Demystifying Software Security

A Euler's Method Approach for Analyzing Complex Software

Ahmed Tamrawi

atamrawi atamrawi.github.io ahmedtamrawi@gmail.com

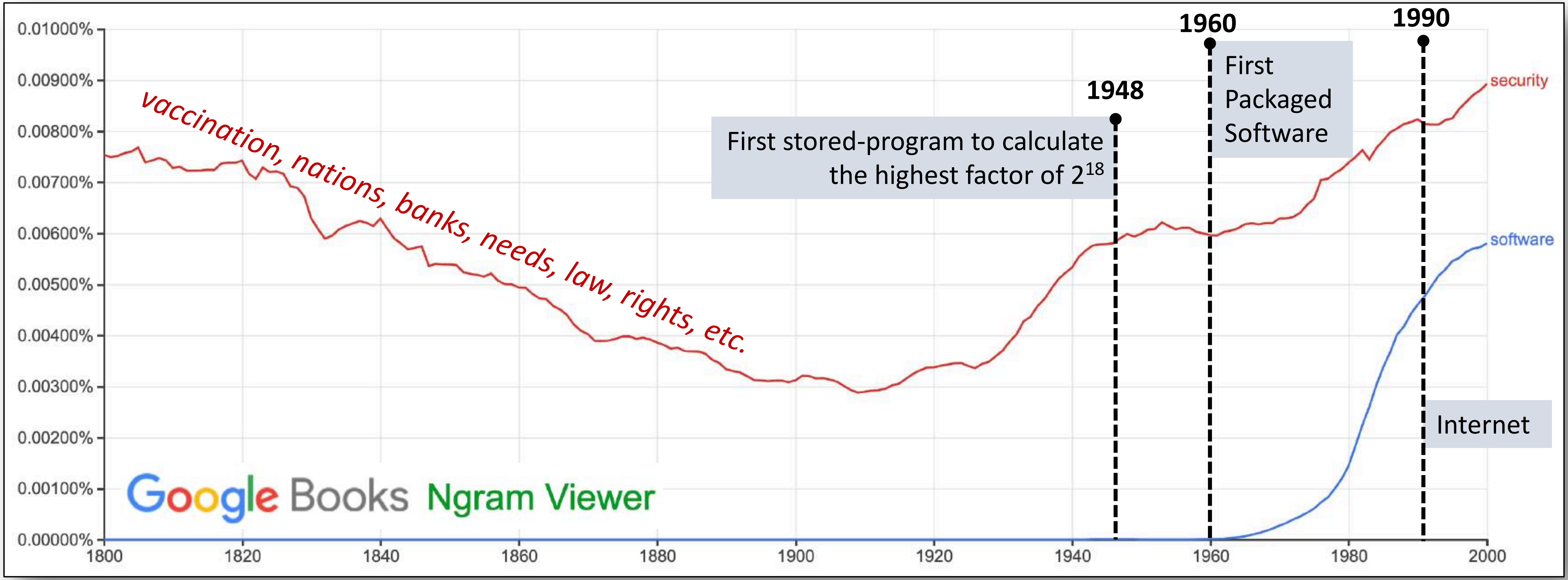
Acknowledgement: *Team members at* ***Iowa State University*** *and* ***EnSoft, DARPA contracts*** *FA8750- 12-2-0126 & FA8750-15-2-0080*

What is Software Security?

What is Software Security?

the state of being free from danger or threat

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

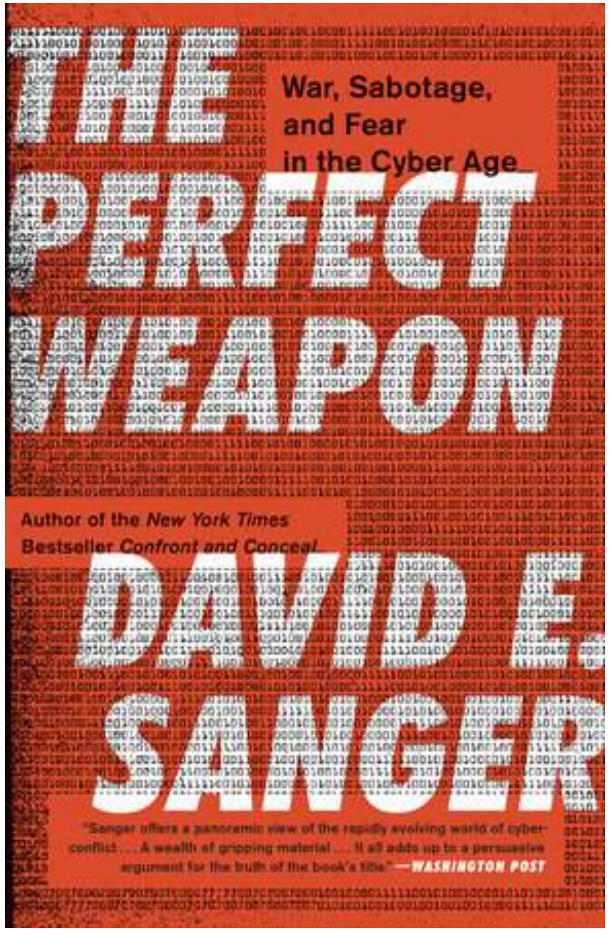


0.01000%
0.00900%
0.00800%
0.00700%
0.00600%
0.00500%
0.00400%
0.00300%
0.00200%
0.00100%
0.00000%
1800
1820
1840
1860
1880
1900
1920
1940
1960
1980
2000
vaccination, nations, banks, needs, law, rights, etc.
1948
First stored-program to calculate the highest factor of 2^{18}
1960
First Packaged Software
1990
Internet
security
software
Google Books Ngram Viewer

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***

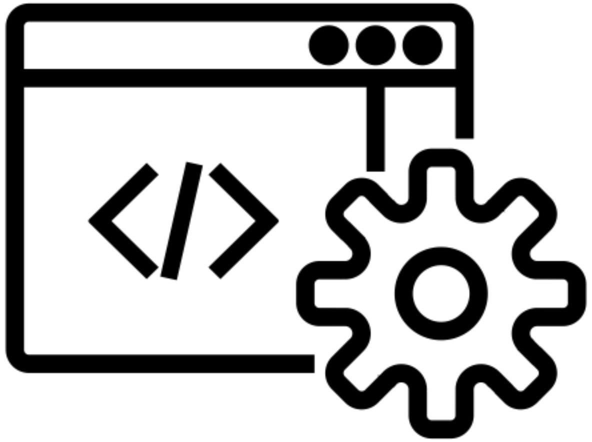
Why do we need Software Security?



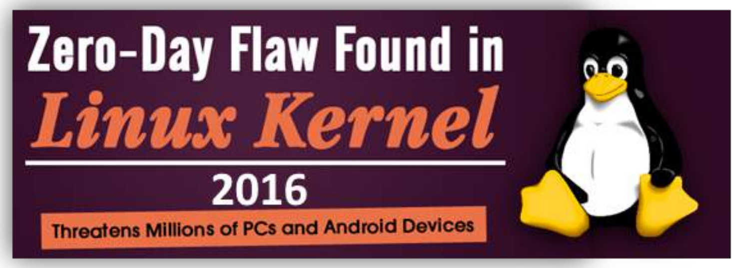
*"Each technology goes through a cycle of development and weaponization, **followed only later** by the formulation of doctrine and occasionally by efforts to control the weapon's use."*

WEAPONIZED INTERNET

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

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

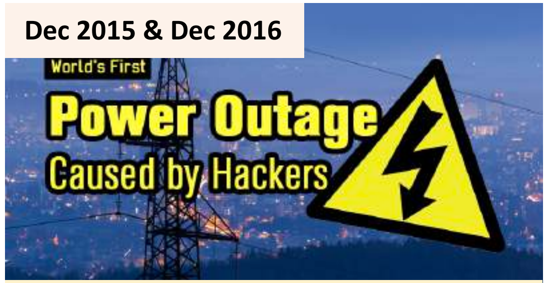
November 29, 2011

HP printers remotely set on fire

August 17, 2009

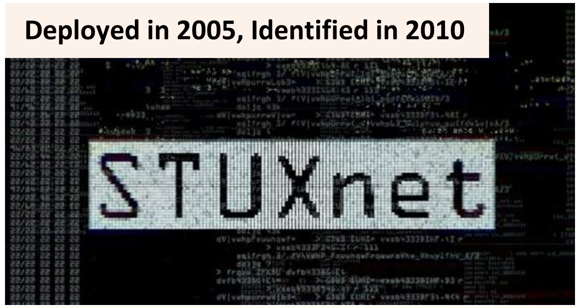
Destruction Sayano-Shushenskaya Hydroelectric Power Plant

Dec 2015 & Dec 2016



Ukraine power grid attacks

Deployed in 2005, Identified in 2010



STUXnet Worm

August 2003

Northeast Power Blackout

August 2003

Davis-Besse Nuclear Power Plant

No need for bombs, ***Plant Malware!***

DARPA *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!
imgflip.com

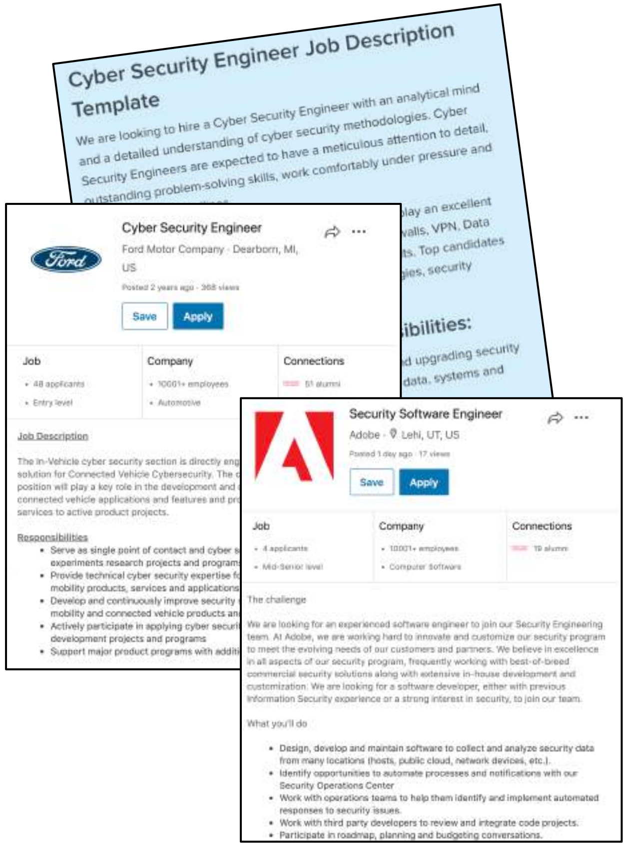
Block User Input!

Sanitize  User Input!

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

How to tell if my Software *is* Secure?

