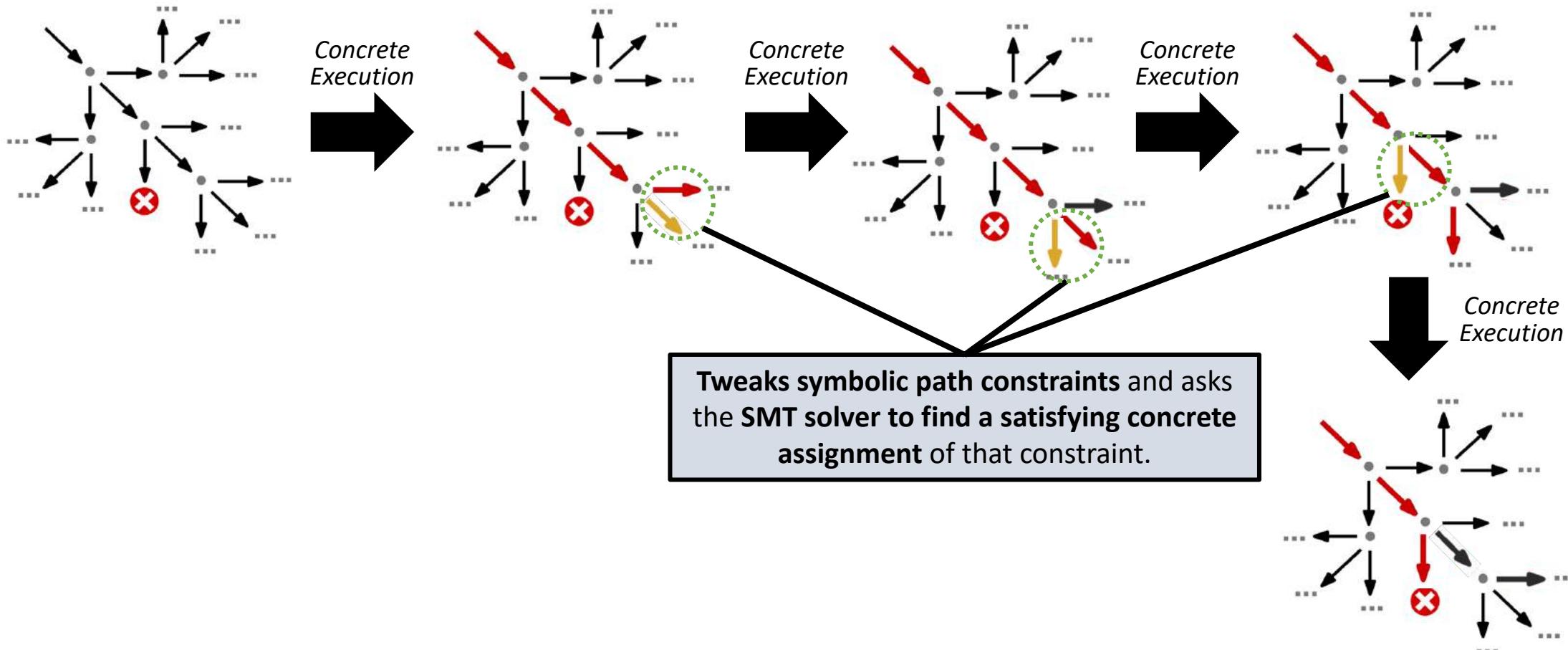
Ultimate Goal is to achieve:

- Requirements Coverage
- Path Coverage

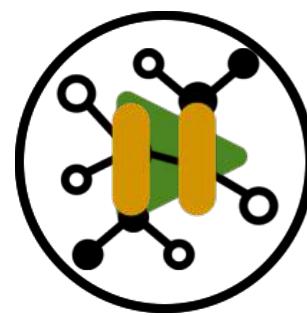


Hybrid Analysis: Concolic Execution

combines both **symbolic** execution and **concrete** execution. The basic idea is to have the concrete execution drive the symbolic execution.



For updated list of available tools to perform Concolic Execution:
https://en.wikipedia.org/wiki/Concolic_testing#Tools

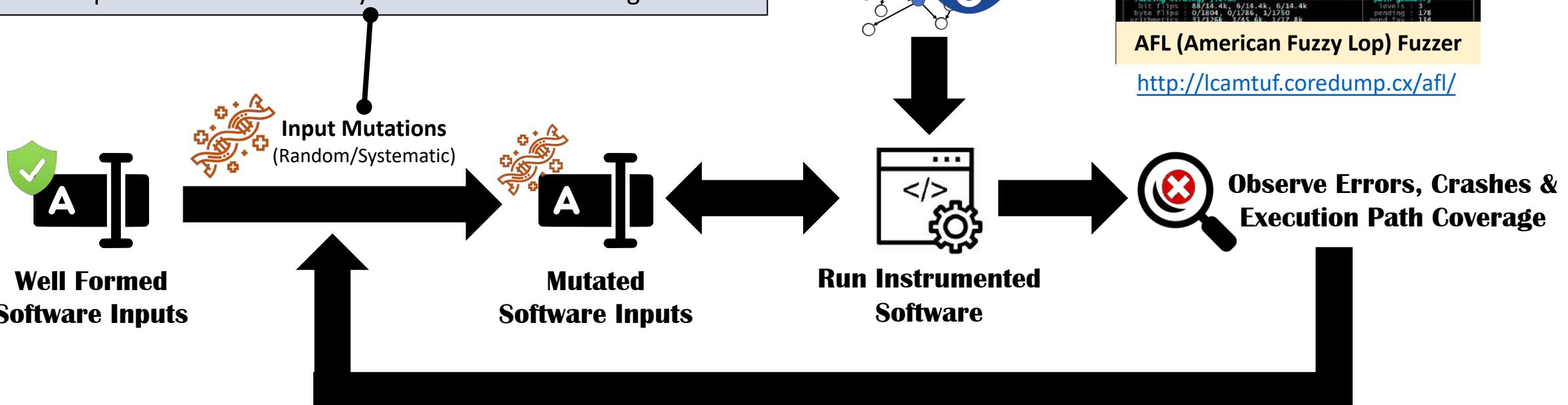


Hybrid Analysis: Smart (Guided) Fuzzing

is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

Note

Heuristics guide **genetic algorithm** to generate program inputs that push the fuzzer deeper into the program control flow, avoiding the common pitfalls of fuzzers to only test “shallow” code regions



If new program paths being explored then prioritize mutations of the tested input



For more info:
<https://www.fuzzingbook.org/>

Software Analysis

②

Pick a **software analysis strategy** or a combination of strategies to verify property conformance or vulnerability absence on each **feasible execution path**.

*What is a **feasible execution path**?*

What are software analysis strategies?

Software Analysis

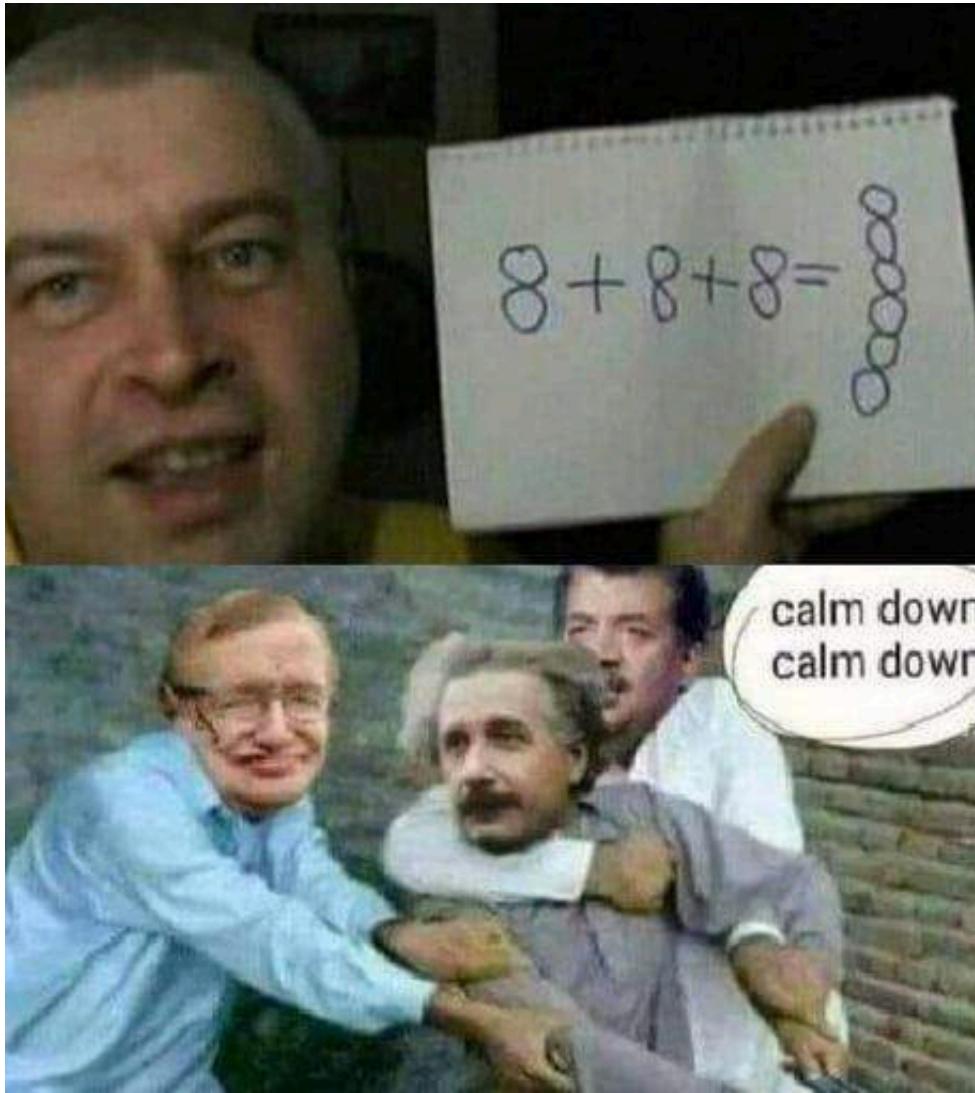
1

Choose the **security property** you want to proof its conformance or the **security vulnerability** you want to proof its absence.

2

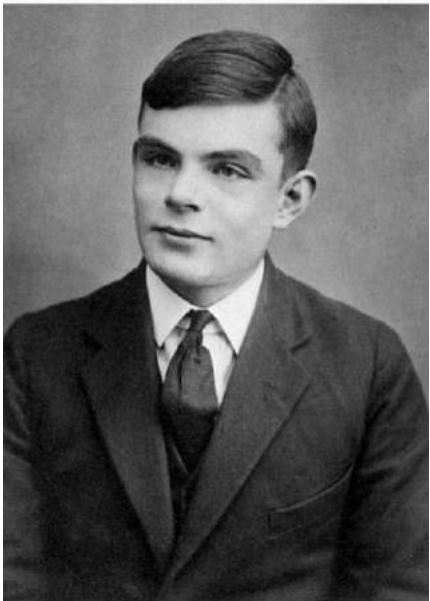
Pick a **software analysis strategy** or a combination of strategies to verify property conformance or vulnerability absence on each **feasible execution path**.

What is next?