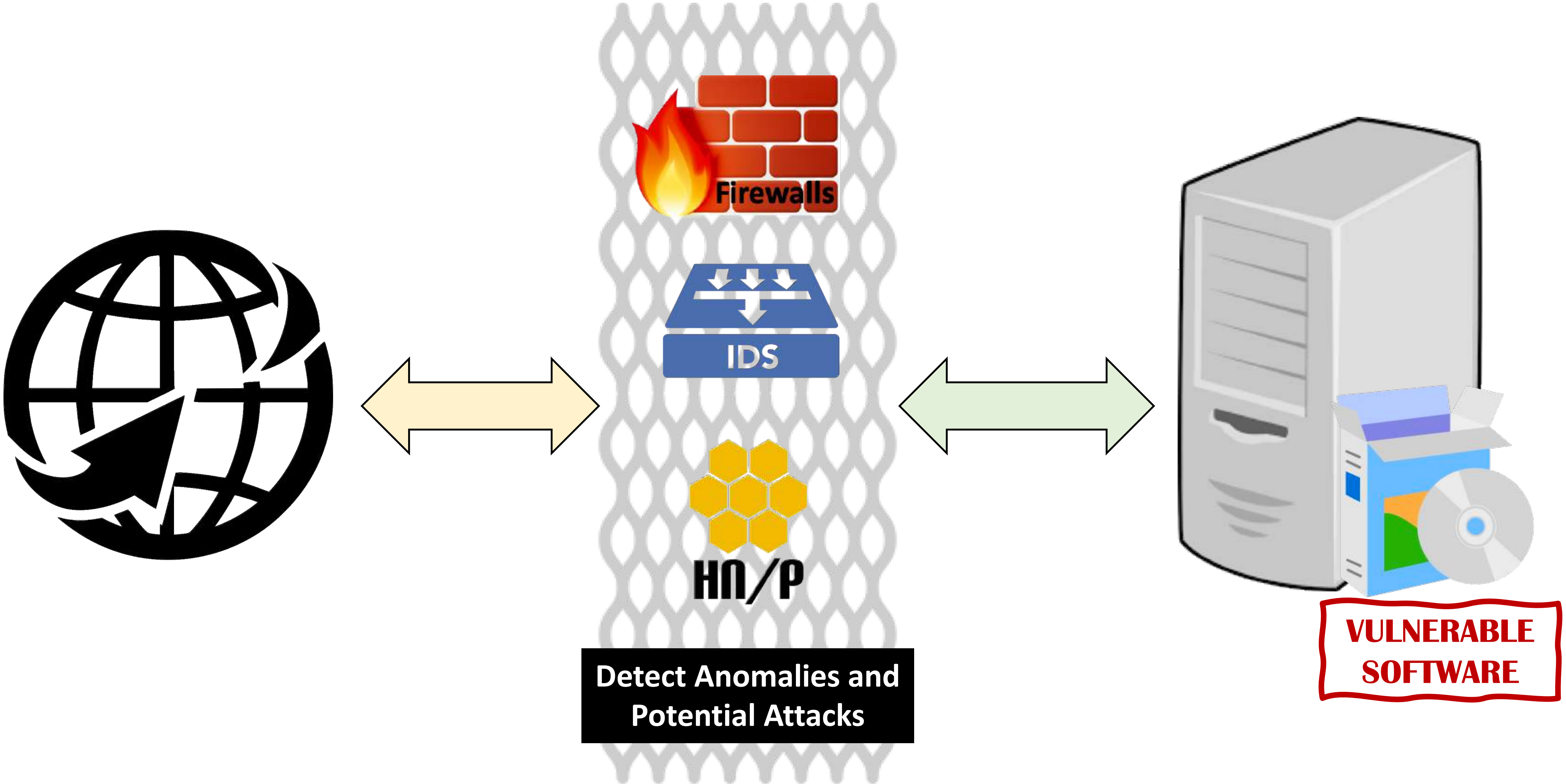
Hire A Cybersecurity Engineer!



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!
imgflip.com

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
[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

2019 Top 25 Vulnerabilities

CWE Common Weakness Enumeration
A Community-Developed List of Software Weakness Types

NVD

EXPLOIT DATABASE

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

```
int a1 = 1, a2 = 2;
int y = 2;
bool C1 = true;
bool C2 = false;
bool C3 = true;
void foo1() {
    int x = a1 + a2;
    int d = a1;
    if(C1){
        x = a1;
    }else{
        x = a2 - 1;
    }

    if(C2){
        if(C3){
            y = a1;
        }else{
            d = d - a1;
        }
    }else{
        d = d + 1;
    }
    int z = x / d;
}
```

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

```
int a1 = 1, a2 = 2;
int y = 2;
bool C1 = true;
bool C2 = false;
bool C3 = true;
void foo1() {
    int x = a1 + a2;
    int d = a1;
    if(C1){
        x = a1;
    }else{
        x = a2 - 1;
    }

    if(C2){
        if(C3){
            y = a1;
        }else{
            d = d - a1;
        }
    }else{
        d = d + 1;
    }
    int z = x / d;
}
```

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

```
int a1 = 1, a2 = 2;
int y = 2;
bool C1 = true;
bool C2 = false;
bool C3 = true;
void foo1() {
    int x = a1 + a2;
    int d = a1;
    if(C1){
        x = a1;
    }else{
        x = a2 - 1;
    }

    if(C2){
        if(C3){
            y = a1;
        }else{
            d = d - a1;
        }
    }else{
        d = d + 1;
    }
    int z = x / d;
}
```

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

```
int a1 = 1, a2 = 2;
int y = 2;
bool C1 = true;
bool C2 = false;
bool C3 = true;
void foo1() {
    int x = a1 + a2;
    int d = a1;
    if(C1){
        x = a1;
    }else{
        x = a2 - 1;
    }

    if(C2){
        if(C3){
            y = a1;
        }else{
            d = d - a1;
        }
    }else{
        d = d + 1;
    }
    int z = x / d;
}
```

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

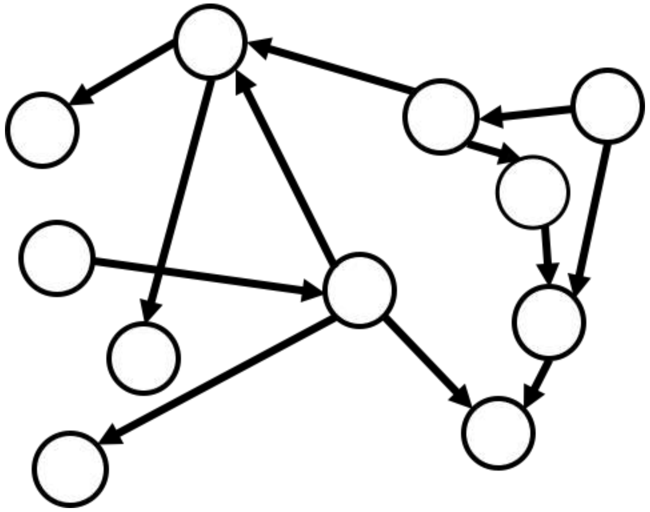
What is a feasible execution path?

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

```
int a1 = 1, a2 = 2;
int y = 2;
bool C1 = true;
bool C2 = false;
bool C3 = true;
void foo1() {
    int x = a1 + a2;
    int d = a1;
    if(C1){
        x = a1;
    }else{
        x = a2 - 1;
    }

    if(C2){
        if(C3){
            y = a1;
        }else{
            d = d - a1;
        }
    }else{
        d = d + 1;
    }
    int z = x / d;
}
```

*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 }
```

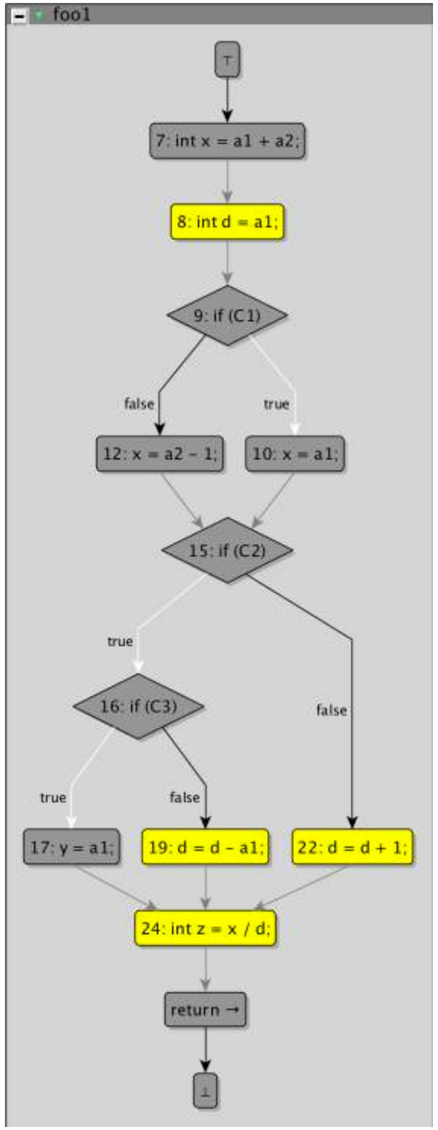
Function `foo1`

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**



Control Flow Graph (CFG)

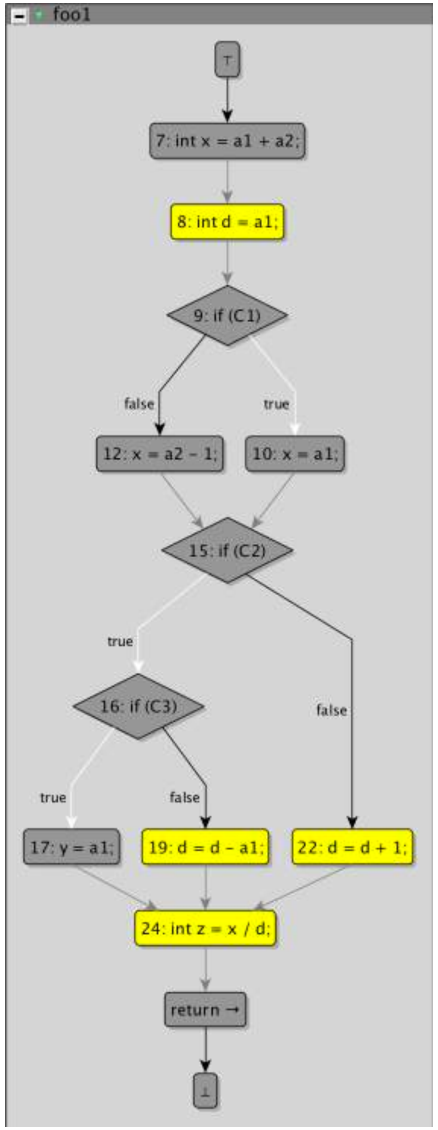
What is a feasible execution path?

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

```
int a1 = 1, a2 = 2;
int y = 2;
bool C1 = true;
bool C2 = false;
bool C3 = true;
void foo1() {
    int x = a1 + a2;
    int d = a1;
    if(C1){
        x = a1;
    }else{
        x = a2 - 1;
    }