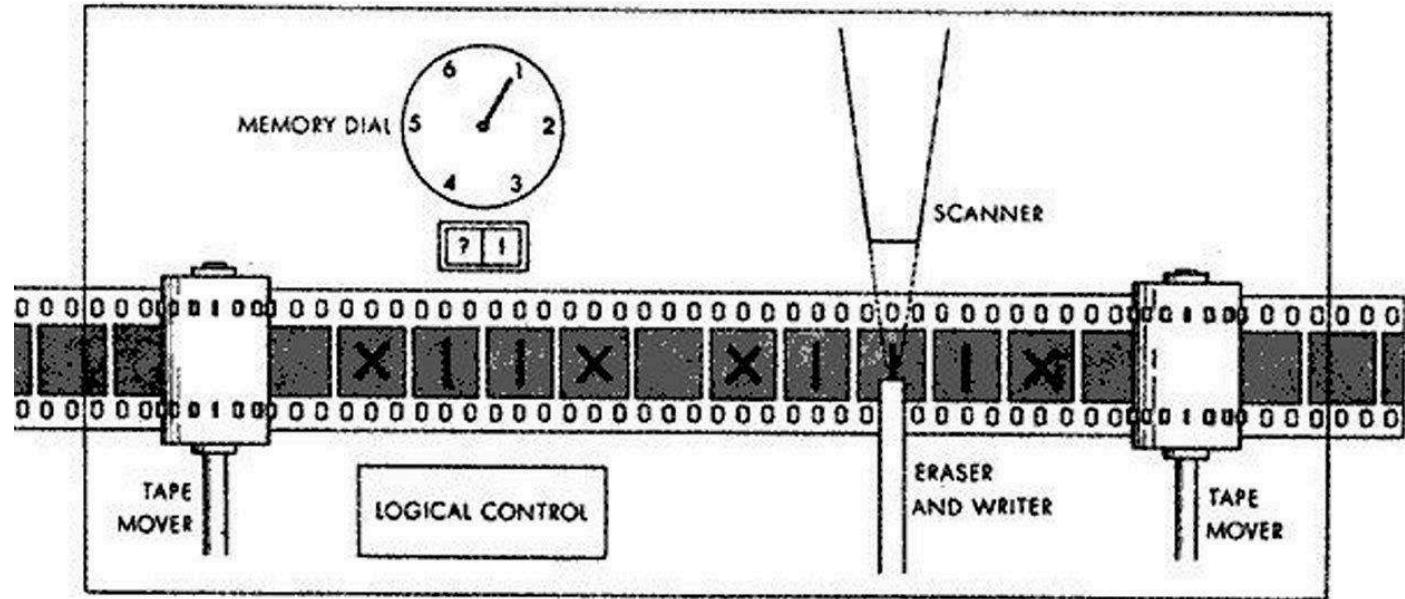


Halting Problem

is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever

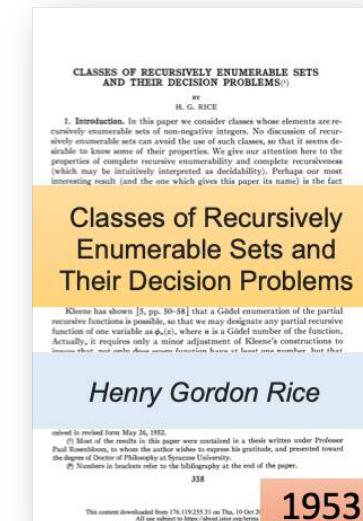
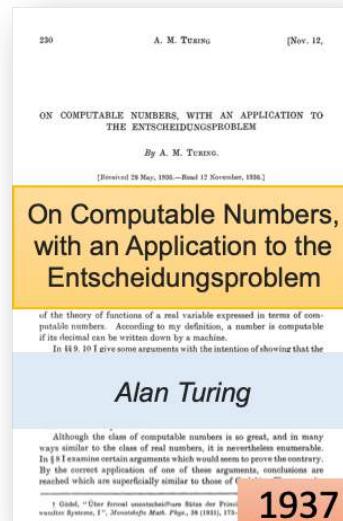
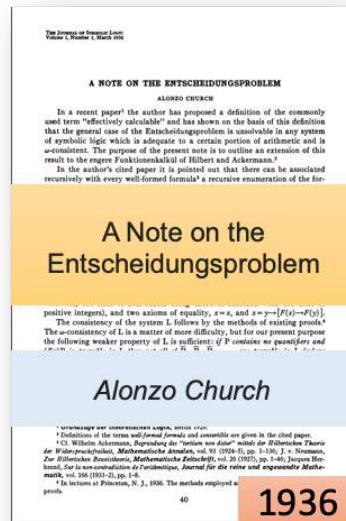


Alan Turing

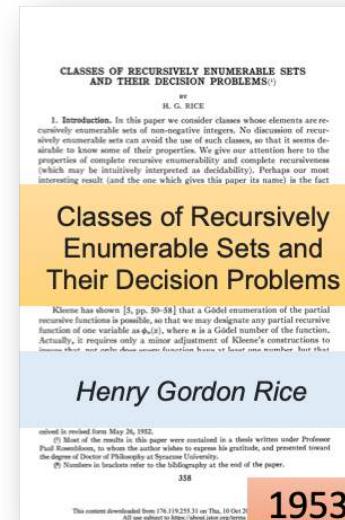
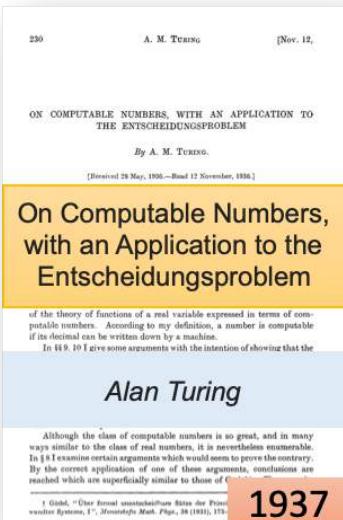
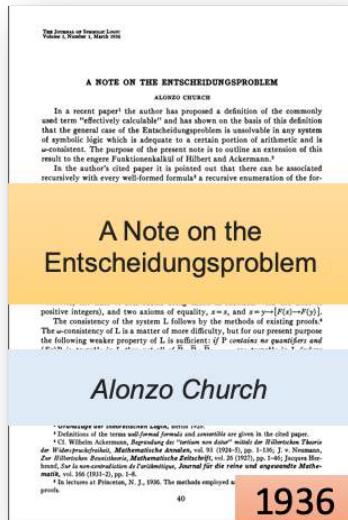


Turing Machine

Software analysis of arbitrarily complex software is known to be an **intractable problem!**



Fully automated software analysis encounters significant difficulties in practice – *it either does not complete or yields inaccurate results!*

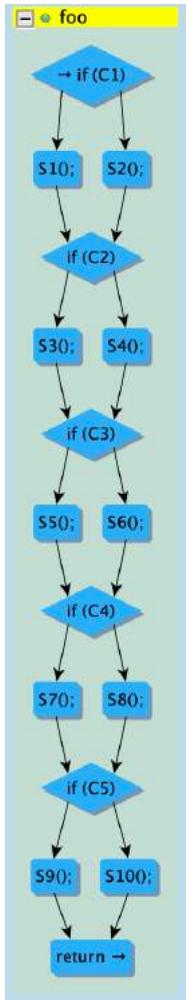




**Let's go over predominant
software analysis challenges!**

Path/State Explosion

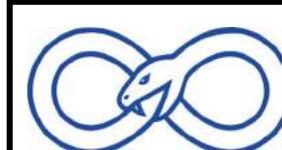
```
1 public void foo() {  
2     if(C1) {  
3         S1();  
4     } else {  
5         S2();  
6     }  
7     if(C2) {  
8         S3();  
9     } else {  
10        S4();  
11    }  
12    if(C3) {  
13        S5();  
14    } else {  
15        S6();  
16    }  
17    if(C4) {  
18        S7();  
19    } else {  
20        S8();  
21    }  
22    if(C5) {  
23        S9();  
24    } else {  
25        S10();  
26    }  
27 }
```



5 non-nested
Branch Points

→ 2⁵ = 32
Execution (Paths)
Behaviors

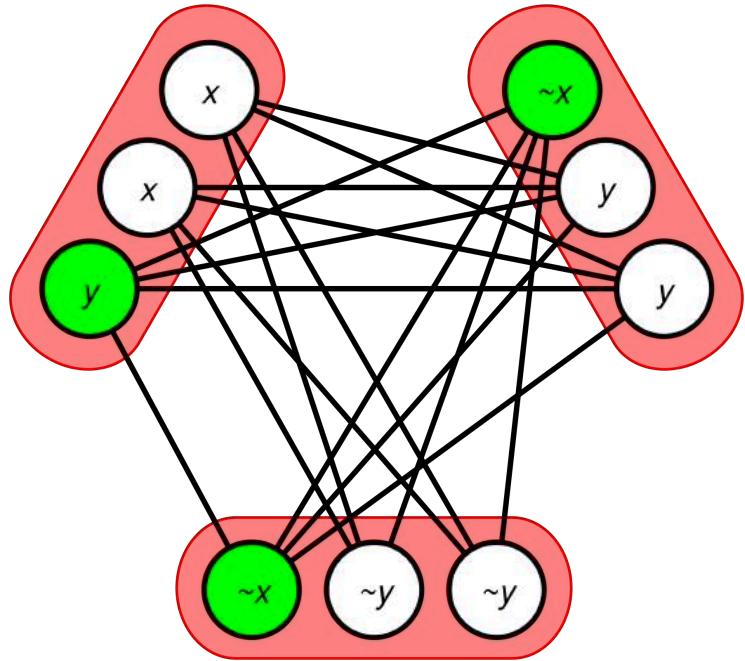
The number of paths
(*behaviors*) **increases**
exponentially with the
number of non-nested
branch points!



Loops and Recursions result
on **infinite execution trees!**

Hard to Reach Code Regions

**Path Analysis Woes when going
*Inter-procedural!***



Computational Intractability of Checking Feasible Behaviors

*The **Satisfiability problem** is known to be **NP-complete** problem!*



Difficult to Analyze Programming Constructs

Heap Modeling: Symbolic representation of data structures and pointers.

Environment Modeling: Dealing with native/system/library calls.

Obscure Flows: Event-driven frameworks, function pointers, polymorphism, reflections.

Variability-Aware Analysis



Operating Environment

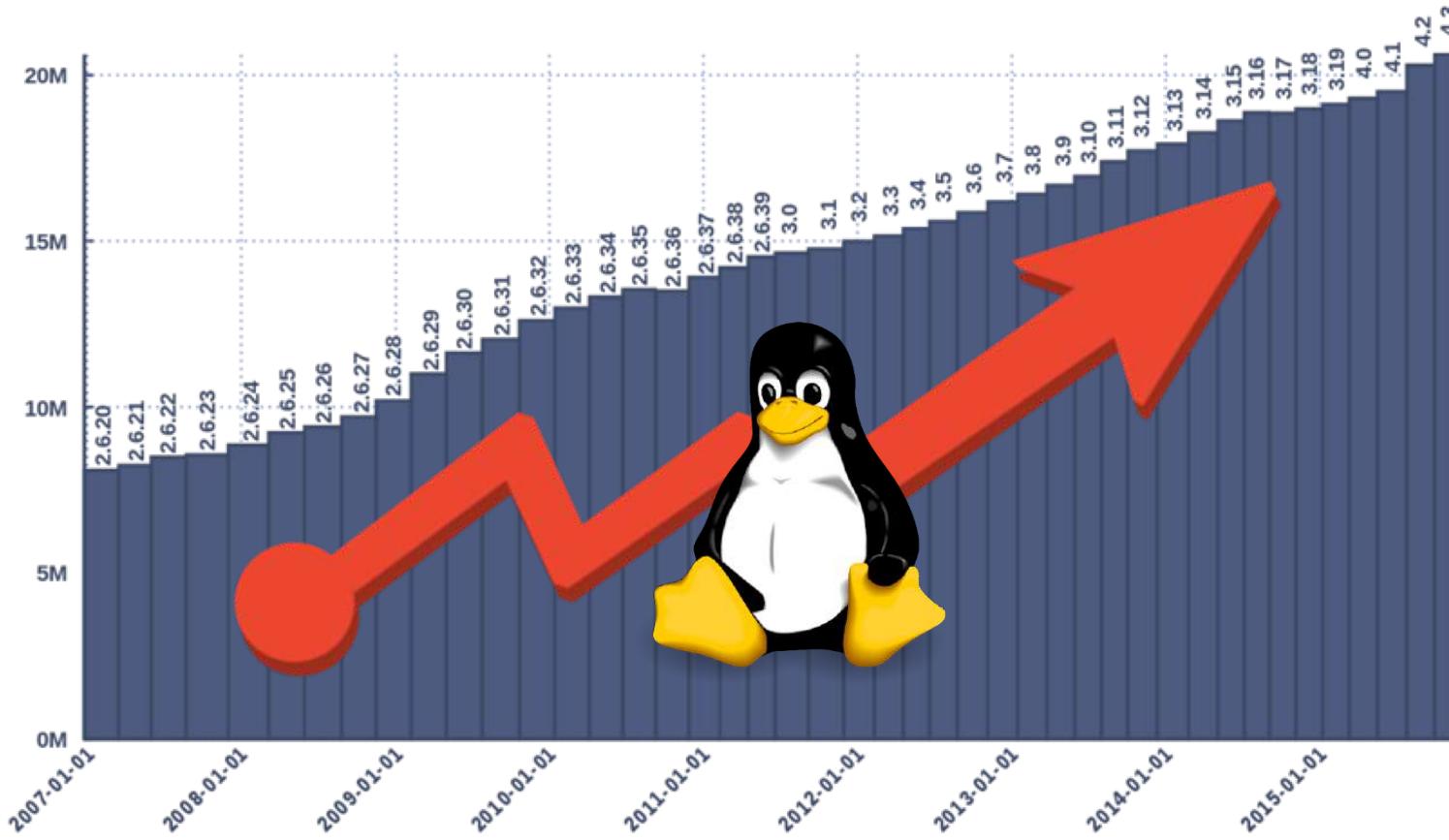
Heterogeneity

- **Environment Modeling:** Dealing with native/system/library calls.
- **Increasing Variability:** Linux Kernel has more than 10,000 configurations parameters.

Ever *Increasing* Complexity



Ever *Increasing* Size



20 MLOC ≈ 360K Pages



Evidence is *hard* to decipher; it does not simplify cross-checking



Traditional Approaches to Detect Vulnerabilities is like:

Search for similar needles in the haystack!



In Reality, Finding Wild and Sophisticated Vulnerabilities is like:

**Searching the Haystack for a needle
without knowing what the needle look like!**



Ambiguity: Malice or Legitimate?