    if(C2){
        if(C3){
            y = a1;
        }else{
            d = d - a1;
        }
    }else{
        d = d + 1;
    }
    int z = x / d;
}
```

Function `foo1`



Control Flow Graph (CFG)

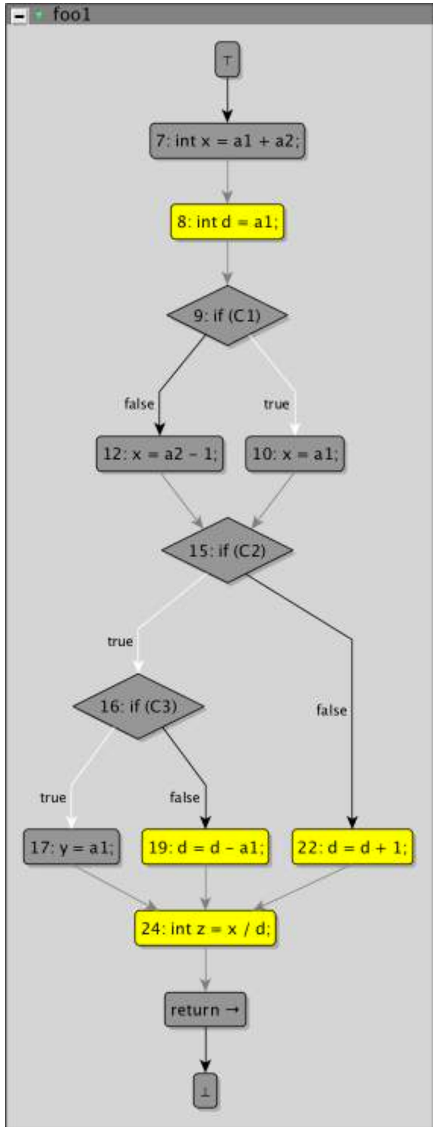
Division-By-Zero (DBZ) Vulnerability?

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

$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c_3}], 19, 24$

$B_2 : 7, 8, 9[\bar{c_1}], 12, 15[c_2], 16[\bar{c_3}], 19, 24$

$B_3 : 7, 8, 9[c_1], 10, 15[\bar{c_2}], 22, 24$

$B_4 : 7, 8, 9[\bar{c_1}], 12, 15[\bar{c_2}], 22, 24$

$B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$

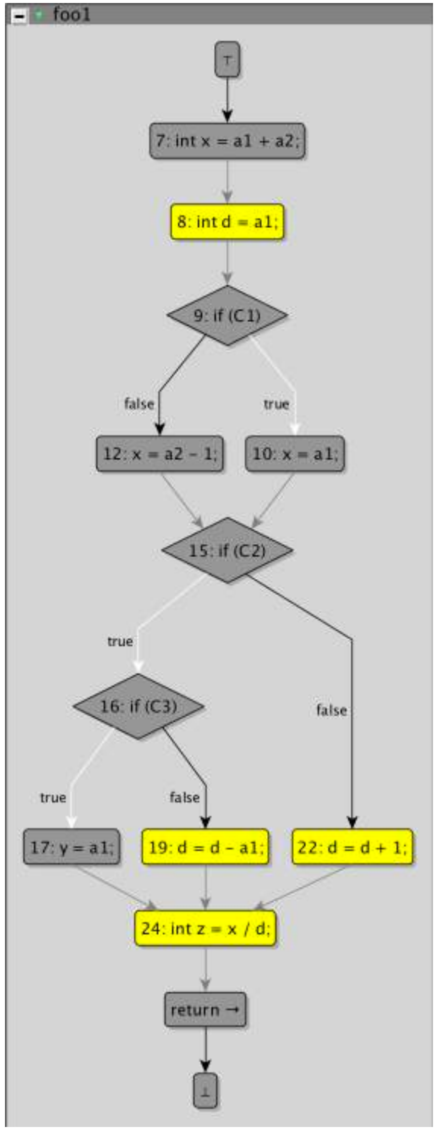
$B_6 : 7, 8, 9[\bar{c_1}], 12, 15[c_2], 16[c_3], 17, 24$

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?

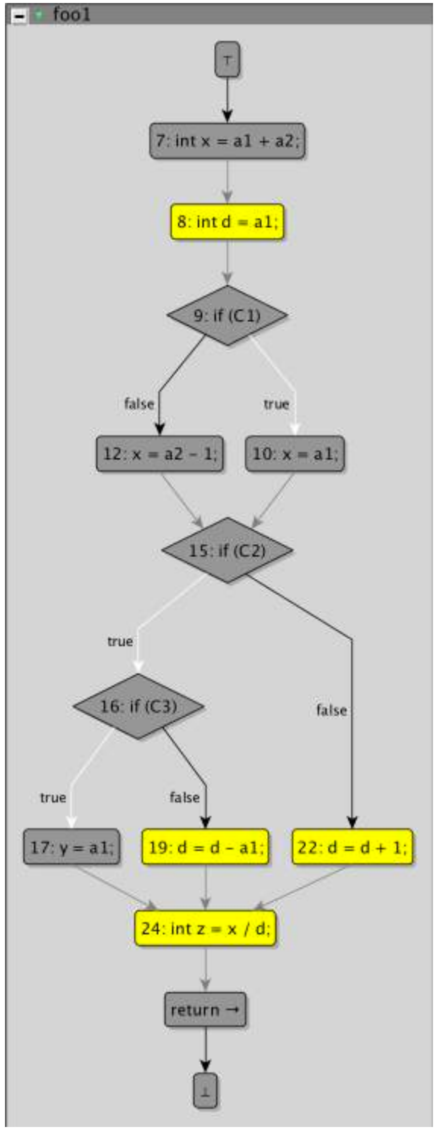
Six Possible Execution Paths
$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c_3}], 19, 24$
$B_2 : 7, 8, 9[\bar{c_1}], 12, 15[c_2], 16[\bar{c_3}], 19, 24$
$B_3 : 7, 8, 9[c_1], 10, 15[\bar{c_2}], 22, 24$
$B_4 : 7, 8, 9[\bar{c_1}], 12, 15[\bar{c_2}], 22, 24$
$B_5 : 7, 8, 9[c_1], 10, 15[c_2], 16[c_3], 17, 24$
$B_6 : 7, 8, 9[\bar{c_1}], 12, 15[c_2], 16[c_3], 17, 24$

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

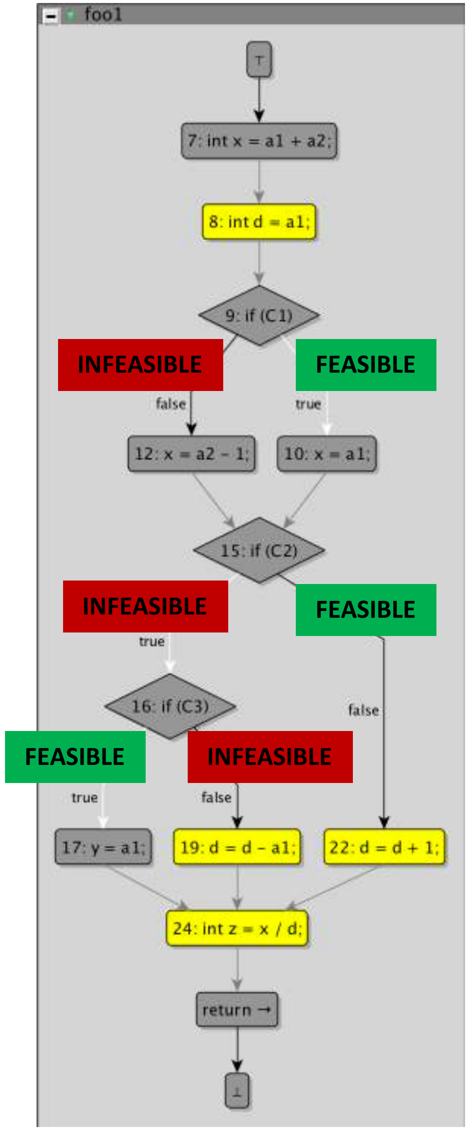
*Are all **feasible**, in other words, are all could be executed at run-time?*

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

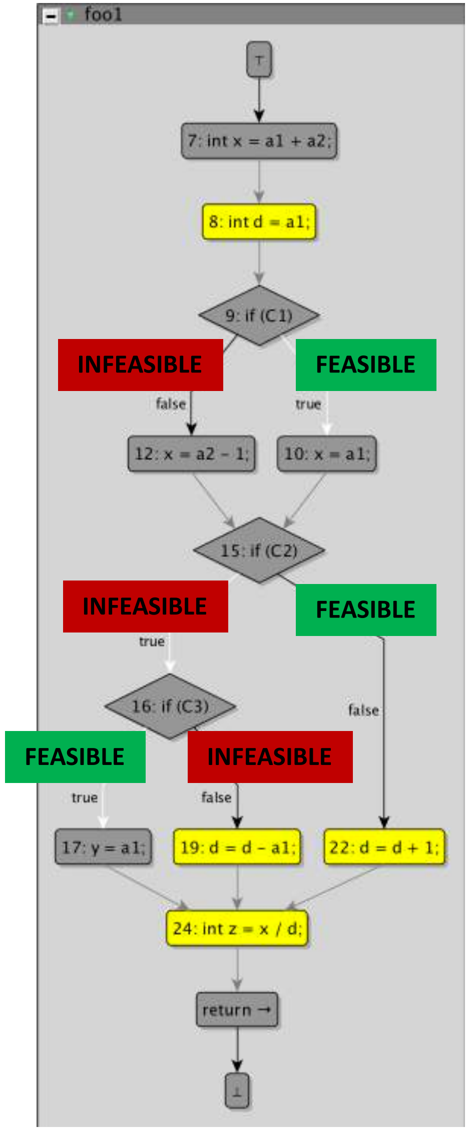
*Are all **feasible**, in other words, are all could be executed at run-time?*

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	$B_1 : 7, 8, 9[c_1], 10, 15[c_2], 16[\bar{c_3}], 19, 24$	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

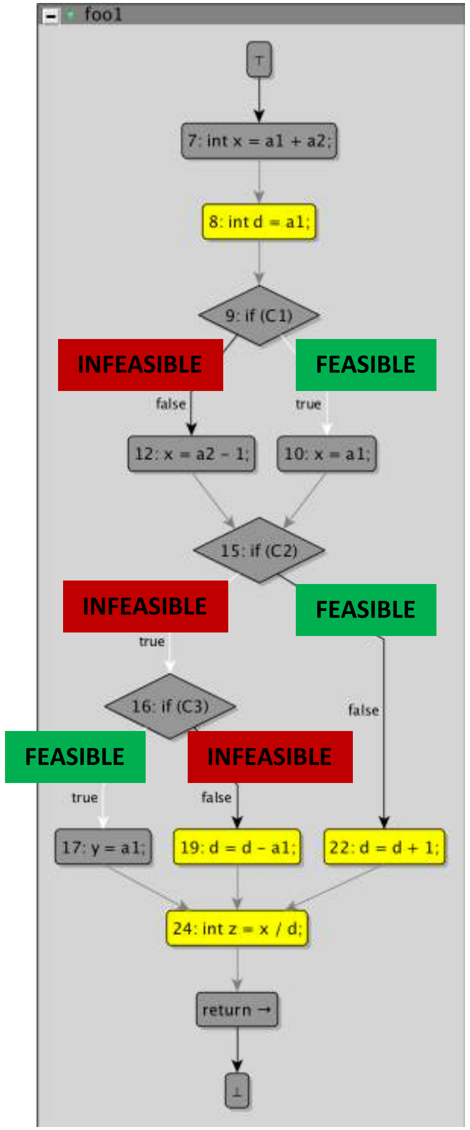
Check if values propagated on feasible vulnerable path can result on DBZ?

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

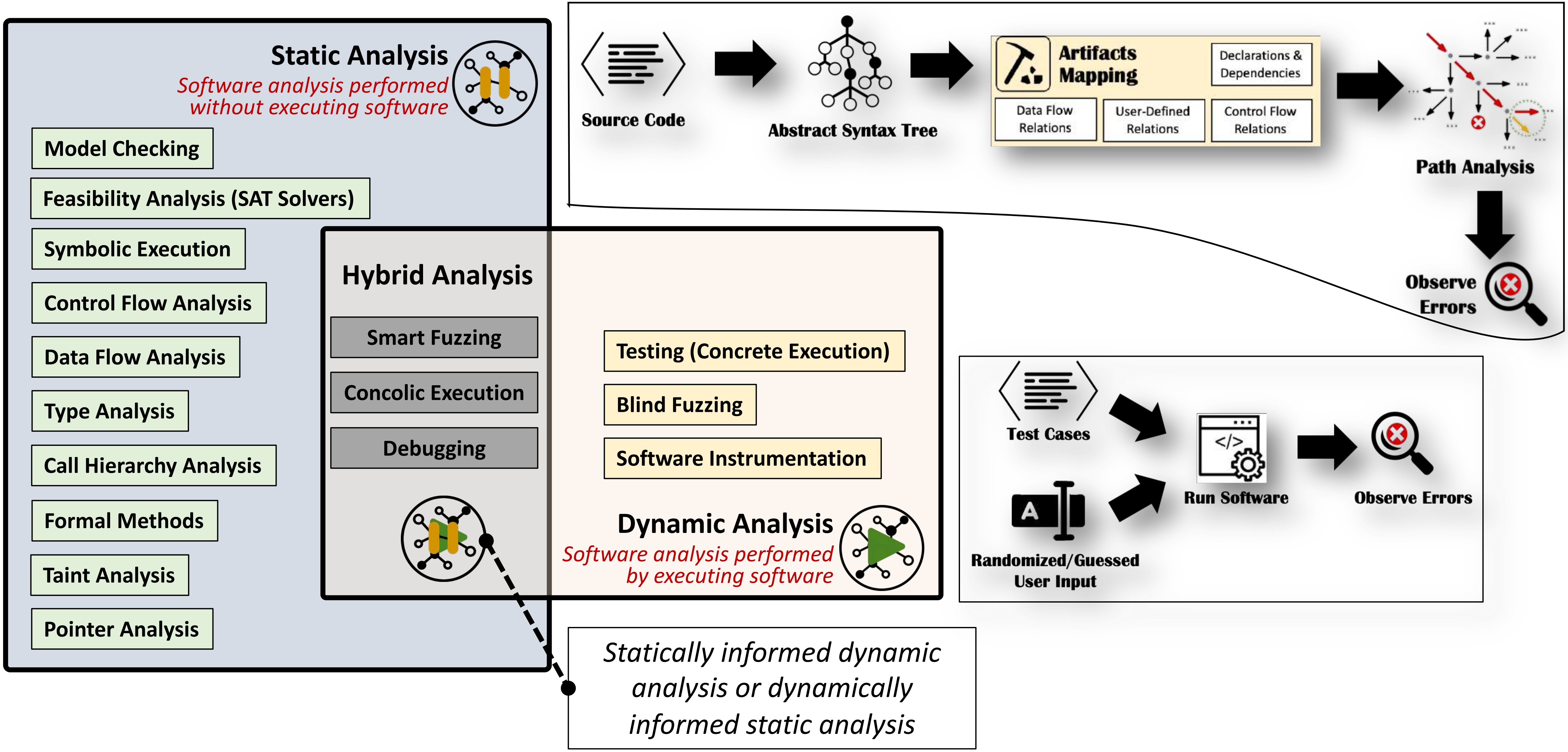
Software Analysis

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**~~*?*

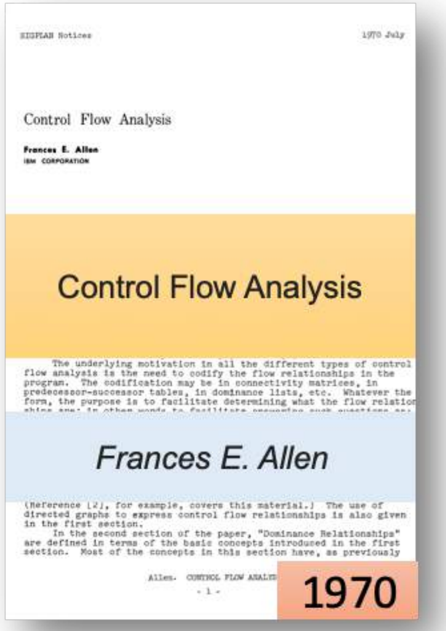
What are **software analysis strategies***?*

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




```
int a1 = 1, a2 = 2;
int y = 2;
bool C1 = true;
bool C2 = false;
bool C3 = true;
void foo1() {
    int x = a1 + a2;
    int d = a1;
    if(C1){
        x = a1;
    }else{
        x = a2 - 1;
    }

    if(C2){
        if(C3){
            y = a1;
        }else{
            d = d - a1;
        }
    }else{
        d = d + 1;
    }
    int z = x / d;
}
```

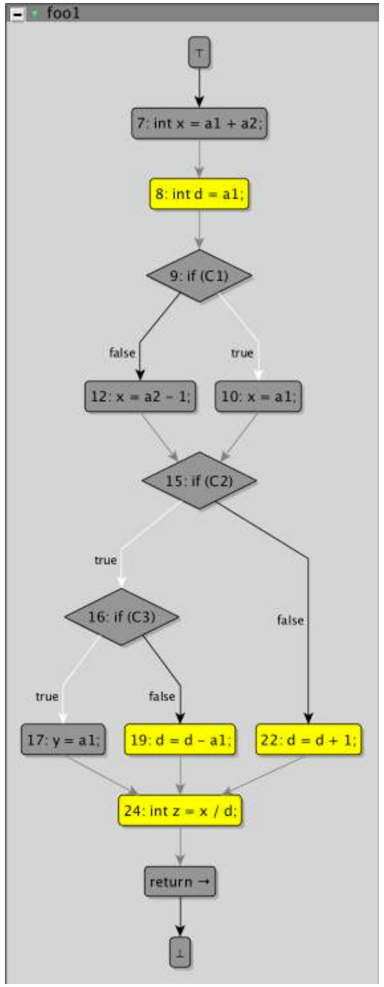
Function foo1

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;
25 }
```

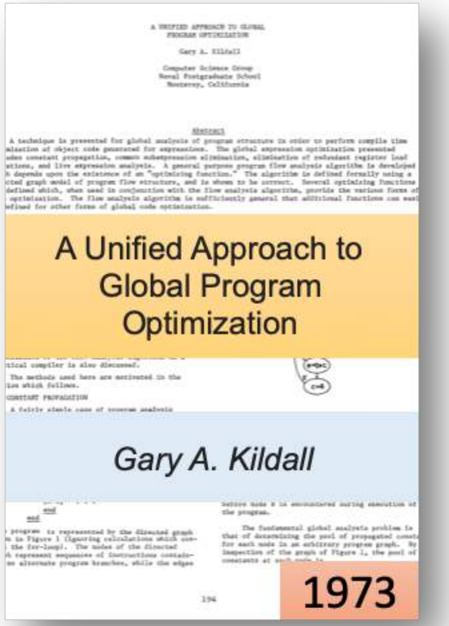
How did we identify the statements at lines **8, 19, and 22** to be relevant to for the analysis of **the potential DBZ at line 24**?

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;
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**?

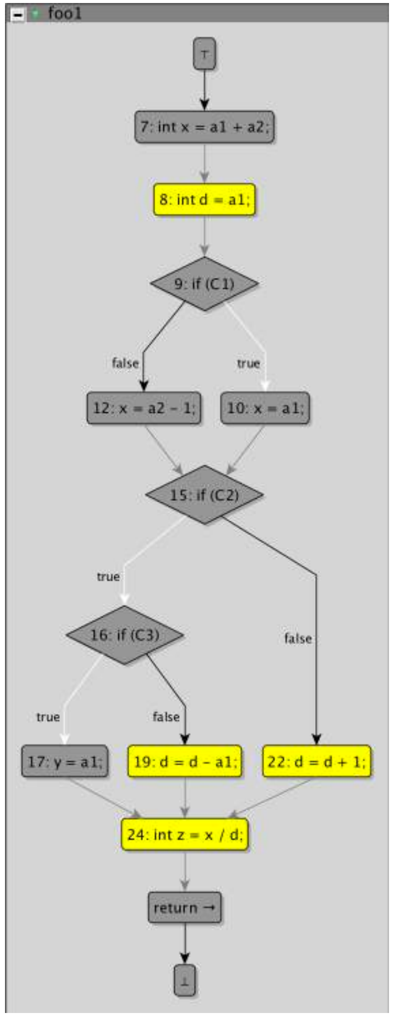


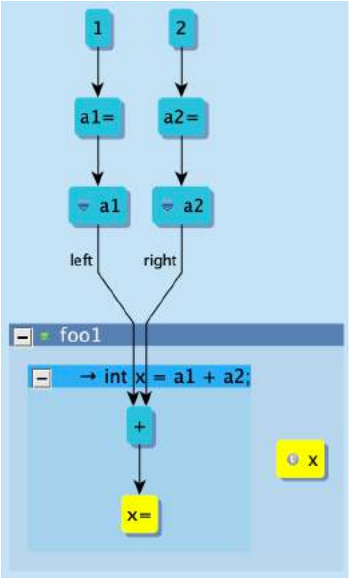
*set up **data-flow equations** for each node of the **control flow graph** and solve them by repeatedly calculating the output from the input locally at each node until the whole system/program stabilizes (reaches a fixpoint)*

Static Analysis: *Data Flow Analysis*

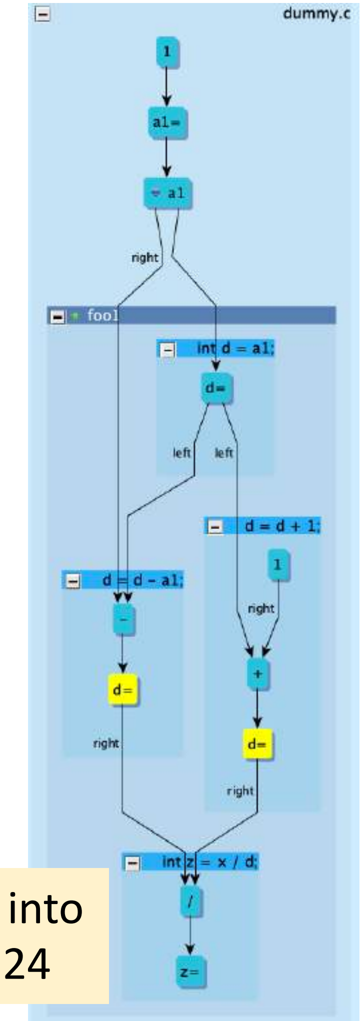
gathers information about the possible set of values calculated at various points in a computer program

```
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;
25  }
```

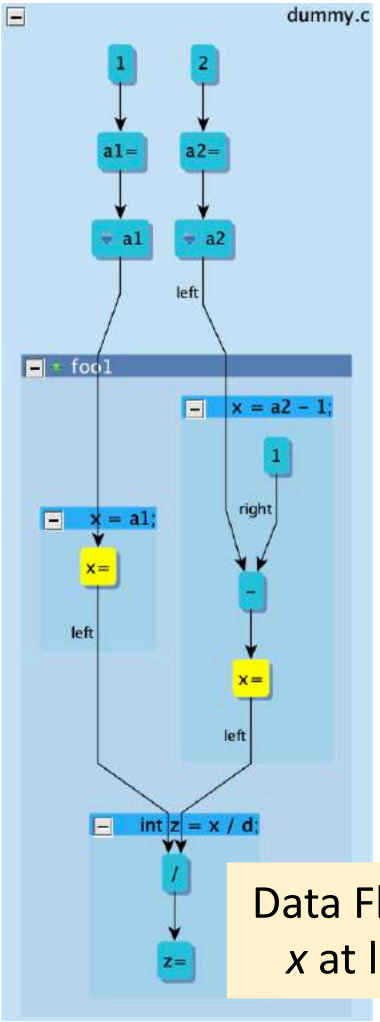




Data Flow into
x at line 7



Data Flow into
d at line 24



Data Flow into
x at line 24

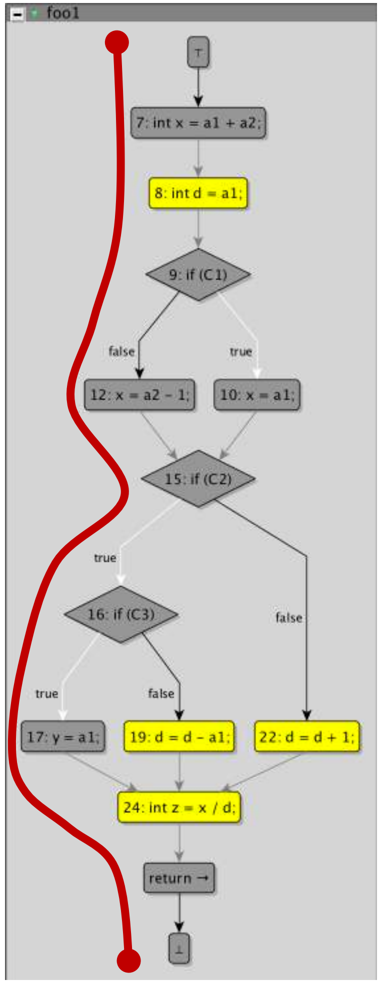
Static Analysis: *Feasibility Analysis*

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

```
int a1 = 1, a2 = 2;
int y = 2;
bool C1 = true;
bool C2 = false;
bool C3 = true;
void foo1() {
    int x = a1 + a2;
    int d = a1;
    if(C1){
        x = a1;
    }else{
        x = a2 - 1;
    }

    if(C2){
        if(C3){
            y = a1;
        }else{
            d = d - a1;
        }
    }else{
        d = d + 1;
    }
    int z = x / d;
}
```

Function foo1



Control Flow Graph (CFG)

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

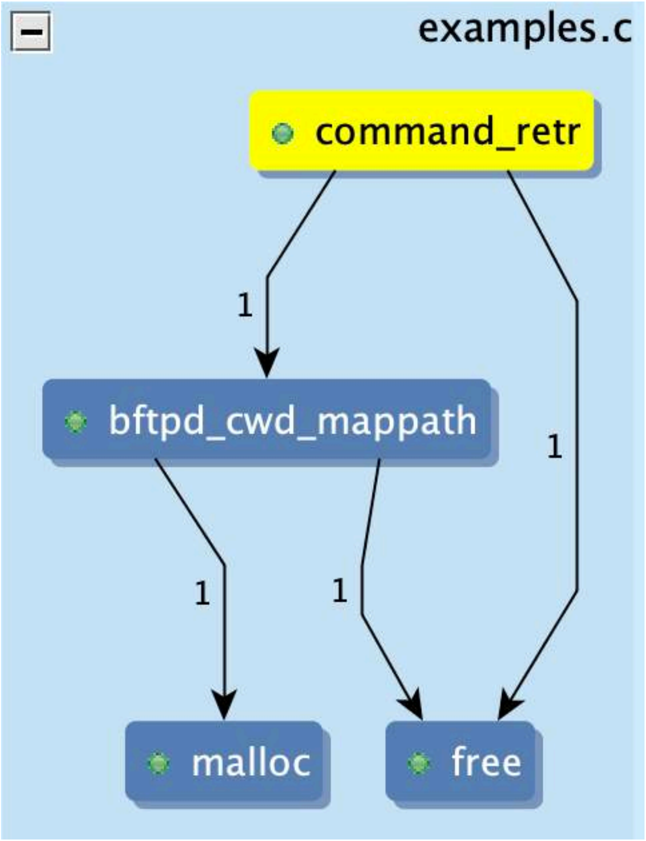
Static Analysis: *Call Hierarchy Analysis*

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

The resultant graph is called the call graph

```c
void bftpd_cwd_mappath() {
    char *result = malloc();
    if(!result) {
        return;
    }
    if(!path2) {
        free();
    }
}

void command_retr() {
    bftpd_cwd_mappath();
    free();
}
```