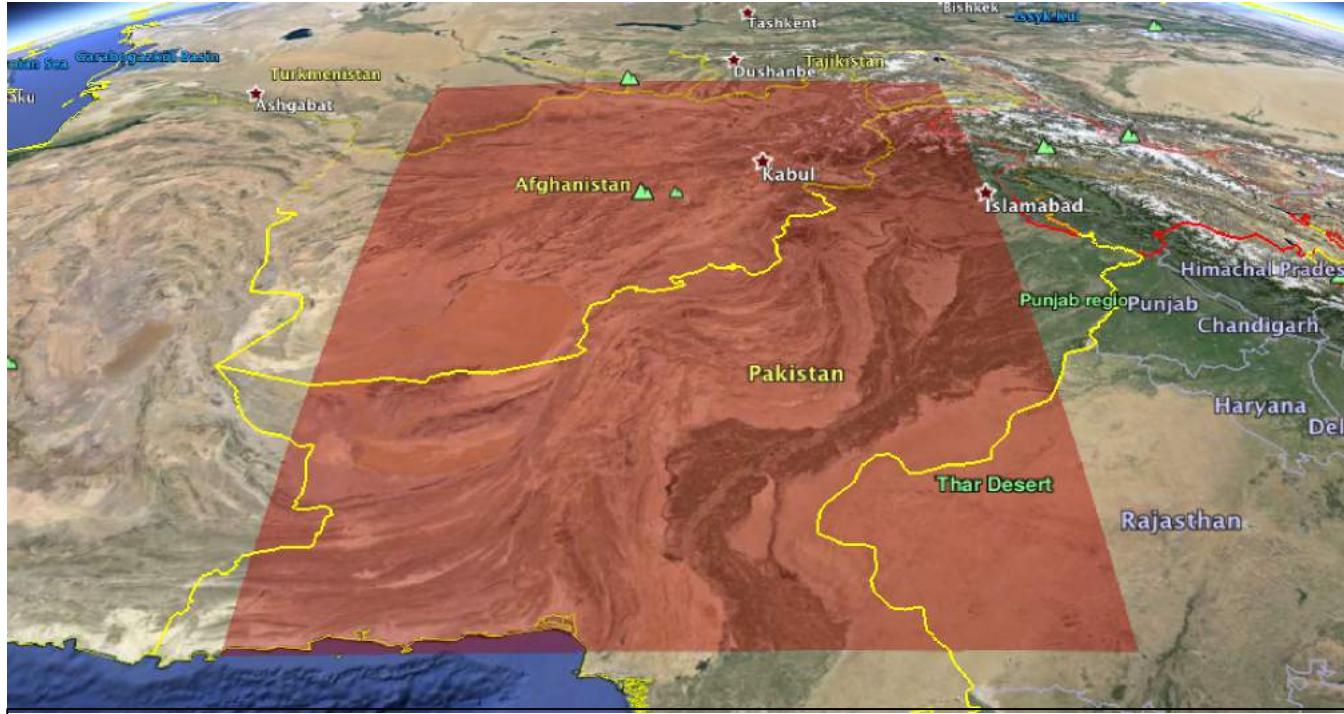
Behavior	App Purpose	Classification
<i>Send location to Internet</i>	Phone locator	Benign
<i>Send location to Internet</i>	Podcast player	Malicious
<i>Selectively block SMS messages</i>	Ad blocker	Benign
<i>Selectively block SMS messages</i>	Navigation	Malicious

There is a need for Domain-Specific Knowledge!



Data Gathering and Relaying App for Military

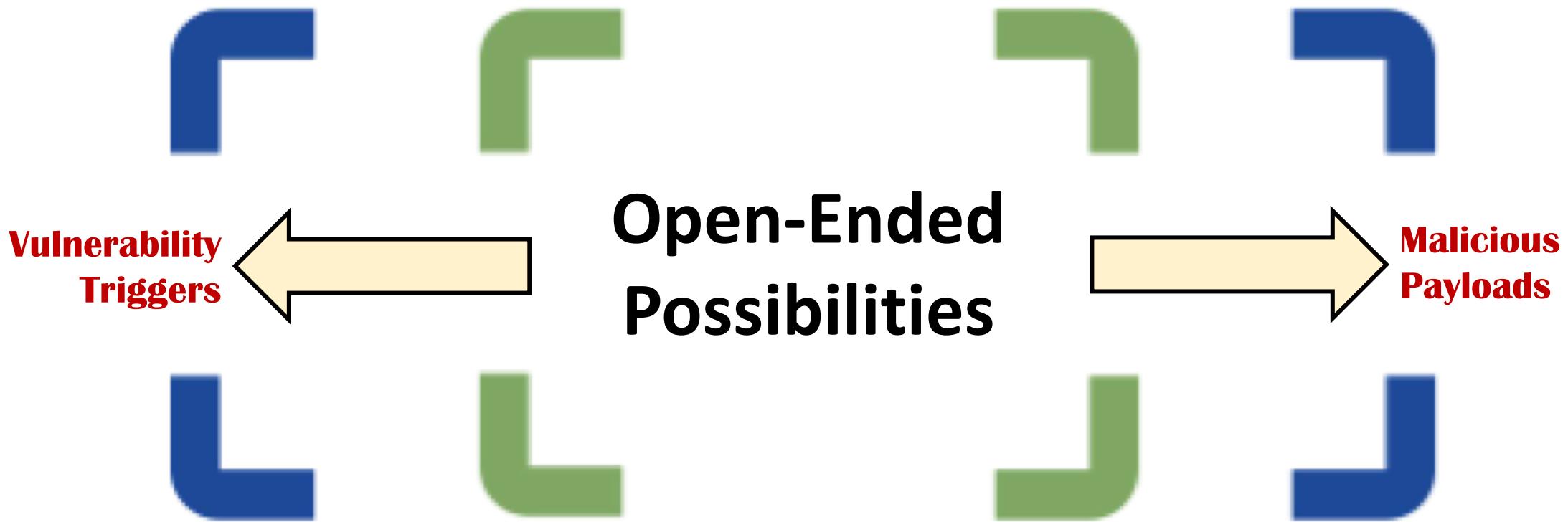
- 55K lines of code
- Strategic mission planning/review
- Audio and video recording
- Geo-tagged camera snapshots
- Real-time map updates based on GPS



```
@Override  
public void onLocationChanged(Location tmpLoc) {  
    location = tmpLoc;  
    double latitude = location.getLatitude();  
    double longitude = location.getLongitude();  
    if((longitude >= 62.45 && longitude <= 73.10) &&  
        (latitude >= 25.14 && latitude <= 37.88)) {  
        location.setLongitude(location.getLongitude() + 9.252);  
        location.setLatitude(location.getLatitude() + 5.173);  
    }  
}
```

Malware triggered by a geographic region!

What is different about detecting sophisticated vulnerabilities?

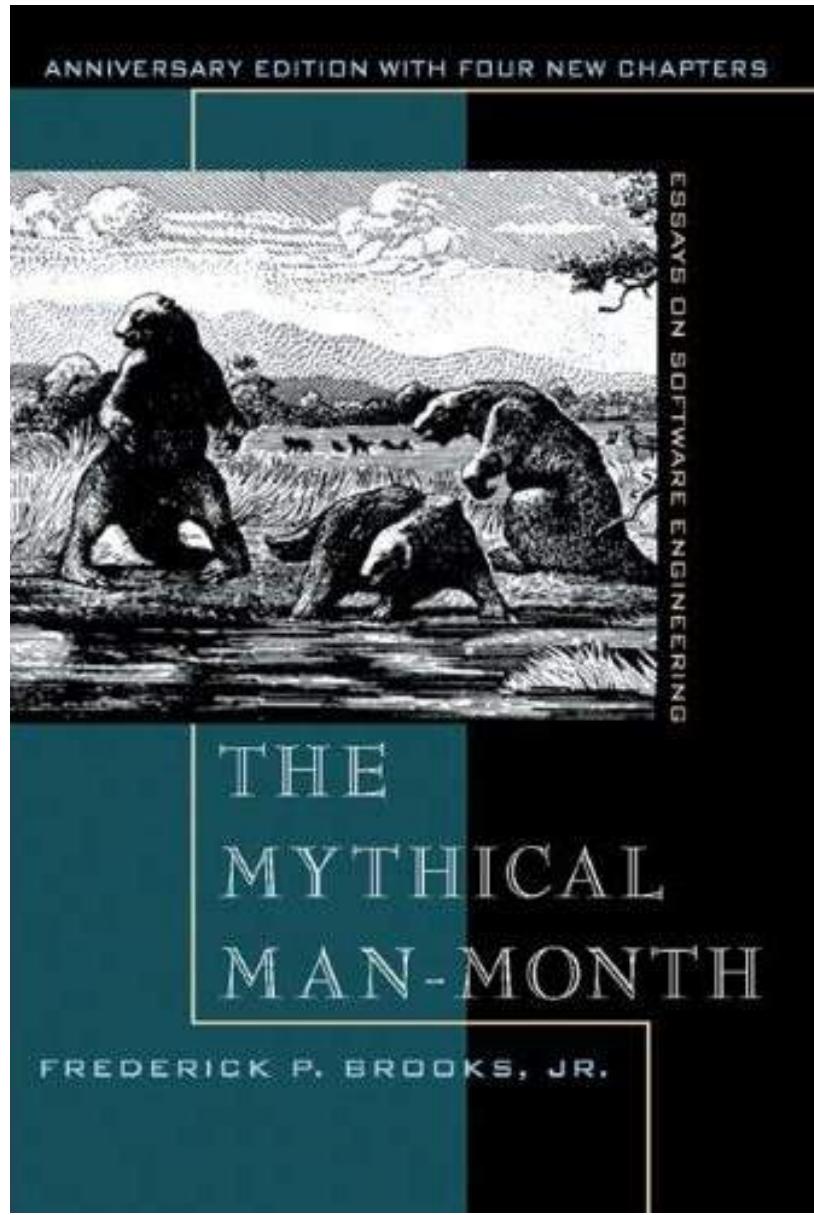


Developing plausible hypotheses for vulnerability trigger and malicious payload becomes a **critical part of malware discovery!**



Fully automated software analysis encounters
significant difficulties in practice – it either does
not complete or yields inaccurate results!





*"If indeed our objective is to build computer systems that solve very challenging problems, my thesis is that IA > AI, that is, that **intelligence amplifying systems** can, at any given level of available systems technology, beat AI systems. That is, a **machine and a mind can beat a mind-imitating machine working by itself.**"*





*The First
Turing Award
recipient*

Reports and Articles

Social Processes and Proofs of Theorems and Programs

Richard A. De Millo,
Georgia Institute of Technology

Richard J. Lipton and Alan J. Perlis
Yale University

Social Processes and Proofs of Theorems and Programs

specification CR Categories: 2.10, 4.6, 5.24

gramming more mathematics-like, is to increase dramatically one's confidence in the correct functioning of a piece of software, and the device that verifiers use to achieve this goal is a long chain of formal, deductive logic. In mathematics, the aim is to increase one's confidence in the correctness of a theorem, and it's nice that

Richard A. De Millo, Richard J. Lipton and Alan J. Perlis

Authors' addresses: R.A. De Millo, Georgia Institute of Technology, Atlanta, GA 30332; A.J. Perlis and R. Lipton, Dept. of Computer Science, Yale University, New Haven, CT 06520.
© 1979 ACM 0001-078X/79/0500-0017 \$00.75.

211

Communications of the ACM

May 1979 Volume 22 Number 5

1979

“Software verification, like “proofs” in mathematics, should provide evidence that humans can follow and thus be able to build trust into the correctness of the software verification.”

EXPERT OPINION
Editor: Daniel Zeng, University of Arizona and Chinese Academy of Sciences, sangtianz@gmail.com

Man versus Machine or Man + Machine?

Mary (Missy) Cummings, Duke University and MIT
 In developing any complex system that involves the integration of human decision making and But how do we know what's the right balance between humans and computers in these complex systems? And how can we make sure that the machine is doing what it's supposed to do?

Man versus machine or man + machine?

The problem of human-autonomy role allocation

Mary Cummings

Commercial aviation, military, robotics, aerospace, and maritime industries increasingly turn to automation to augment and sometimes replace human performance. The relative strengths and limitations of machine and human performance have been well documented (see Table 1), sometimes referred to as the "Human-Machine Performance Spectrum." This spectrum ranges from the machine being superior to the human in all respects to the human being superior to the machine in all respects.