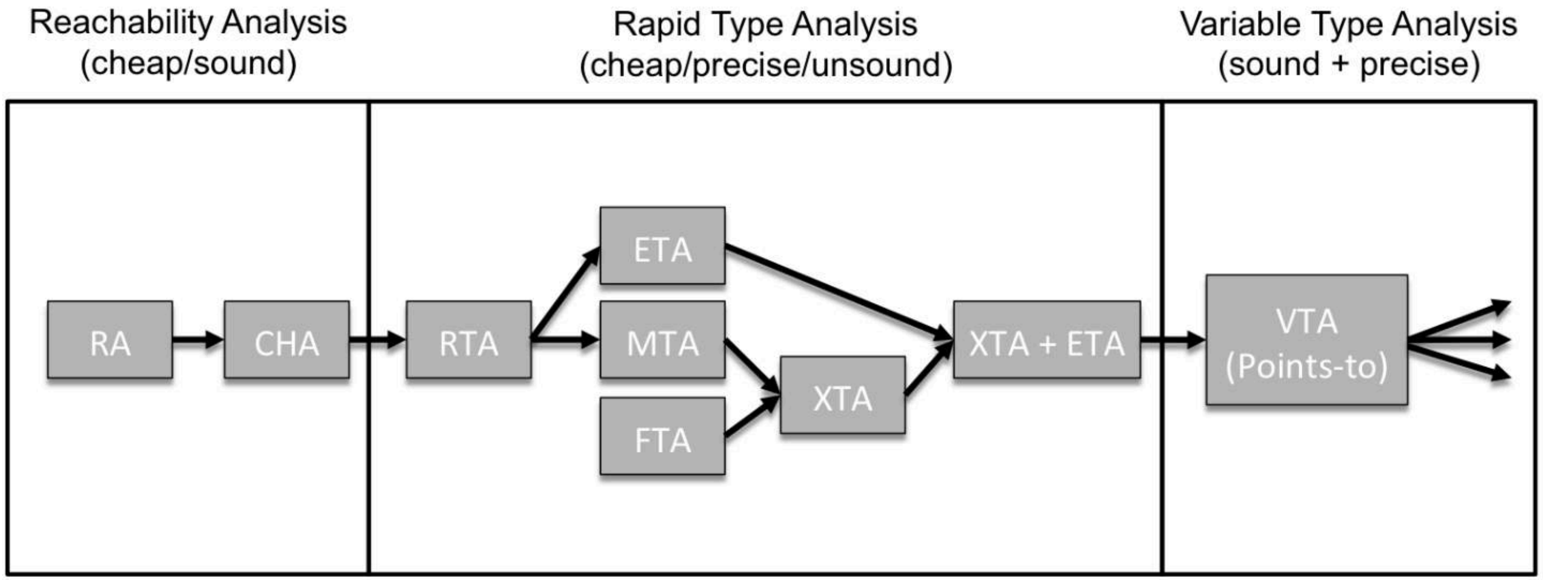
Call Graph

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.

```
void foo() {
    int y = read();
    if(y > 10 && y < -10) {
        return;
    }
    int z = y * 2;
    if(z == 12) {
        crash();
    } else {
        printf("OK");
    }
}
```

$y \in \mathbb{N}$

$y \in]-\infty, -10[\cup]10, +\infty[$

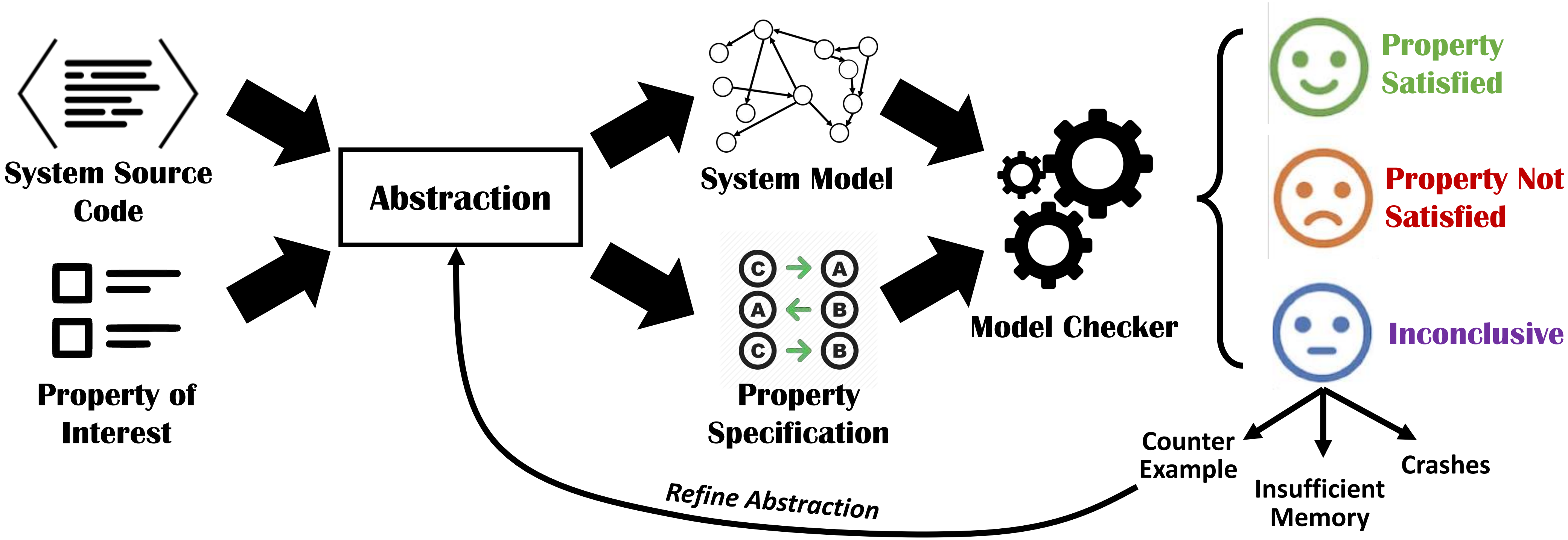
$y \in [-10, 9]$

$z \in [-20, 18]$

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



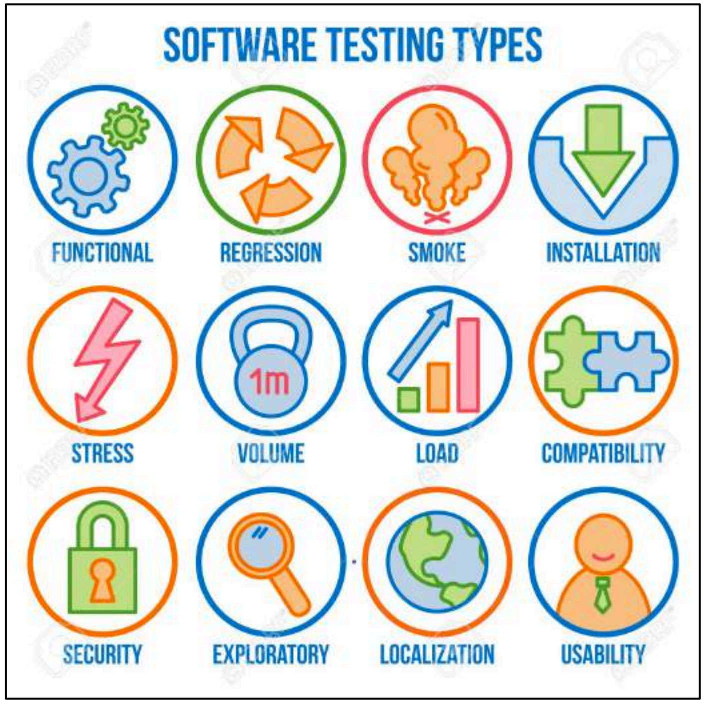
For updated list of Model Checkers:
https://en.wikipedia.org/wiki/List_of_model_checking_tools

Dynamic Analysis: *Testing*

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

Ultimate Goal is to achieve:

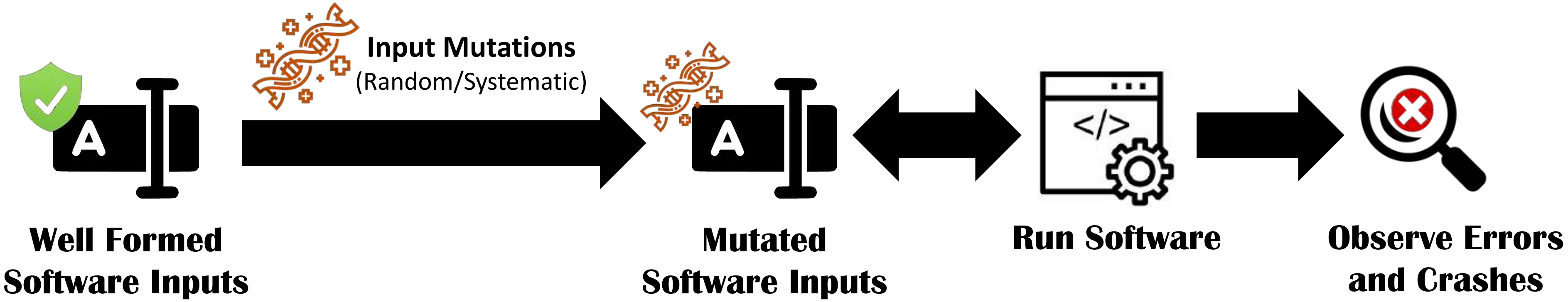
- **Requirements Coverage**
- **Path Coverage**

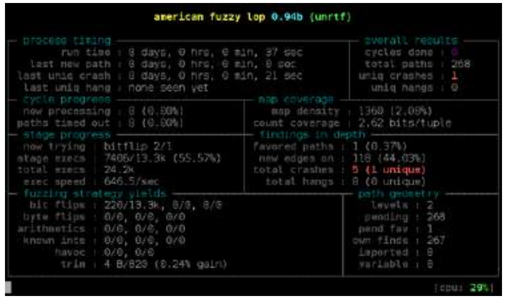


For more info: https://www.fuzzingbook.org/

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.





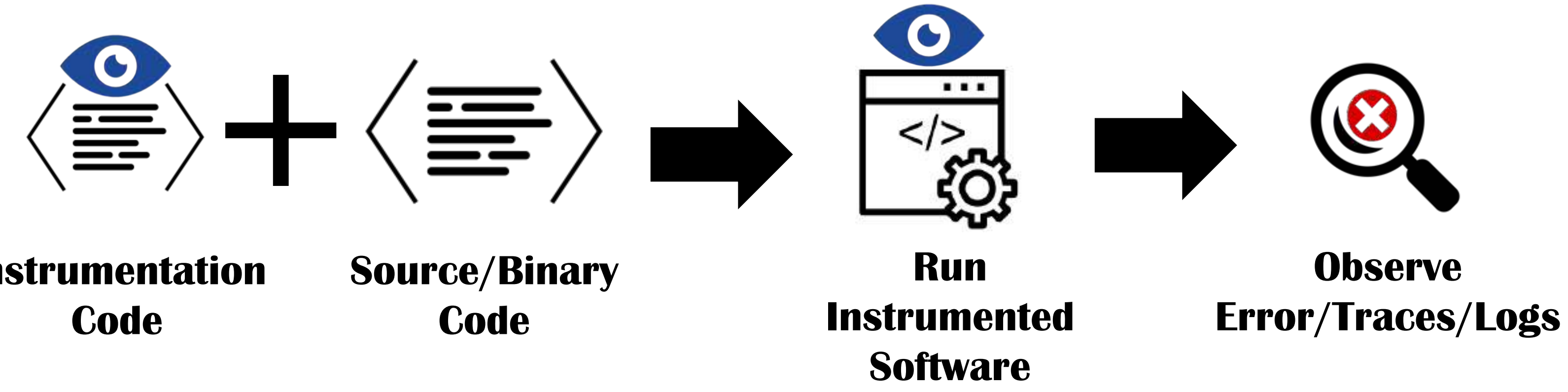
Ultimate Goal is to achieve:

- **Requirements Coverage**
- **Path Coverage**

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

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

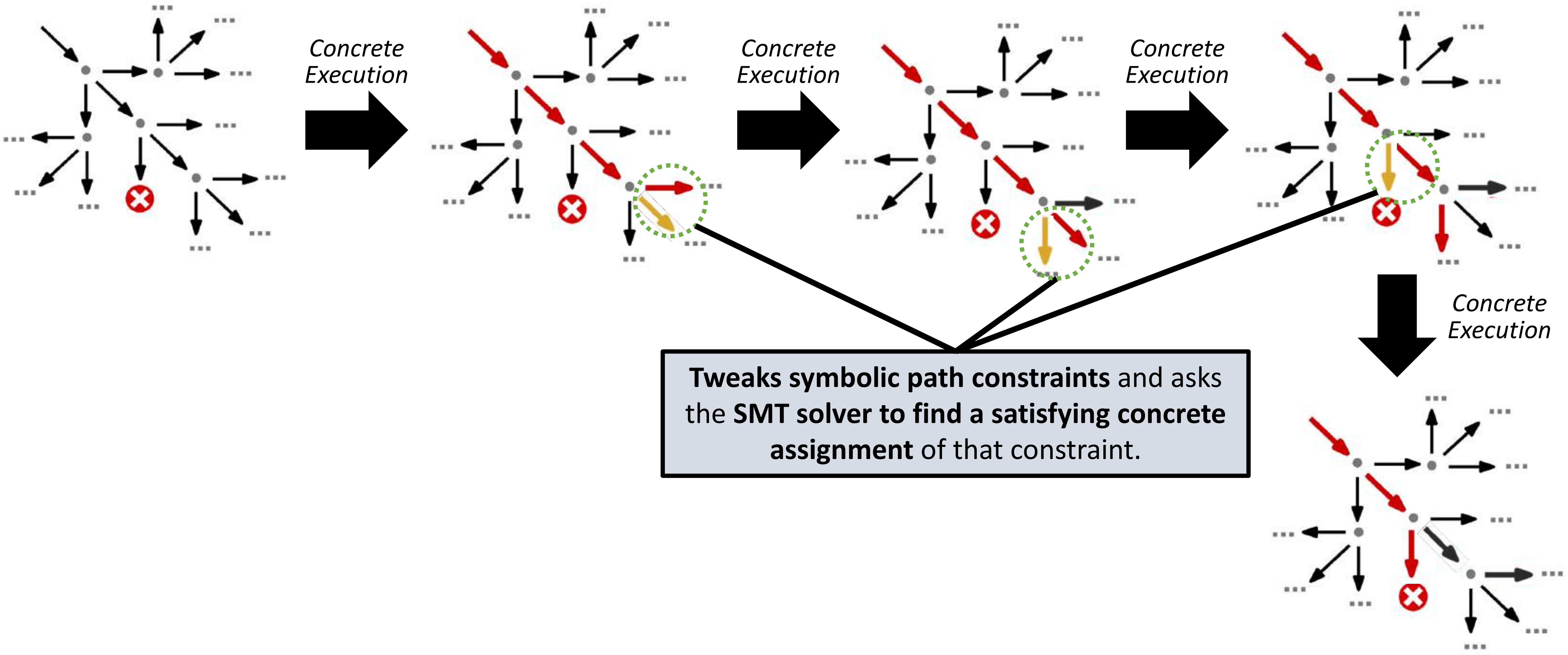


Ultimate Goal is to achieve:

- 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*

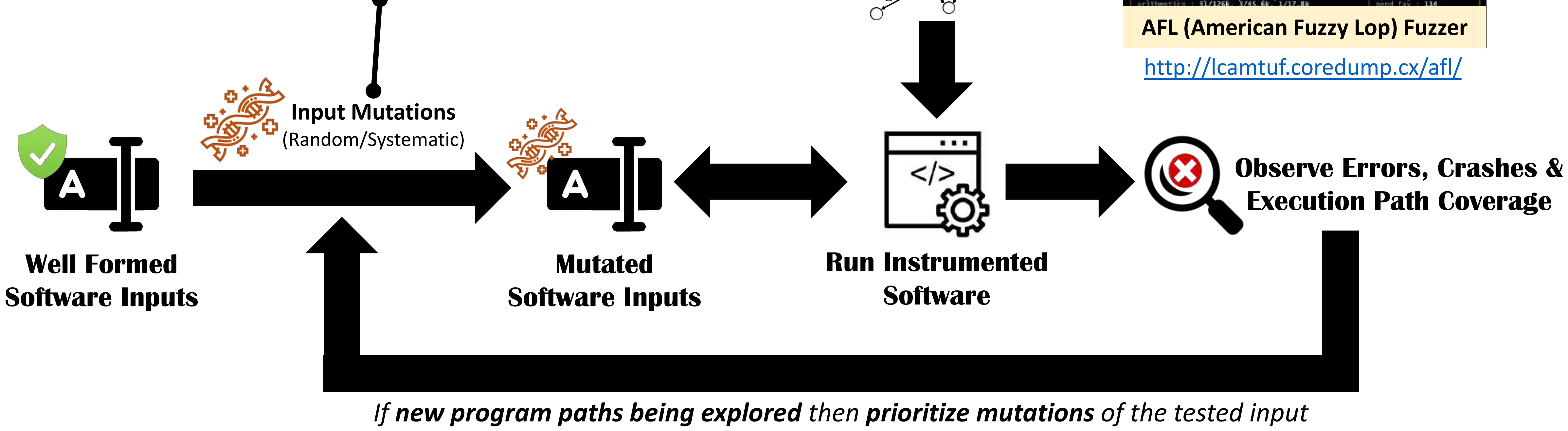
is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

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

AFL (American Fuzzy Lop) Fuzzer

http://lcamtuf.coredump.cx/afl/



*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***?*~~

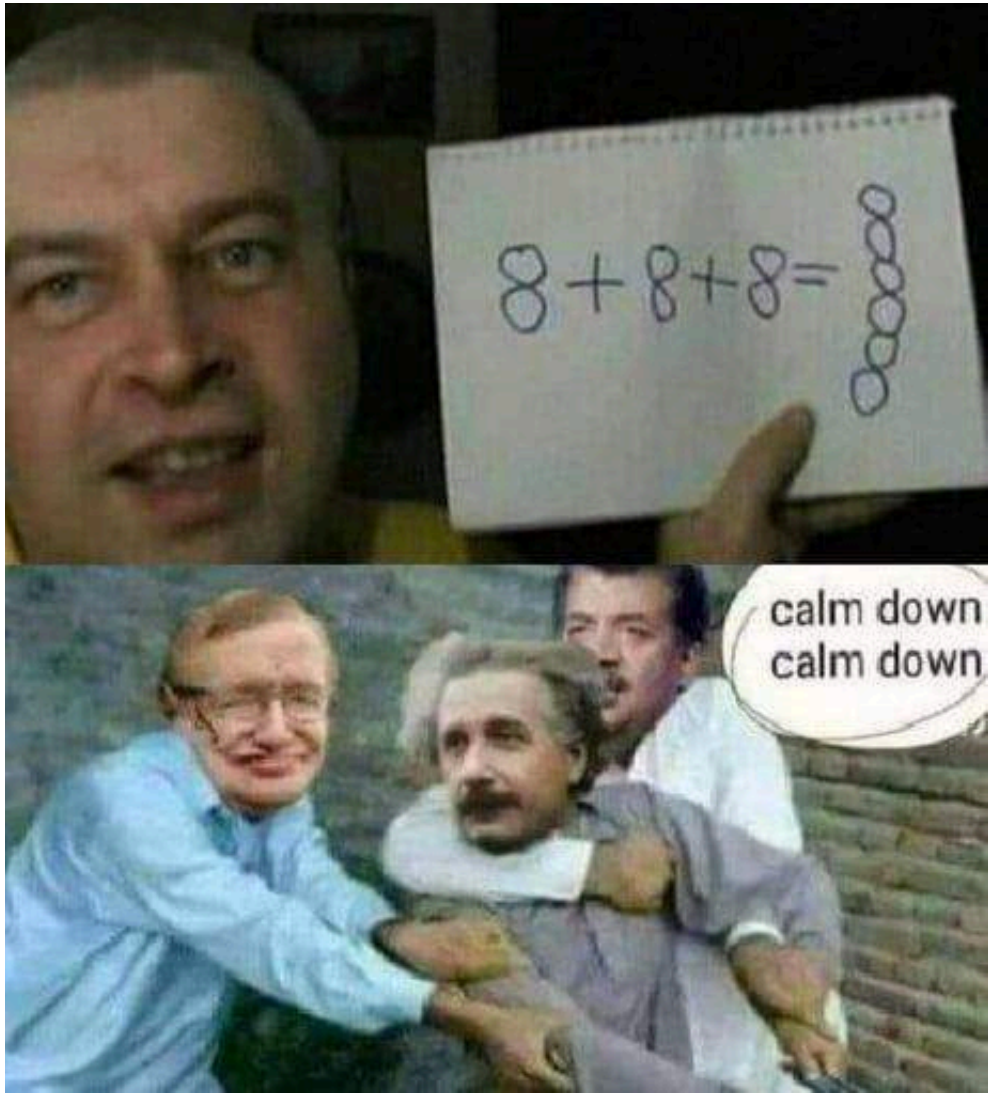
~~*What are* **software analysis strategies***?*~~

Software Analysis

① Choose the **security property you want to proof its conformance** or the **security vulnerability you want to proof its absence**.

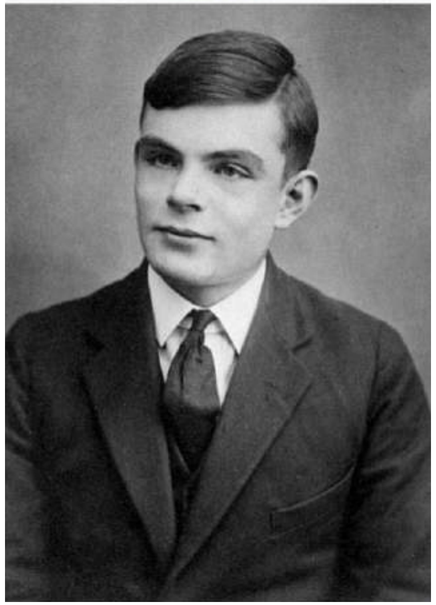
② Pick a **software analysis strategy** or a combination of strategies to verify property conformance or vulnerability absence on each **feasible execution path**.

What is next?