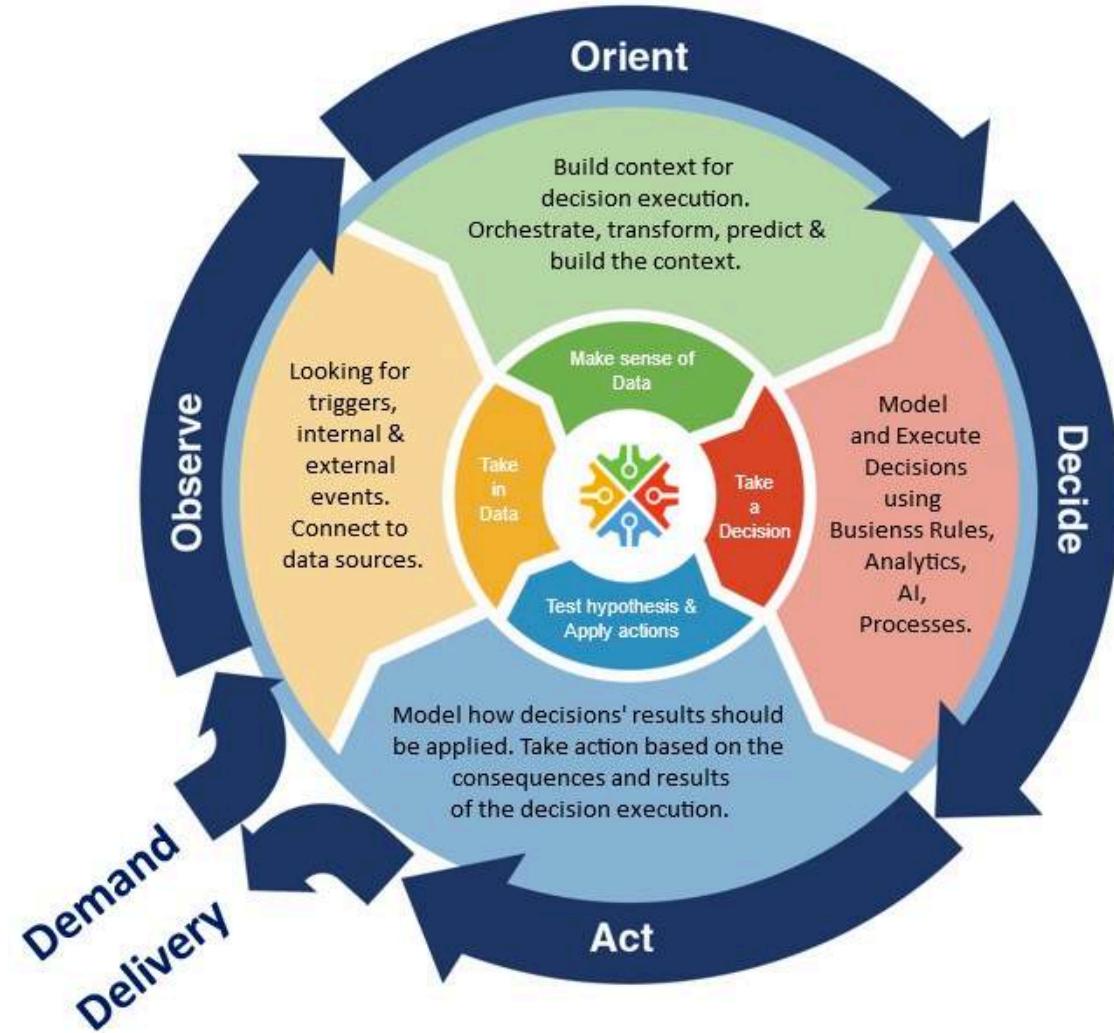
IEEE Intelligent Systems 2014

home robots will make the human-autonomy interaction issue and associated competing demands ubiquitous. The predominant engineering viewpoint across these systems is to automate as much as possible, and minimize the amount of human interaction. Indeed, many commercial systems see the human as a potential source of the errors that can and should be designed out. Others may begin to recognize that humans must play a role in such systems, either for regulatory requirements or low probability event intervention (such as problems in nuclear reactors).

with justification (possibly erroneous) for how to replace the human with automation. For traditional engineers with no training in human-autonomy interaction, this is exactly what they're trained to do—reduce disturbances and variability in a system and make it more predictable. Indeed, they're trying to "capitalize on the strengths of the machine and mitigate the weaknesses of the machine."² and this is an important piece of ethnographic information critical for understanding why traditional engineers and computer scientists are so attracted by such representations.

IEEE INTELLIGENT SYSTEMS © 2014 IEEE
 Published by the IEEE Computer Society

Attribute	Machine	Human
Speed	Superior	Comparatively slow
Power Output	Superior in level in consistency	Comparatively weak
Consistency	Ideal for consistent, repetitive action	Unreliable learning and fatigue are factors
Information capacity	Multichannel	Primarily single channel
Memory	Ideal for literal reproduction, access restricted, and formal	Better for principles and strategies, access is versatile and innovative
Reasoning computation	Deductive, tedious to program, fast and accurate, poor error correction	Inductive, easier to program, slow, accurate, and good error correction
Sensing	Good at quantitative assessment, poor at pattern recognition	Wide ranges, multifunction, judgment
Perceiving	Copes with variation poorly, susceptible to noise	Copes with variation better, susceptible to noise



USAF Colonel John Boyd developed the OODA framework as a way to explain the superior agility of US fighter pilots in aerial combat situations. The pilot must iterate the OODA loop faster than his opponent in order to **decide**, and **act** before his opponent has a chance to **observe**, **orient** himself to new information. Both pilots are aided by machines and a superior pilot may still lose the race if his instruments fail to him at any point in the cycle.

The paradigm of OODA loops applies equally well to the context of software analysis and there is no reason that a human cannot be included in the cycle!

$$e^{i\pi} + 1 = 0$$

$$e^{i\pi} + 1 = 0$$

Euler's Identity: *The Most Beautiful Equation!*

New technological advances are crucial
for using the Euler's method with
software of gigantic proportions!



Ideas are great arrows,
but there has to be a bow.

Bill Moyers

Human-In-The-Loop

approach to
Detecting
Sophisticated
Vulnerabilities

© 2024



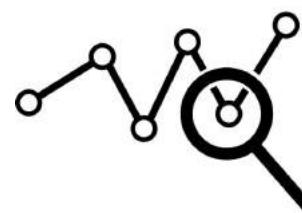
Threat Modeling

It is software-specific and requires human expertise

Developing plausible hypotheses for vulnerability trigger and malicious payload



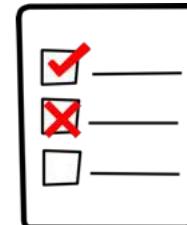
Software
Analysis
Problem



Software Analysis

A multi-stage process with Human-on-loop automated analysis

Analyzing software to gather evidence based on specified hypotheses



An exploit or
refutation of the
Threat Model

Our goal is to build an intelligence amplifying framework that mines and connects various software artifacts and enables human-machine interaction to solve complex software problems



The screenshot shows the Atlas software interface. At the top, there's a red header bar with the "atlas for Java and C" logo. Below it is a navigation bar with links: HOME, SIMDIFF, MODELIFY, ATLAS, SIMENGINE, SUPPORT, COMPANY, CAREERS, and NEWS. To the right of the navigation bar is a banner with the text "Understand code someone else wrote!" and the Ensoft logo. On the left, there's a sidebar titled "For Developers" with links: Atlas SDK and Shell, Big Data for Code, Free Atlas Lite, Try Atlas Professional, Complimentary Academic License (which is highlighted with a red box), Apply for a Community Atlas Standard License, Download Atlas, Buy Atlas, and System Requirements. The main content area has a section titled "Atlas" with a list of questions like "Q: When is this code ever used?", "Q: Where does this variable come from?", etc. It also includes a section titled "ATLAS QUICK FACTS" with bullet points: "Big Data for Code", "Understand code someone else wrote", "Leap through thousands of lines with a single click", "Free for academia and open source projects", "Try it on your own code today!", and "Latest version: Atlas 3.5.0". Below this is a "Large Codebase" visualization showing a grid of code snippets with various icons (like heart, magnifying glass) overlaid. At the bottom, there's a video player with a play button, the text "atlas 05:43", and a "Large Codebase" button.

Get a free license of Atlas now! In academia? Get a year long license here!

Atlas is an **intelligence amplifying framework** that **provides a new way to interactively explore software artifacts** and **enables analysts to write analysis scripts to tackle complex software problems!**



Atlas – A new way to explore software:
<https://www.youtube.com/watch?v=cZOWIJ-IO0k>



[www. http://www.ensoftcorp.com/atlas/](http://www.ensoftcorp.com/atlas/)



Source Code
C,C++,Java,Jimple

Abstract Syntax Tree



Artifacts Mapping

Declarations &
Dependencies

Control Flow
Relations

Data Flow
Relations

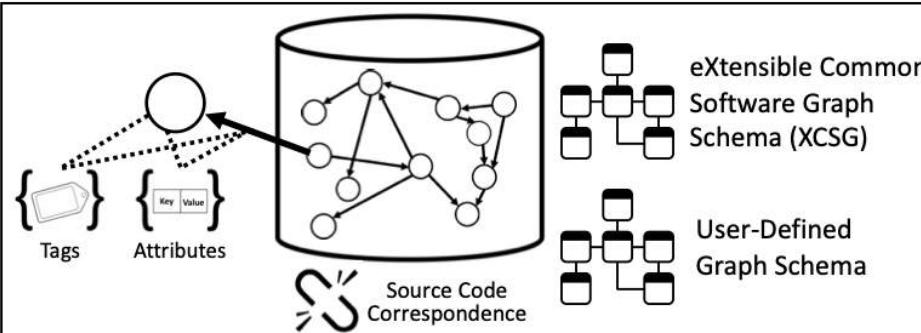
User-Defined
Relations



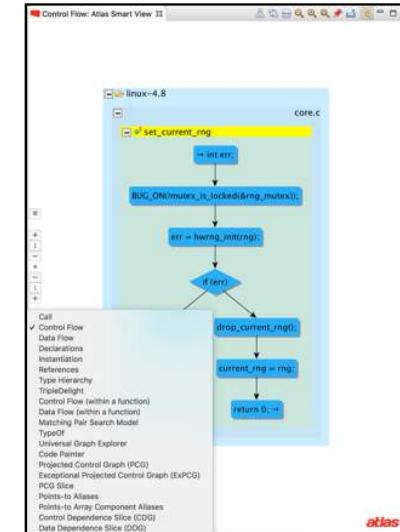
Multi-Tier Attributed Graph Database



**Atlas
Shell**



**Atlas
SmartView**



Query and Application Programming Interface



**Atlas
Toolboxes**



**User-Defined
Toolboxes**

**Android
Security
Toolbox**

Top Performer in DARPA APAC
A collection of program analysis tools specifically developed to make program analysis of Android apps simpler

**Loop
Analysis
Toolbox**

Top Performer in DARPA STAC
A suite of tools to reason about loops and to assist an analyst in detection of Algorithmic Complexity Vulnerabilities

**Verification
Toolbox**

Efficient, Scalable, and Practical Synchronization analysis of 12MLOC Linux Kernel

**PCG
Toolbox**

The PCG is a compact projection of the CFG to retain only the relevant execution behaviors and elide duplicate paths with identical execution behavior.

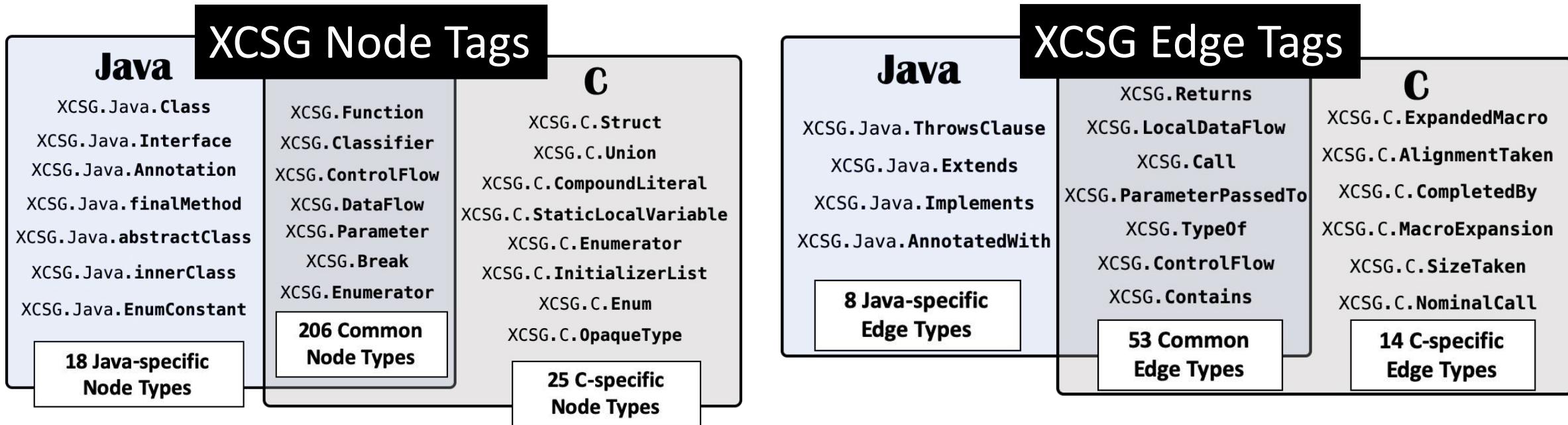
**DynaDoc
Toolbox**

Automated On-Demand Context Specific Documentation

Other Atlas Toolboxes are available at:
EnSoft Repository: <https://github.com/EnSoftCorp>
KCS LAB Repository: <https://github.com/kcs1>

eXtensible Common Software Graph (XCSG)

a harmonious representation of software written in different languages



XCSG defines a variety of **program artifacts (nodes)** and **relations (edges)** to capture the semantics of programming languages

Extensibility: New nodes and edges tags can be added to incorporate domain-specific knowledge!

Atlas Smart View and Atlas Element Detail View

The screenshot illustrates the integration of the Data Flow Smart View and the Element Detail View within the Atlas IDE.

Data Flow Smart View: This view shows the flow of data through the selected artifact, `dswrite.c`. A yellow box highlights the variable `buff` in the code, which is then tracked through its uses in the data flow graph. The graph shows nodes representing memory locations (`buff`, `df(local)`, `drptr->irbuff = buff;`) and assignments (`. =`). Arrows indicate the flow of data between these nodes.

Selected Artifact: The code editor on the left shows the `dswrite.c` file, with the `buff` variable highlighted in yellow, indicating it is the selected artifact.

Element Detail View: This view provides detailed information about the selected artifact and its components.

- Edge Attributes:** These are the attributes associated with the edges in the data flow graph, such as `NormalizedAddress` and `XCSG.Language.C`.
- Edge Tags:** These are the tags assigned to the edges, including `##index`, `XCSG.Language.C`, `XCSG.LocalDataFlow`, `XCSG.DataFlow (Edge)`, `XCSG.Edge`, and `XCSG.ModelElement`.
- Node Attributes:** These are the attributes associated with the nodes in the data flow graph, such as `NormalizedAddress` and `XCSG.id`.
- Node Tags:** These are the tags assigned to the nodes, including `##index`, `XCSG.Language.C`, `XCSG.Parameter`, `XCSG.CallInput`, `XCSG.ModelElement`, `XCSG.Node`, and `XCSG.Variable`.
- Node Attributes:** These are the attributes associated with the nodes in the data flow graph, such as `NormalizedAddress` and `XCSG.ModelElement.name`.
- Node Tags:** These are the tags assigned to the nodes, including `##index`, `XCSG.InstanceVariableAssignment`, `XCSG.Language.C`, `XCSG.Assignment`, `XCSG.DataFlow (Node)`, `XCSG.InstanceVariableAccess`, `XCSG.ModelElement`, and `XCSG.Node`.

Atlas Shell, Custom Scripts, and Atlas SDK

The screenshot shows the Atlas IDE interface with several windows:

- Source Code View:** Shows the content of `dfalloc.c` and `VerifierDriver.java`.
- Custom Script using Atlas SDK:** A green box containing the following Java code:

```
    * [1] corresponds to the list of CFG nodes for second event
    * [2] corresponds to the list of CFG nodes for re-definition
    * [3] corresponds to the list of CFG nodes for call-sites event
    */
    AtlasSet<Node> mallocNodes = getCFGNodeContainingCallsite("malloc");
    AtlasSet<Node> freeNodes = getCFGNodeContainingCallsite("free");
    AtlasSet<Node> barNodes = getCFGNodeContainingCallsite("bar");
    AtlasSet<Node> booNodes = getCFGNodeContainingCallsite("boo");

    AtlasMap<Node, List<AtlasSet<Node>>> functionEventsMap = new
    List<AtlasSet<Node>> nodes = new ArrayList<AtlasSet<Node>>();
    nodes.add(mallocNodes);

    // ...
    nodes.add(booNodes);
    functionEventsMap.put(CommonQueries.functions("bar").eval(), nodes);

    nodes = new ArrayList<AtlasSet<Node>>();
    nodes.add(new AtlasHashSet<Node>());
    nodes.add(freeNodes);
    nodes.add(new AtlasHashSet<Node>());
    nodes.add(new AtlasHashSet<Node>());
    functionEventsMap.put(CommonQueries.functions("foo").eval(), nodes);

    Q verificationCallGraphQ = CommonQueries.functions("foo", "bar");
    verificationCallGraphQ = verificationCallGraphQ.induceCommon();

```
- Graph 2:** A call graph for the `XINU` system, showing nodes like `dsopen.c`, `dfalloc.c`, and `getpid.c`.
- Graph 1:** A detailed control flow graph for the `dfalloc` function in `dfalloc.c`.

The screenshot shows the Atlas Shell interface with the following content:

- Common Queries:** A list of selected queries:

```
CommonQueries.cfg(selected)
res0: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>
show(res0)

CommonQueries.call(selected)
```
- Atlas Graph Queries:** A bracketed section indicating the use of the `CommonQueries` module.
- Atlas Shell:** A green box containing the text "Atlas Shell".



**KEEP
CALM
AND
START YOUR
ENGINES!**

①

Request your **Academic complimentary License** at:
<http://www.ensoftcorp.com/atlas/>

②

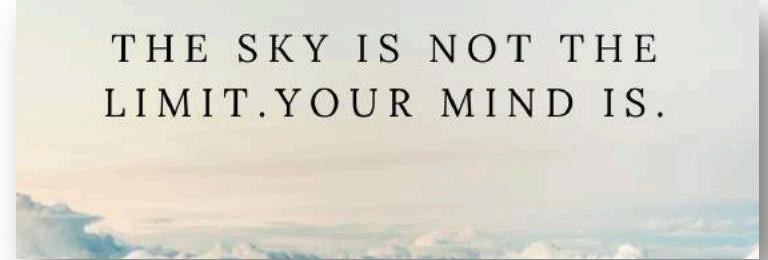
Read the **Atlas Installation Guide**

③

Go through our easy-to-follow tutorials:
http://ensoftatlas.com/wiki/Learning_Atlas_for_C
http://ensoftatlas.com/wiki/Learning_Atlas

④

*Unleash your
experience and build
your own beast!*



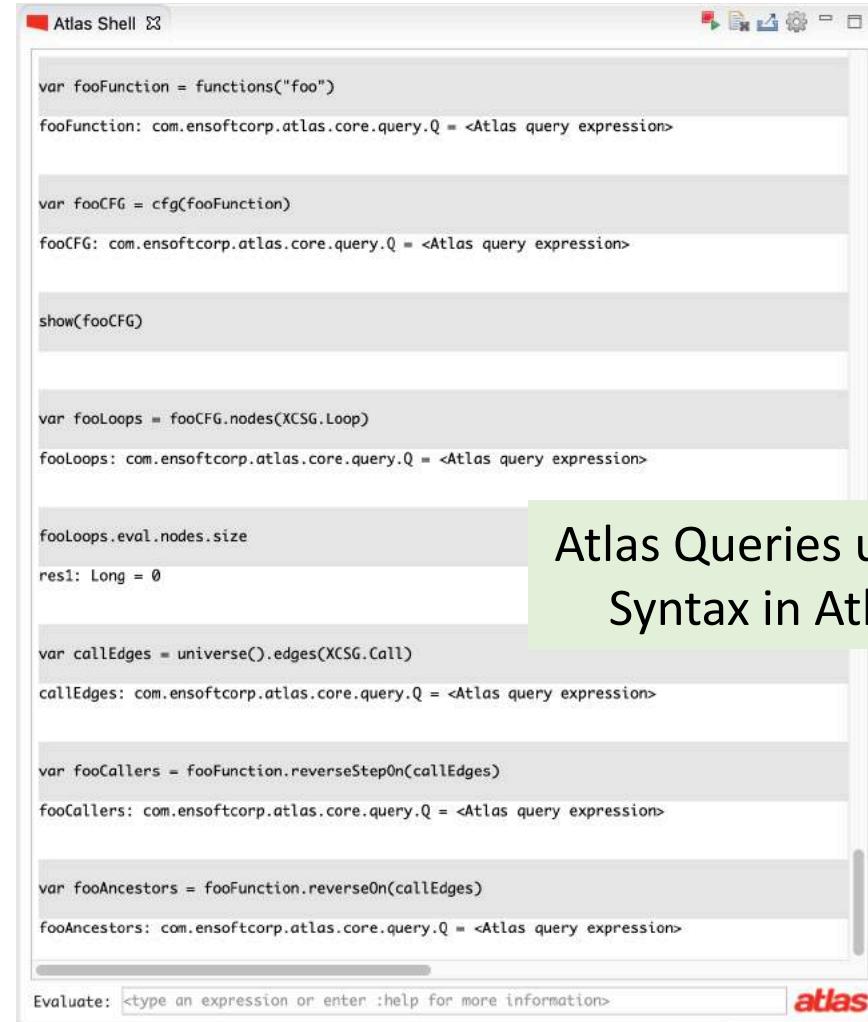
THE SKY IS NOT THE
LIMIT. YOUR MIND IS.

Atlas Query Language Examples

Atlas query language relies on graph calculus language to enable powerful computations with just a few lines of code

```
1 // We first find the function that we want to reason about.  
2 Q fooFunction = CommonQueries.functions("foo");  
3  
4 // Let us find all the loops in function "foo".  
5 Q fooCFG = CommonQueries.cfg(fooFunction);  
6  
7 // display the control flow graph of function "foo".  
8 DisplayUtil.displayGraph(fooCFG.eval(), null, "foo CFG");  
9  
10 // print out the number of loops  
11 Q fooLoops = fooCFG.nodes(XCSG.Loop);  
12 System.out.println(fooLoops.eval().nodes().size());  
13  
14 // Let us find all the functions that directly call "foo".  
15 Q callEdges = Query.universe().edges(XCSG.Call);  
16 Q fooCallers = fooFunction.reverseStepOn(callEdges);  
17  
18 // Let us find all the functions that have call chains to "foo".  
19 Q fooAncestors = fooFunction.reverseOn(callEdges);
```

Custom script written in Java using Atlas SDK



The screenshot shows the Atlas Shell interface with a window titled "Atlas Shell". Inside the window, several lines of Scala code are being executed:

```
var fooFunction = functions("foo")
fooFunction: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>

var fooCFG = cfg(fooFunction)
fooCFG: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>

show(fooCFG)

var fooLoops = fooCFG.nodes(XCSG.Loop)
fooLoops: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>

fooLoops.eval.nodes.size
res1: Long = 0

var callEdges = universe().edges(XCSG.Call)
callEdges: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>

var fooCallers = fooFunction.reverseStepOn(callEdges)
fooCallers: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>

var fooAncestors = fooFunction.reverseOn(callEdges)
fooAncestors: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>
```

At the bottom of the shell window, there is an "Evaluate" input field containing the placeholder text: "Evaluate: <type an expression or enter :help for more information>".

Atlas Queries using Scala
Syntax in Atlas Shell



Why a Graph Calculus Language?

*With the advent of powerful computers, many applications of graphs have evolved:
genetics, internet search engines, social networks, and many yet to come!*

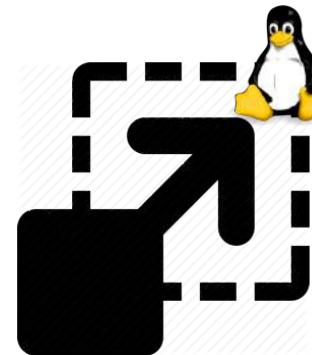
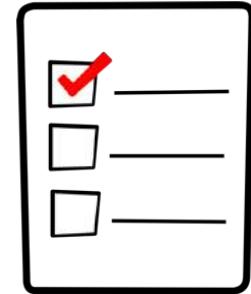
How are we using Atlas?



www. <https://kcs1.github.io/L-SAP>



Scalable, Efficient, and Practical
Linux Verification against
Synchronization Problems





Verification Results against Top Performing Tool

Kernel	Lock Type	Lock Instances	Explainable Verification (EV) tool based on Atlas platform			Linux Driver Verification (LDV) tool the top performers in the SV-COMP '14, 15' and '16					
			Safe	Unsafe	Incomplete Verification	Safe	Unsafe	Potential Bug	Incomplete Verification		
									Crash	Timeout	Total
3.17-rc1	Mutex	7887	7813	1	73	5494	0	91	1200	1102	2302
	Spin	14180	14097	1	82	8962	0	366	2188	2664	4852
3.18-rc1	Mutex	7893	7801	0	92	5427	0	98	2283	85	2368
	Spin	14265	14188	3	74	9152	0	383	4236	494	4730
3.19-rc1	Mutex	7991	7938	1	52	5527	0	103	2272	89	2361
	Spin	14393	14314	2	77	9204	0	358	4362	469	4831
Total		66609	66151	8	450	43766	0	1399	16541	4903	21444
Distribution		100.0%	99.3%	0.01%	0.7%	65.7%	0.0%	2.1%	24.8%	7.4%	32.2%

Importing Linux Kernel in Atlas

Linux Build	Atlas Nodes	Atlas Edges	Atlas Mapping Time
Small Build (defconfig)	7,493,303	23,264,592	15 min
Large Build (allmodconfig)	117,381,443	362,539,717	250 min

All Linux verification graphs are publicly available to cross-check verification results



<http://lsap.KnowledgeCentricSoftwareLab.com>

L-SAP

Search Text: Search

Kernel Version: 4.13

Type: Filter by Type

Status: Filter by Status

Driver: Filter by Driver

LOCK	LOCK	LOCK	LOCK
 spin 41016	 spin 41086	 spin 41178	 spin 118673
IRQ_TASK_LOCK	IRQ_TASK_LOCK	IRQ_TASK_LOCK	EVENT_LOCK
 spin 155353	 spin 155443	 spin 155475	 spin 168559
LOCK	LOCK	LOCK	LOCK
 spin 169939	 spin 171175	 spin 171193	 spin 171238

L-SAP

Matching Pair Graph

drivers/tty/serial/serial_core.c
Locking type: spin
Instance ID: 301251
Length: 28
Title: lock
Offset: 25823
Status: PAIRED