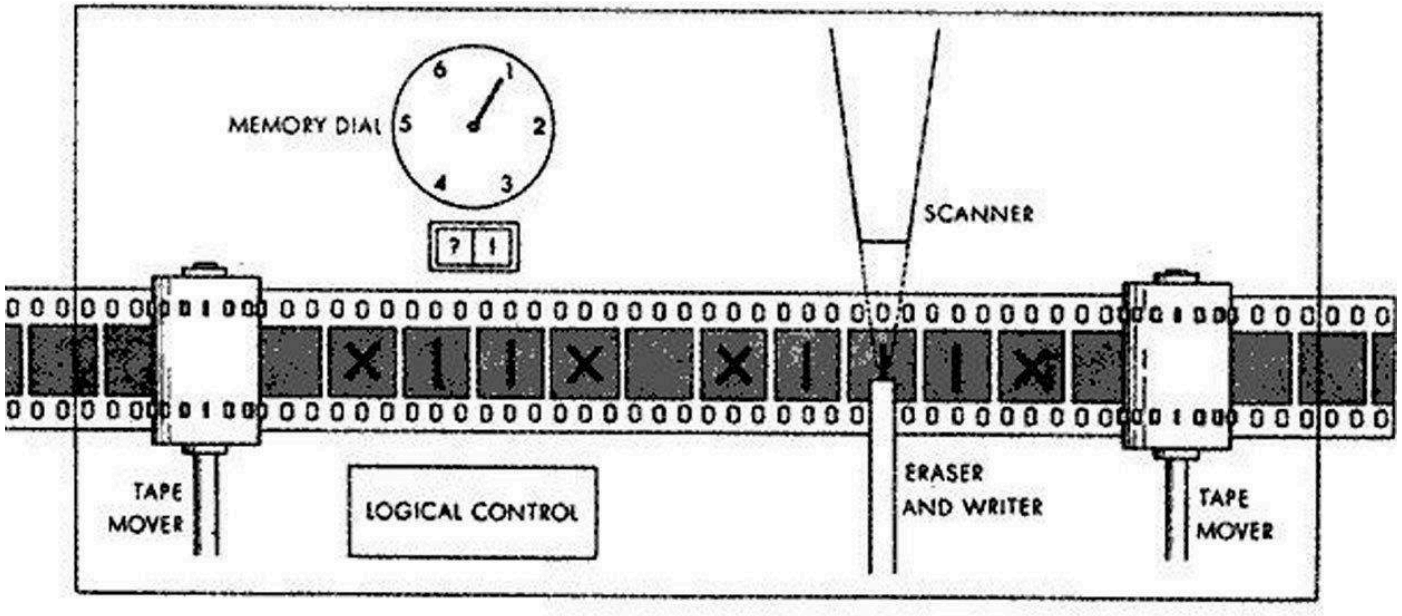

8+8+8=88888
calm down
calm down

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



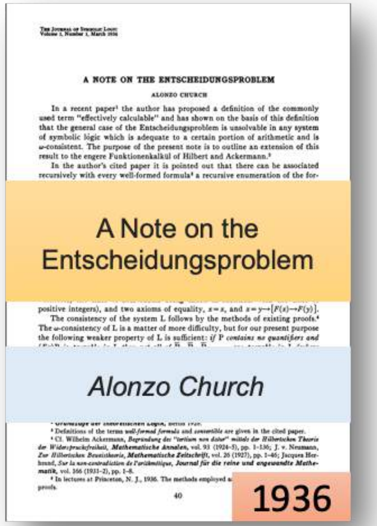
Alan Turing

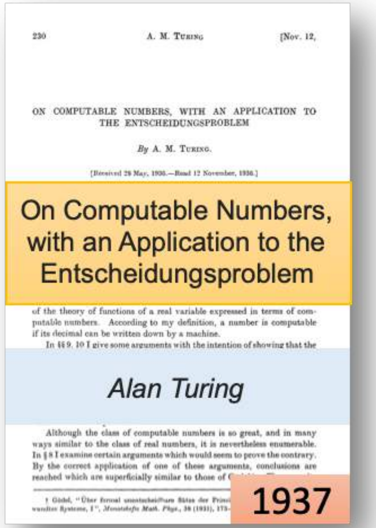


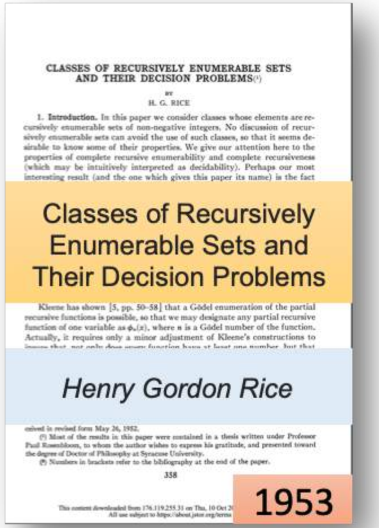
Turing Machine

Note #1

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

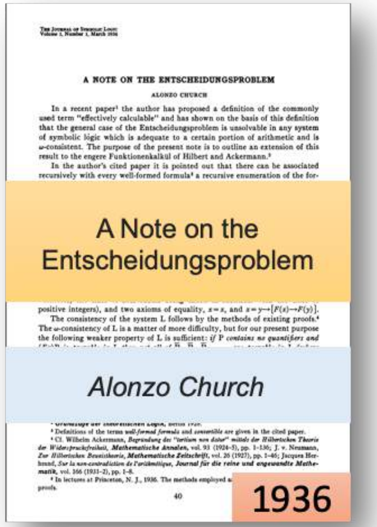


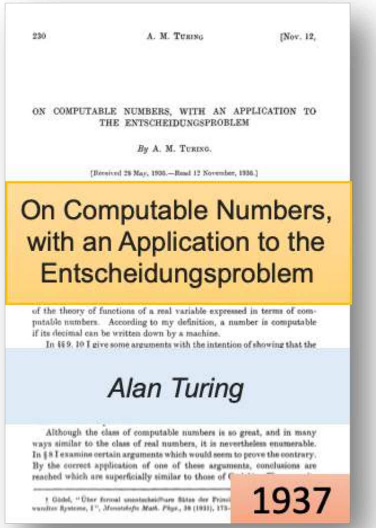


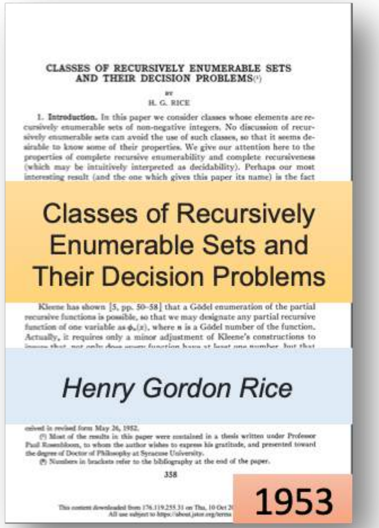


Note #2

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



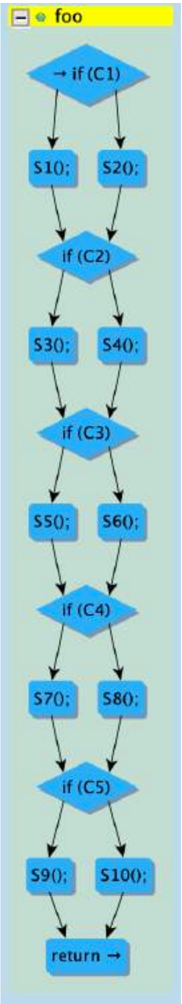




Let’s go over predominant **software analysis challenges**!

Path/State Explosion

```
public void foo() {
    if(C1) {
        S1();
    } else {
        S2();
    }
    if(C2) {
        S3();
    } else {
        S4();
    }
    if(C3) {
        S5();
    } else {
        S6();
    }
    if(C4) {
        S7();
    } else {
        S8();
    }
    if(C5) {
        S9();
    } else {
        S10();
    }
}
```



5 non-nested Branch Points

$2^5 = 32$

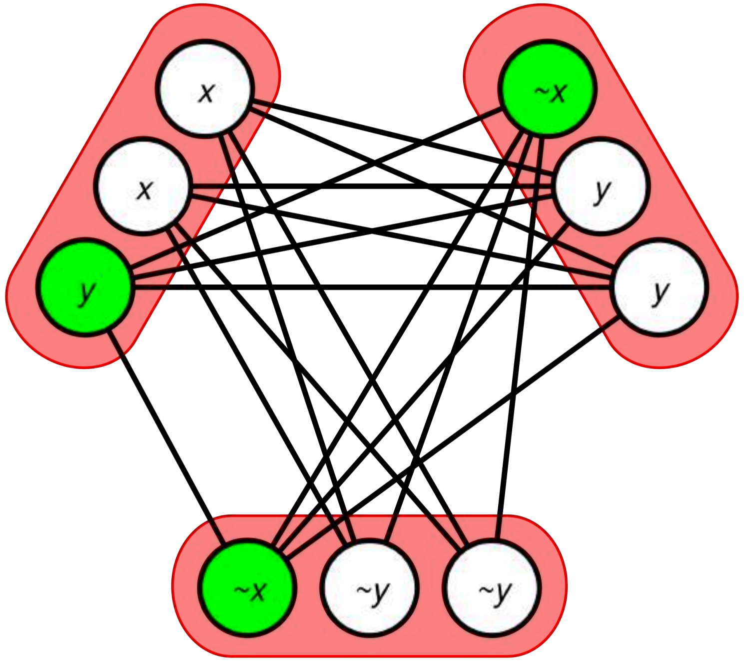
Execution (Paths) Behaviors

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.

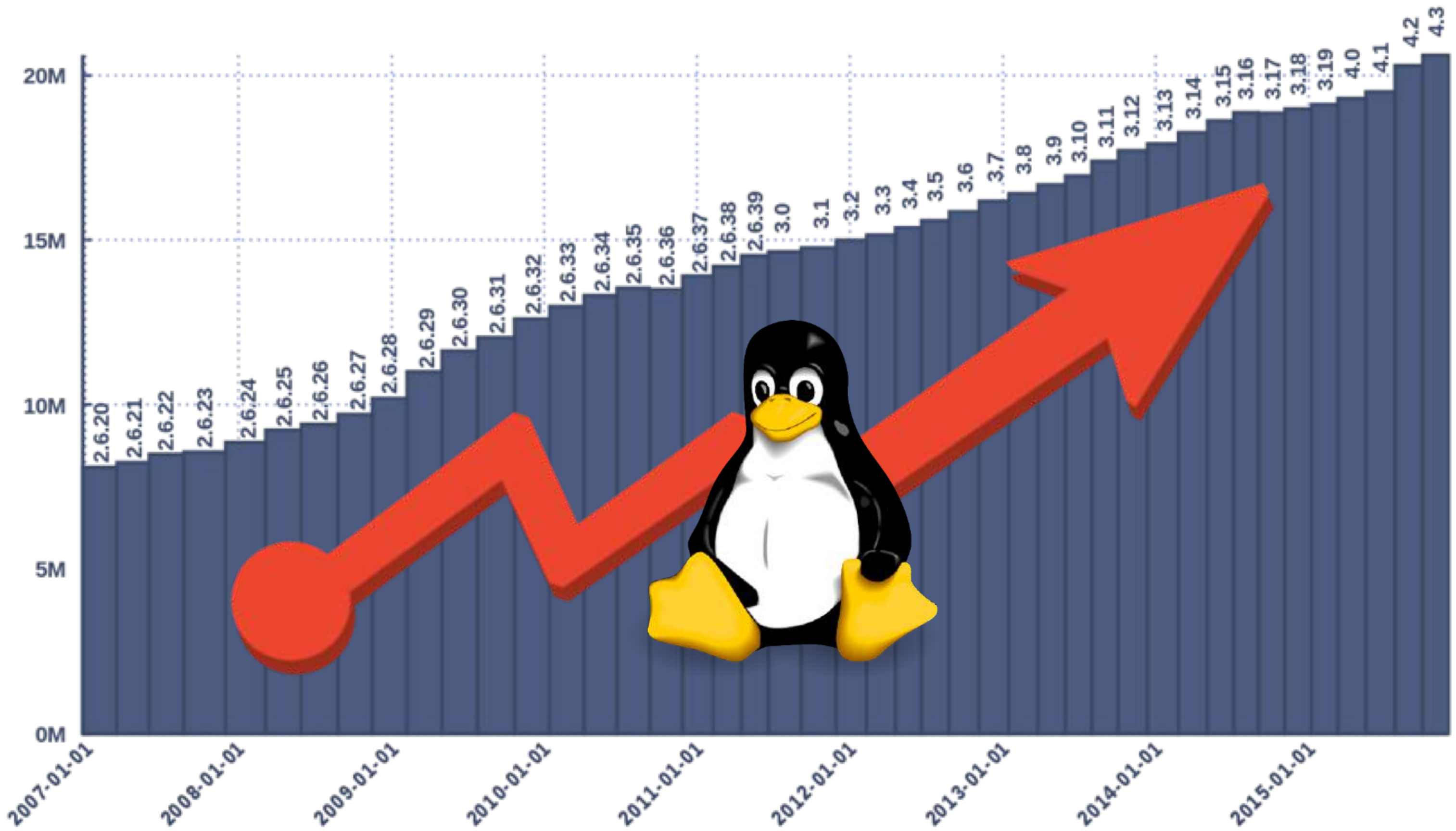
Variability-Aware Analysis

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!

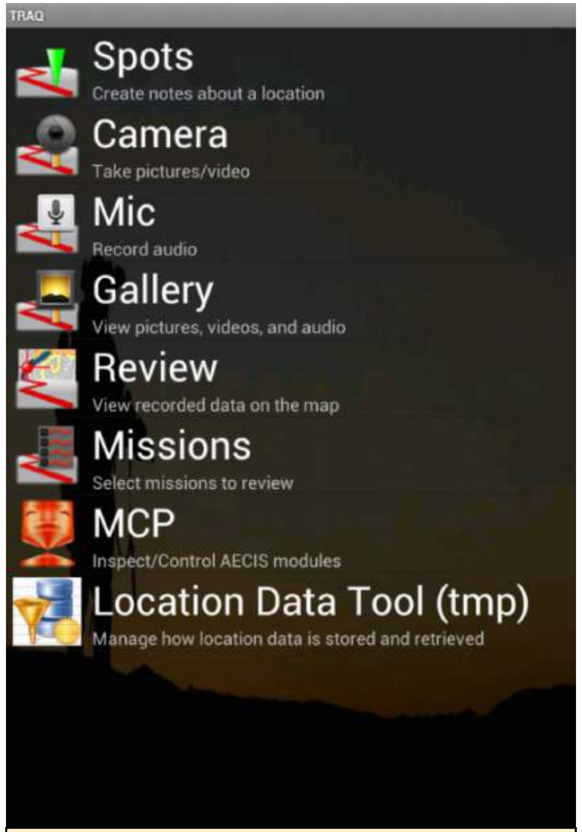
In Reality, Finding Wild and Sophisticated Vulnerabilities is like:

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

Ambiguity: Malice or Legitimate?