Control Flow Graphs (1)

Projected Control Graphs (1)

About



<https://ensoftcorp.github.io/pcg-toolbox/>



Projected Control Graph (PCG)

is a compact projection of the Control Flow Graph (CFG) that retain only the relevant execution behaviors and elide duplicate paths with equivalent execution behavior

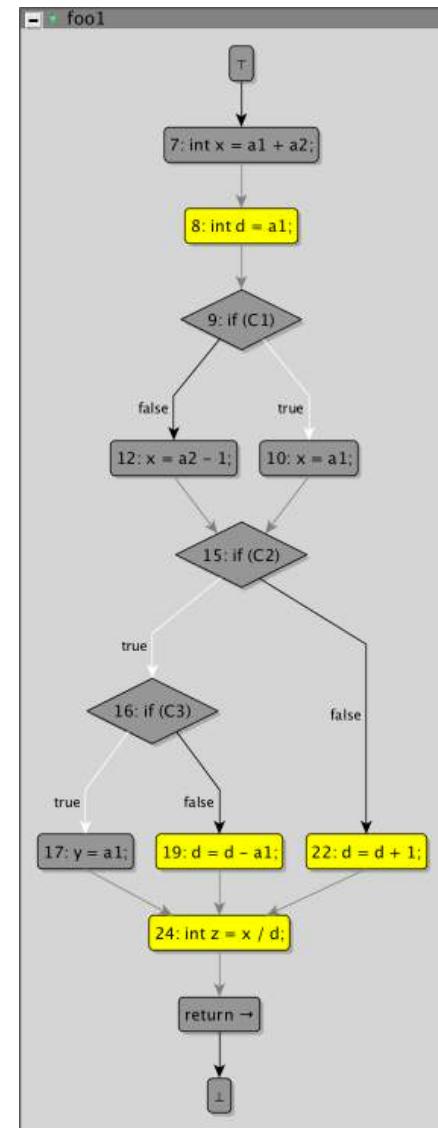
Algorithmic Challenge: Compute the distinct relevant behaviors without going through each path!

For any given analysis problem, the number of **distinct relevant execution behaviors** may be much smaller than the number of CFG paths!

Division-By-Zero (DBZ) Vulnerability?

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

Function foo1



Control Flow Graph (CFG)

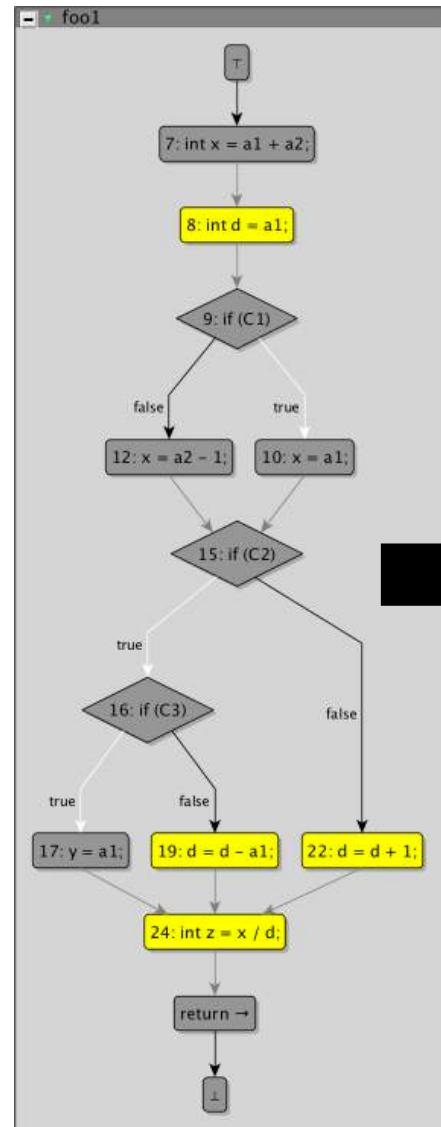
Six Possible Execution Paths

✗	$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c}_3], 19, 24$
✗	$B_2 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[\bar{c}_3], 19, 24$
✗	$B_3 : 7, 8, 9[c_1], 10, 15[\bar{c}_2], 22, 24$
✗	$B_4 : 7, 8, 9[\bar{c}_1], 12, 15[\bar{c}_2], 22, 24$
✗	$B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$
✗	$B_6 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[c_3], 17, 24$

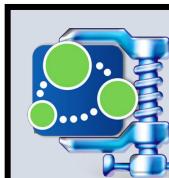
Division-By-Zero (DBZ) Vulnerability?

```
1 int a1 = 1, a2 = 2;
2 int y = 2;
3 bool C1 = true;
4 bool C2 = false;
5 bool C3 = true;
6 void foo1() {
7     int x = a1 + a2;
8     int d = a1;
9     if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18         }else{
19             d = d - a1;
20         }
21     }else{
22         d = d + 1;
23     }
24     int z = x / d;
25 }
```

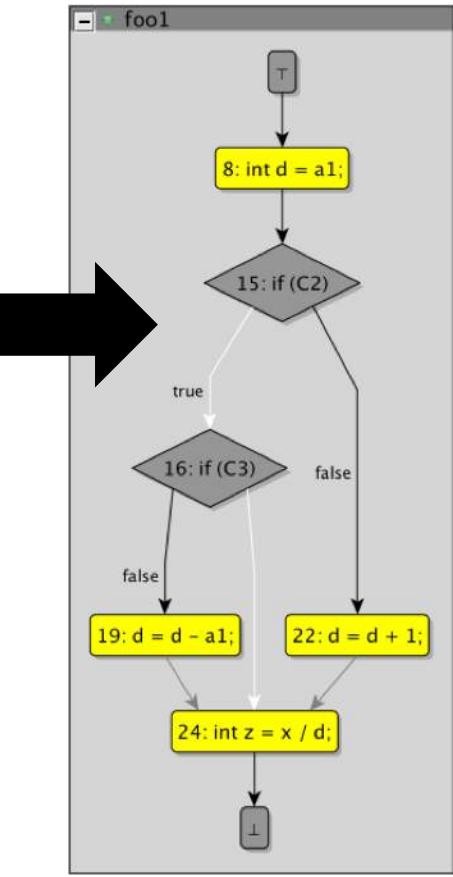
Function foo1



Control Flow Graph

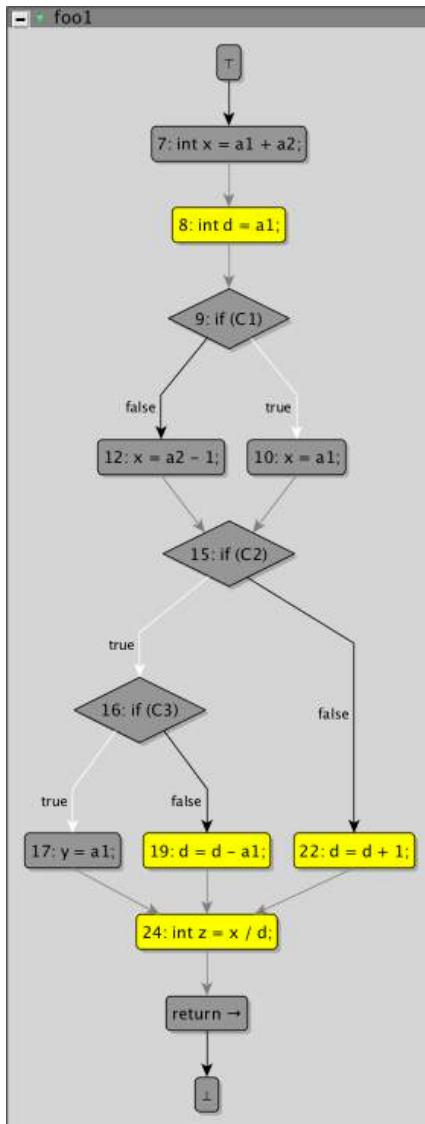


Efficient Graph
Transformations based on
famous Tarjan's Algorithm



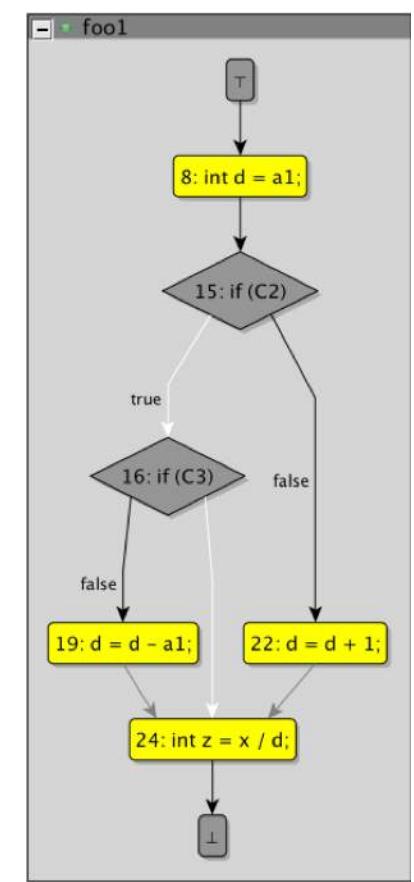
Projected Control Graph

Division-By-Zero (DBZ) Vulnerability?



Six Possible Behaviors	3 Relevant Behaviors
$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c}_3], 19, 24$	$RB_1 : 8, 15[c_2], 16[\bar{c}_3], 19, 24$
$B_2 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[\bar{c}_3], 19, 24$	
$B_3 : 7, 8, 9[c_1], 10, 15[\bar{c}_2], 22, 24$	$RB_2 : 8, 15[\bar{c}_2], 22, 24$
$B_4 : 7, 8, 9[\bar{c}_1], 12, 15[\bar{c}_2], 22, 24$	
$B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$	$RB_3 : 8, 15[c_2], 16[c_3], 24$
$B_6 : 7, 8, 9[\bar{c}_1], 12, 15[c_2], 16[c_3], 17, 24$	

Control Flow Graph



Projected Control Graph

Linux Kernel Case Study

with respect to lock/unlock operations as relevant events of interest

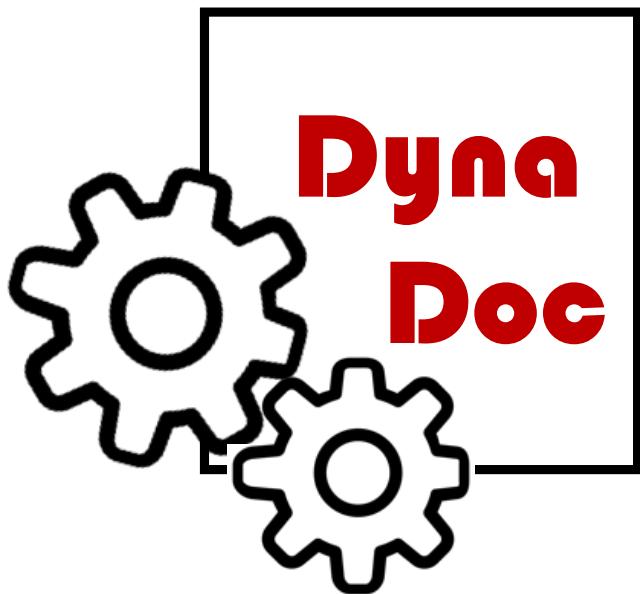
Function Name	Nodes		Edges		Branch Nodes		Paths	
	CFG	PCG	CFG	PCG	CFG	PCG	CFG	PCG
ptlrv_connect_interpret	791	8	1000	9	214	2	380414	3
kiblnd_passive_connect	668	24	840	40	174	17	34216	18
client_common_fill_super	644	17	801	29	162	13	1724067	14
qib_make_ud_req	630	9	833	13	160	5	20586	6
xfrm6_input_addr	574	8	769	11	151	4	1719	7
kiblnd_create_conn	568	16	714	27	149	12	3748	12
jbd2_journal_commit_transaction	522	4	648	3	127	0	2697	1
ceph_writenodes_start	416	13	540	21	126	9	1004	7
arcnet_interrupt	408	6	588	6	183	1	4004200	2
macsec_post_decrypt	390	8	521	9	104	2	1381	3

Control Flow Graph (CFG) captures the entire semantics!

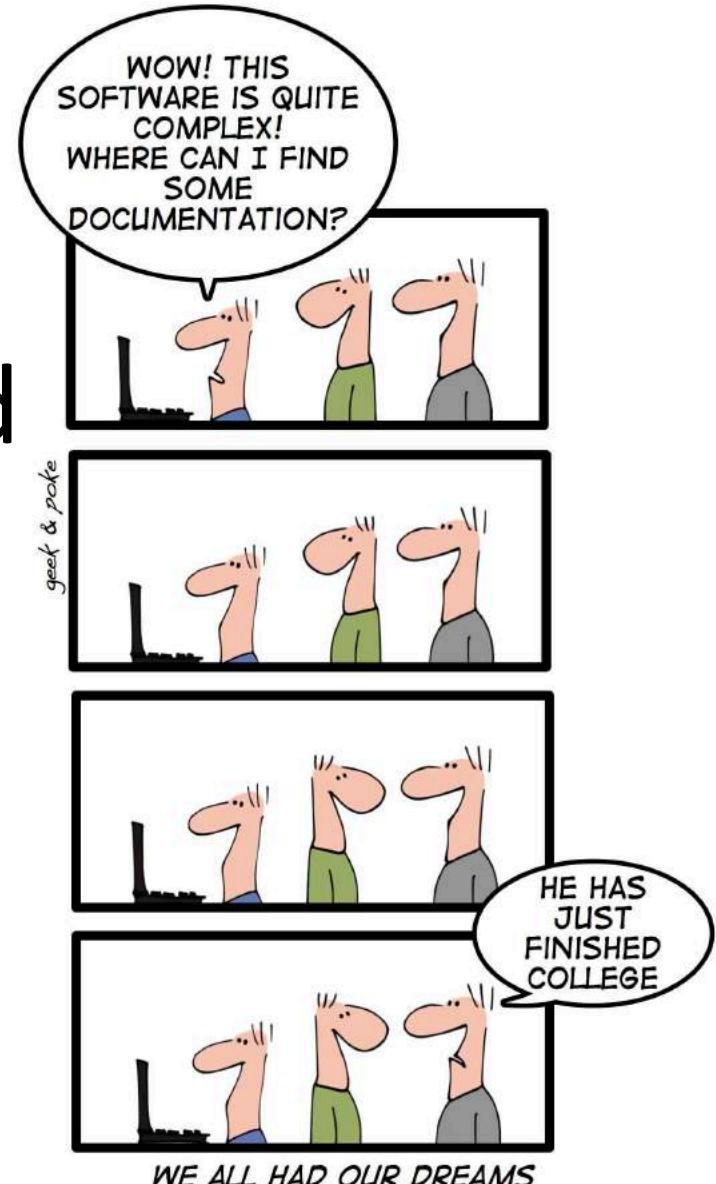
Exponentially many paths but **only a small number of relevant behaviors!**

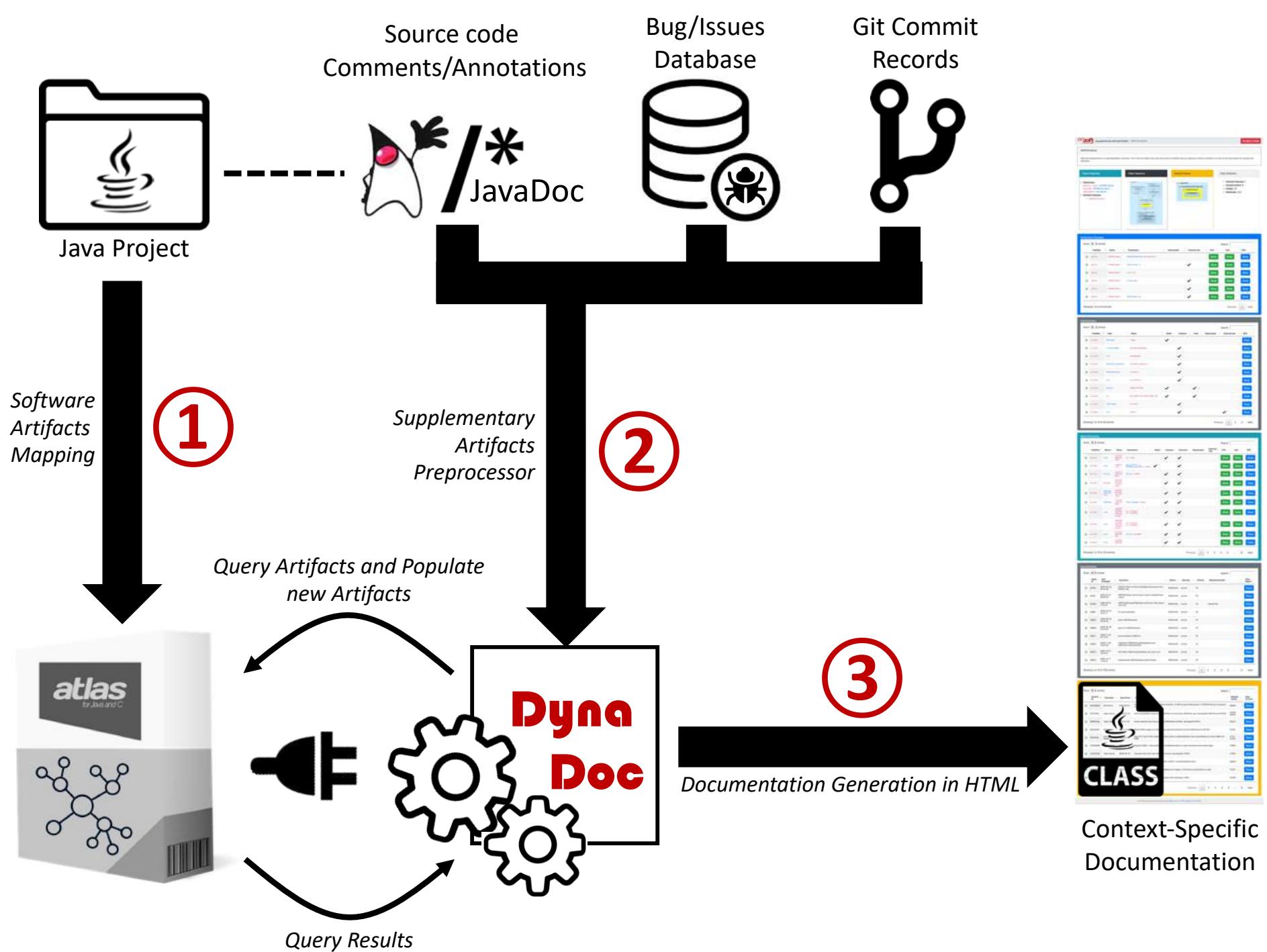


<https://github.com/EnSoftCorp/DynaDoc>



Automated On-Demand Context-Specific Documentation for Java Source Code





Participation in DARPA Programs

DARPA is investing billions of dollars into Securing Software



APAC

Automated Program
Analysis for Cybersecurity

STAC

Space/Time Analysis for
Cybersecurity

CASE

Cyber Assured Systems
Engineering

Blue Team on APAC and STAC programs and
as the **White Team** on CASE program

*We have competed with about a dozen Blue Teams on more than 200
malware challenges*

DARPA APAC Program

Automated Program Analysis for Cybersecurity

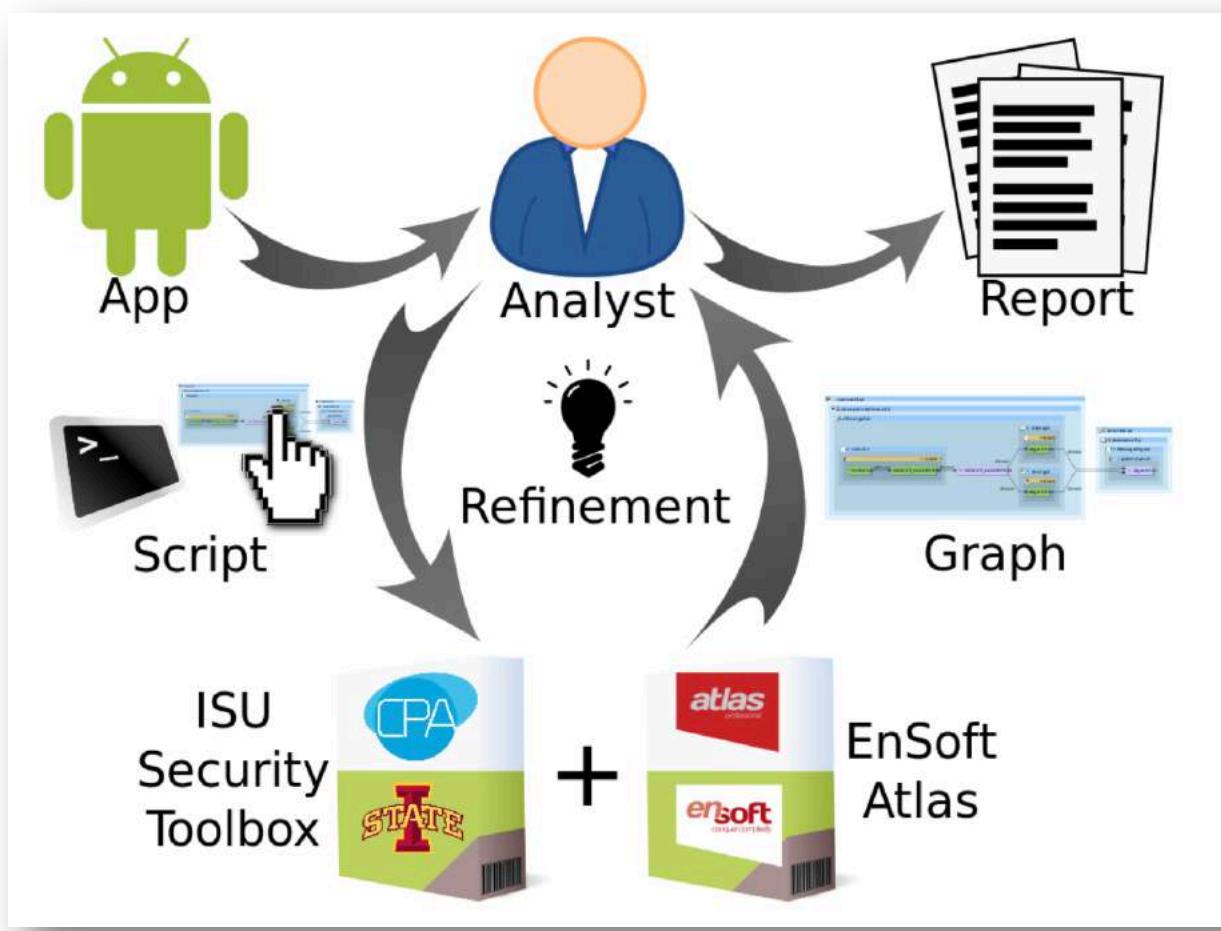
The program aims to address the challenge of **timely and robust security validation of mobile apps** by first defining security properties to be measured against and then developing automated tools to perform the measuring. The second challenge APAC aims to address is **producing practical, automated tools to demonstrate the cybersecurity properties identified.** *Successful tools would minimize false alarms, missed detections and the need for human filtering of results to prove properties.*



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

DARPA APAC Program

Automated Program Analysis for Cybersecurity



Android Security Toolbox

<https://ensoftcorp.github.io/android-essentials-toolbox/>

A screenshot of the EnSoft Atlas IDE interface. On the left, the **Package Explorer** shows a file named `TerminalManager.java` with code related to vibration. On the right, there is a dependency graph titled `vibrate call to vibrate` showing relationships between classes like `ConnectBotBad`, `TerminalManager`, and `Vibrator`. A status bar at the bottom indicates `249M of 784M` and `113M of 135M`.



Android Toolbox Demo

<https://www.youtube.com/watch?v=WhcoAX3HiNU>



Time-lapse Audit of DARPA APAC Challenge App:

<https://www.youtube.com/watch?v=p2mhfOMmgKI>

DARPA STAC Program

Space/Time Analysis for Cybersecurity

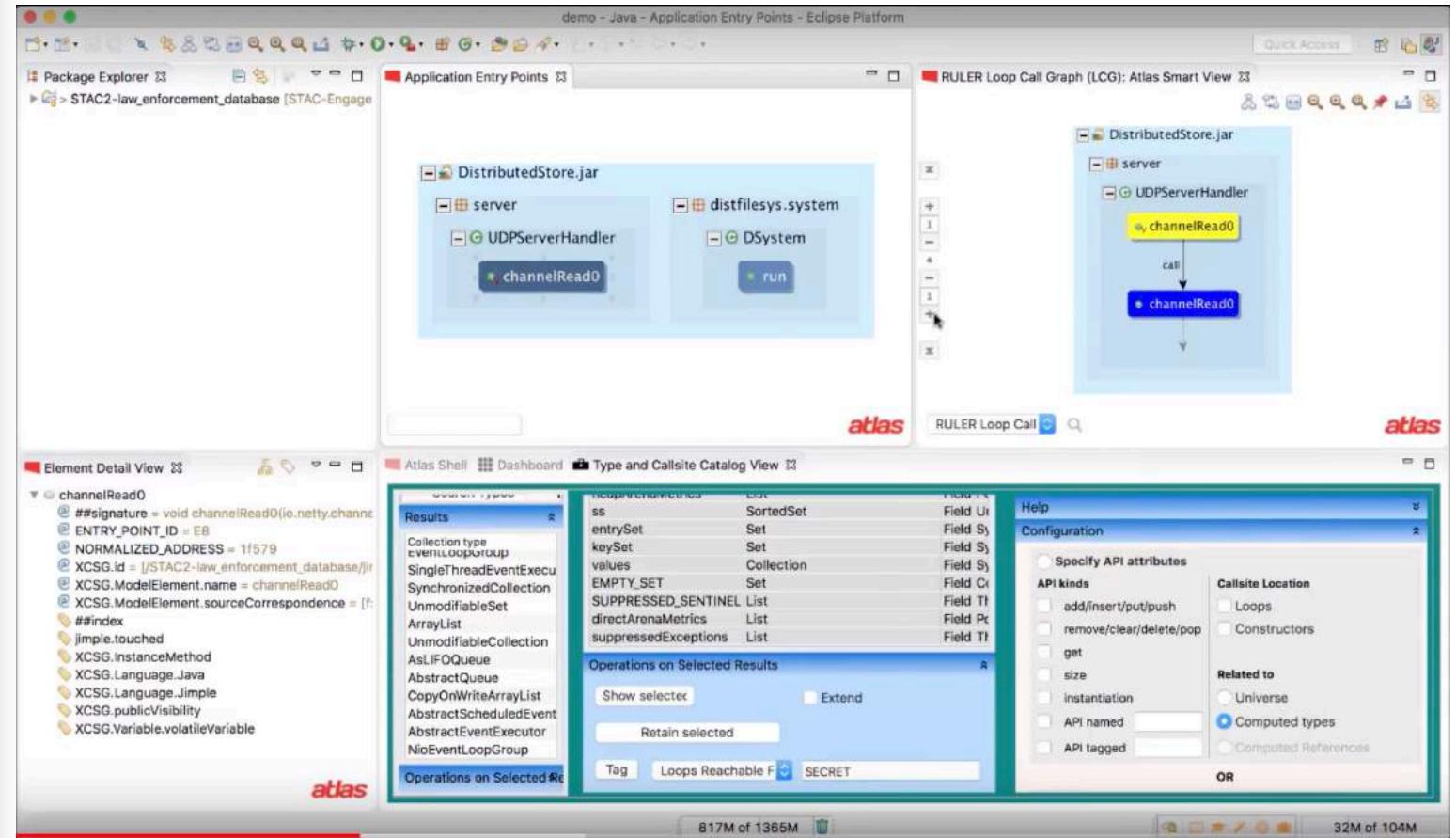
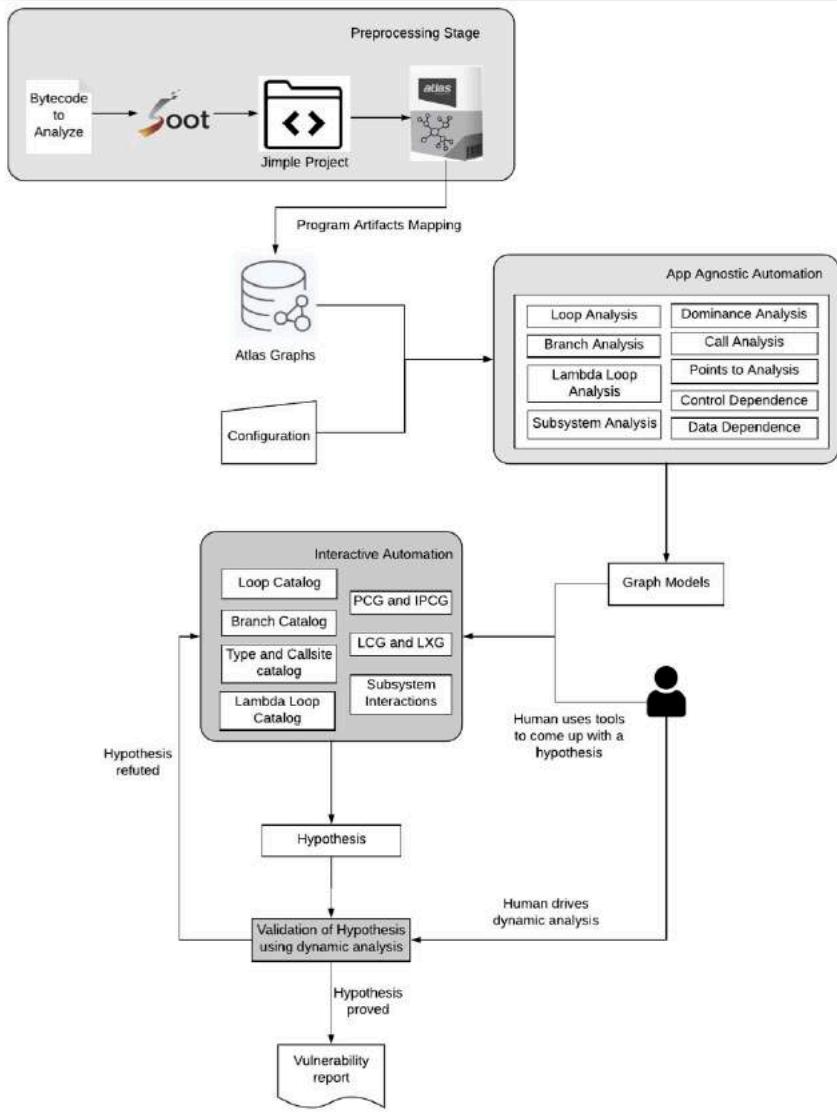
The program aims to **develop new program analysis techniques and tools for identifying vulnerabilities related to the space and time resource usage behavior of algorithms**, specifically, vulnerabilities to *algorithmic complexity and side channel attacks*. STAC seeks to **enable analysts to identify algorithmic resource usage vulnerabilities in software at levels of scale and speed great enough to support a methodical search for them in the software upon which the U.S. government, military, and economy depend.**



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

DARPA STAC Program

Space/Time Analysis for Cybersecurity



STAC Toolbox Demo

<https://www.youtube.com/watch?v= vMAYWTP6kg>

DARPA CHESS Program

Computers and Humans Exploring Software Security

The program aims to develop capabilities to **discover and address vulnerabilities of all types in a scalable, timely, and consistent manner.**

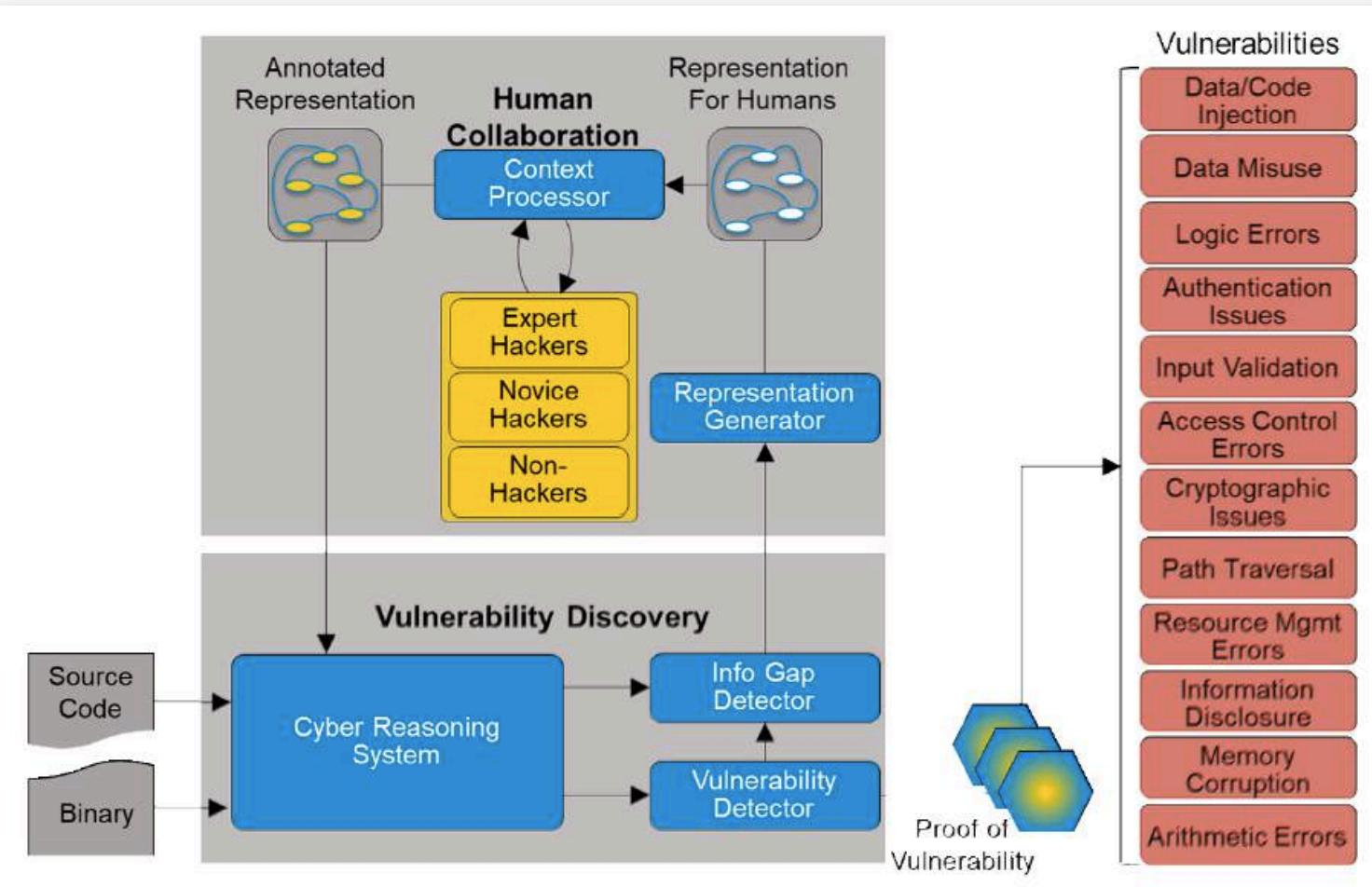
Achieving the necessary *scale and timelines in vulnerability discovery* will require innovative combinations of **automated program analysis techniques with support for advanced computer-human collaboration.** Due to the cost and scarcity of expert hackers, such capabilities must be able to **collaborate with humans of varying skill levels**, even those with no previous hacking experience or relevant domain knowledge.



<https://www.darpa.mil/program/computers-and-humans-exploring-software-security>

DARPA CHESS Program

Computers and Humans Exploring Software Security



Representations for high-order reasoning and computer-human collaboration

Context Processing for employing domain-specific knowledge to empower software analysis and verification

Cyber Reasoning to model the open-ended spectrum of vulnerabilities

Human-on-the-loop balanced use of static and dynamic analyses

Read High-Quality Papers by Computer Science Pioneers



Donald Knuth



Kurt Gödel



Alan Turing



Tony Hoare



Gary Kildall



Robert Tarjan



Fred Brooks



Edsger Dijkstra

and many others ...

For further information and Resources:

 atamrawi  atamrawi.github.io  ahmedtamrawi@gmail.com

 www. <http://www.ece.iastate.edu/kcsl/>

Knowledge-Centric Software Engineering Lab

L-DAR

Publications

- Competitions (1)
- Papers (16)
- Short Courses (2)
- Talks (9)
- Tutorials (9)
- Upcoming (3)

Monthly Activity

- October 2017 (2)
- September 2017 (3)
- August 2017 (1)
- July 2017 (2)
- June 2017 (1)
- March 2017 (3)
- December 2016 (1)
- November 2016 (2)
- October 2016 (4)
- September 2016 (2)
- August 2016 (1)
- May 2016 (4)
- December 2015 (2)
- November 2015 (2)
- October 2015 (1)
- May 2015 (1)
- December 2014 (2)
- October 2014 (1)
- September 2014 (1)
- May 2014 (1)

Authors

- Ahmed Tamrawi
- Akshay Deepak
- Benjamin Holland
- Ganesh Ram Santhanam

Recent research funding has come primarily from DARPA contracts FA8750-12-2-0126 and FA8750-15-2-0080.

Director

- Suresh Kothari (Richardson Professor)

Current Members

- Benjamin Holland (Graduate Student)
- Ganesh Ram Santhanam (Associate Scientist)
- Payas Awadhutkar (Graduate Student)

Past Members

Ahmed Tamrawi, Akshay Deepak, Curtis Ullerich, Daman Singh, Dan Harvey, Dan Stiner, Jeremias Sauceda, Jim Carl, Jon Mathews, Kang Gui, Luke Bishop, Murali Ravirala, Nikhil Ranade, Sandeep Krishnan, Sergio Ferrero, Srinivas Neginali, Tom Deering, Xiaozheng Ma, Yogi Namara, Yunbo Deng, Zach Lones



 www. <https://www.ensoftcorp.com/>

EnSoft Corp (free download of Atlas)



About EnSoft

EnSoft was founded in 2002 with the goal of tackling the growing complexity in software systems. We believe that human intelligence combined with powerful tools is the key to tackling complexity. Today our products and services are used by over 350 companies worldwide including every major automotive, aerospace, and defense company in North America, Europe, and Asia.

Products and Services



SimDiff 4 - Everything you need to collaborate on Simulink models.

SimDiff has become the leading diff and merge tool for Simulink models since its first release in 2005. SimDiff's accuracy, speed, and robustness has made it the preferred choice amongst the world's leading companies in the automotive, aerospace, and defense industries.

Supports all major blocks sets, including Stateflow, dSPACE blocksets, RTW, Xilinx. Runs on Windows or Linux and supports all MATLAB versions.



Modelify - Convert C code to Simulink models.

Modelify is a new technology from EnSoft to convert large C-

OUR CUSTOMERS



EnSoft's products and services are used by companies in North America, Asia, and Europe.

NEWS



Thank you