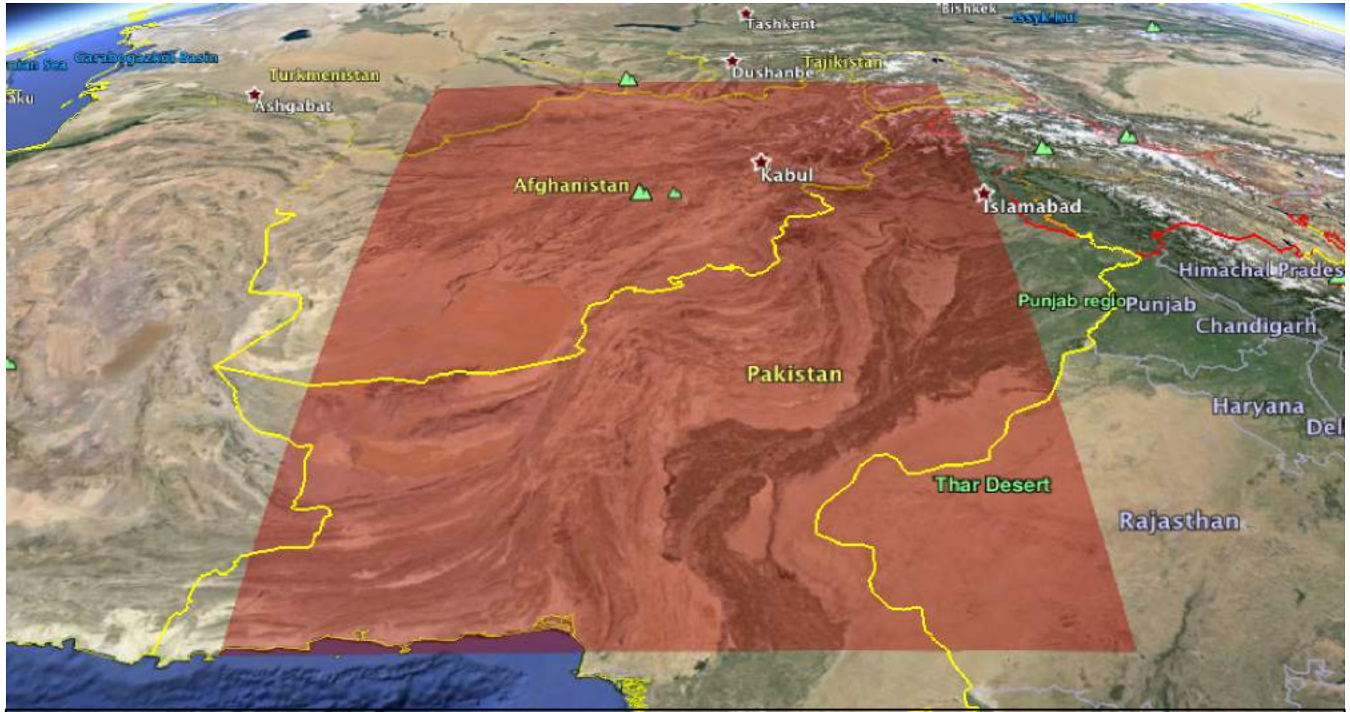
Behavior	App Purpose	Classification
Send location to Internet	Phone locator	Benign
Send location to Internet	Podcast player	Malicious
Selectively block SMS messages	Ad blocker	Benign
Selectively block SMS messages	Navigation	Malicious

There is a need for Domain-Specific Knowledge!



Data Gathering and Relaying App for Military

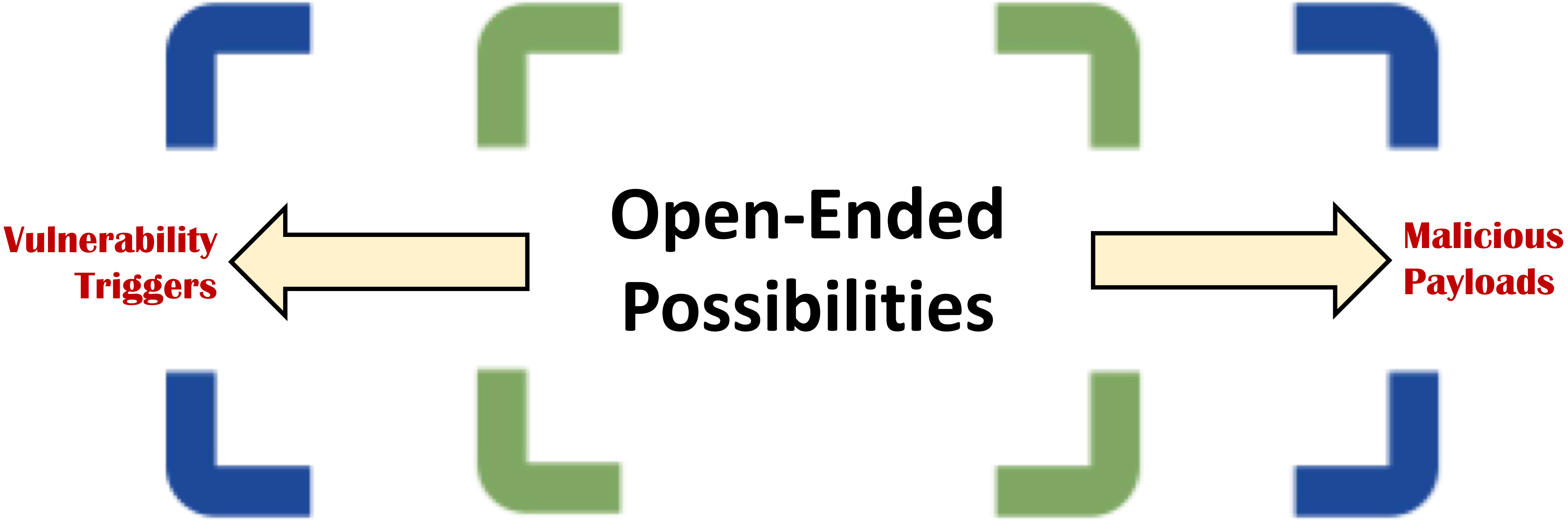
- 55K lines of code
- Strategic mission planning/review
- Audio and video recording
- Geo-tagged camera snapshots
- Real-time map updates based on GPS




```
@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!**

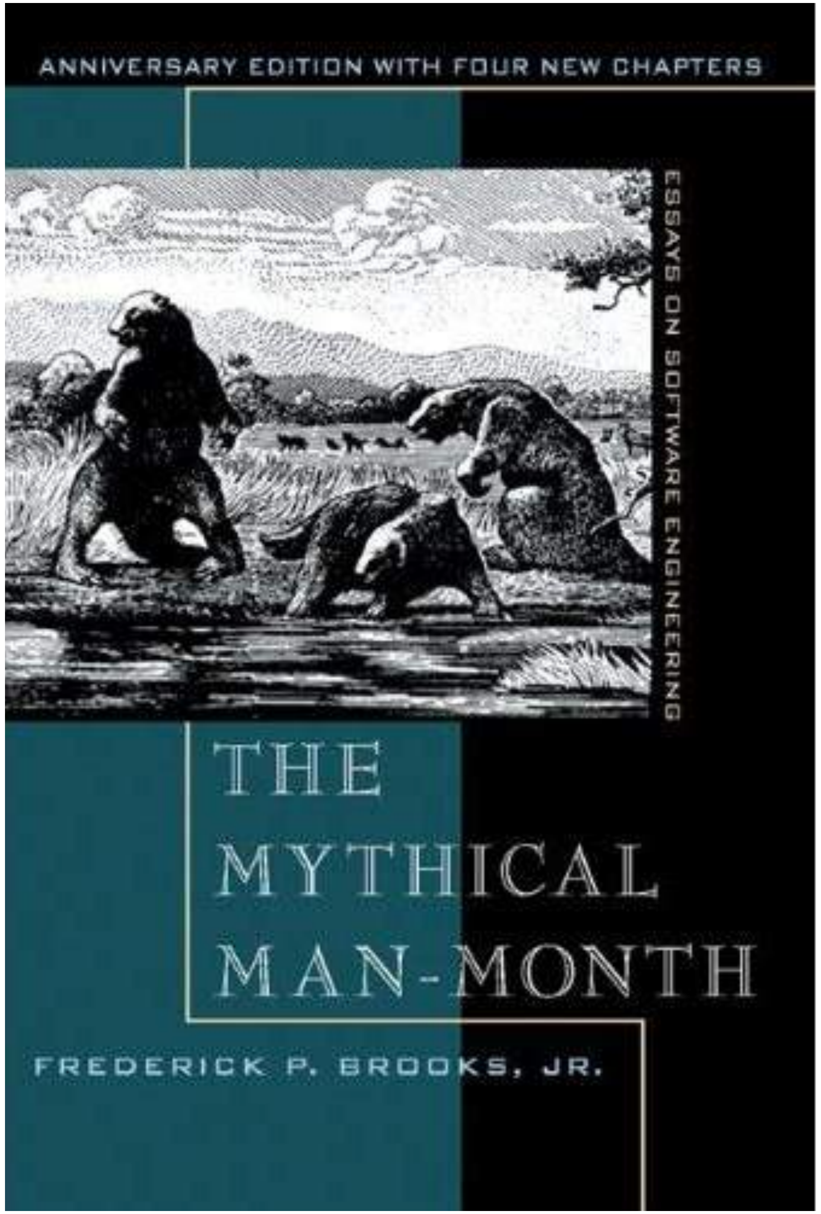
What is different about detecting sophisticated vulnerabilities?



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

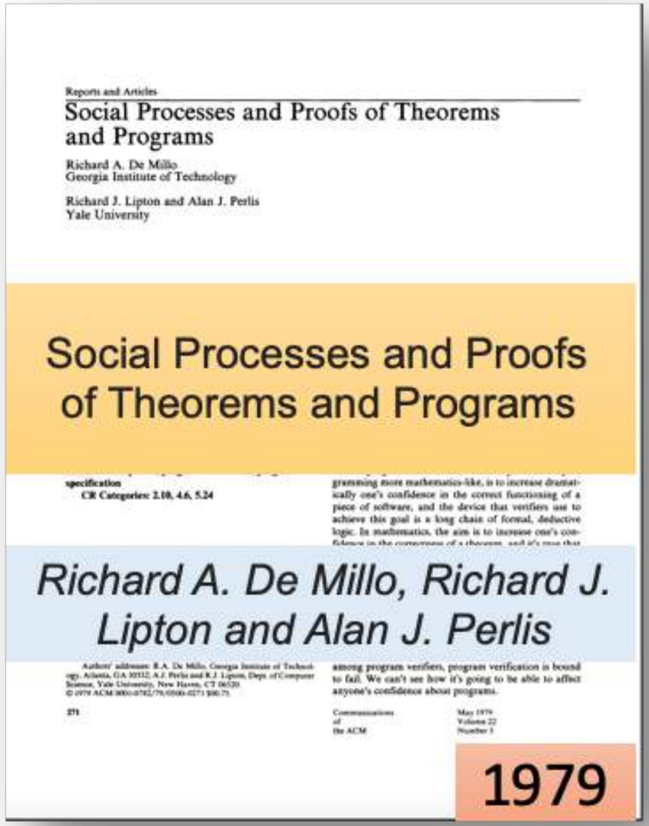
Fully automated software analysis encounters **significant difficulties in practice** – *it either does not complete or yields inaccurate results*!


شو
الحل يا
خال



"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



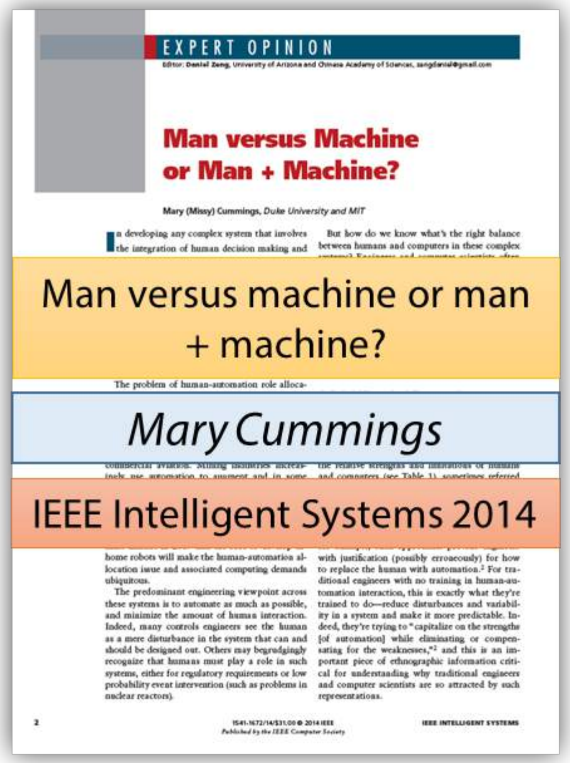
Reports and Articles

Social Processes and Proofs of Theorems and Programs

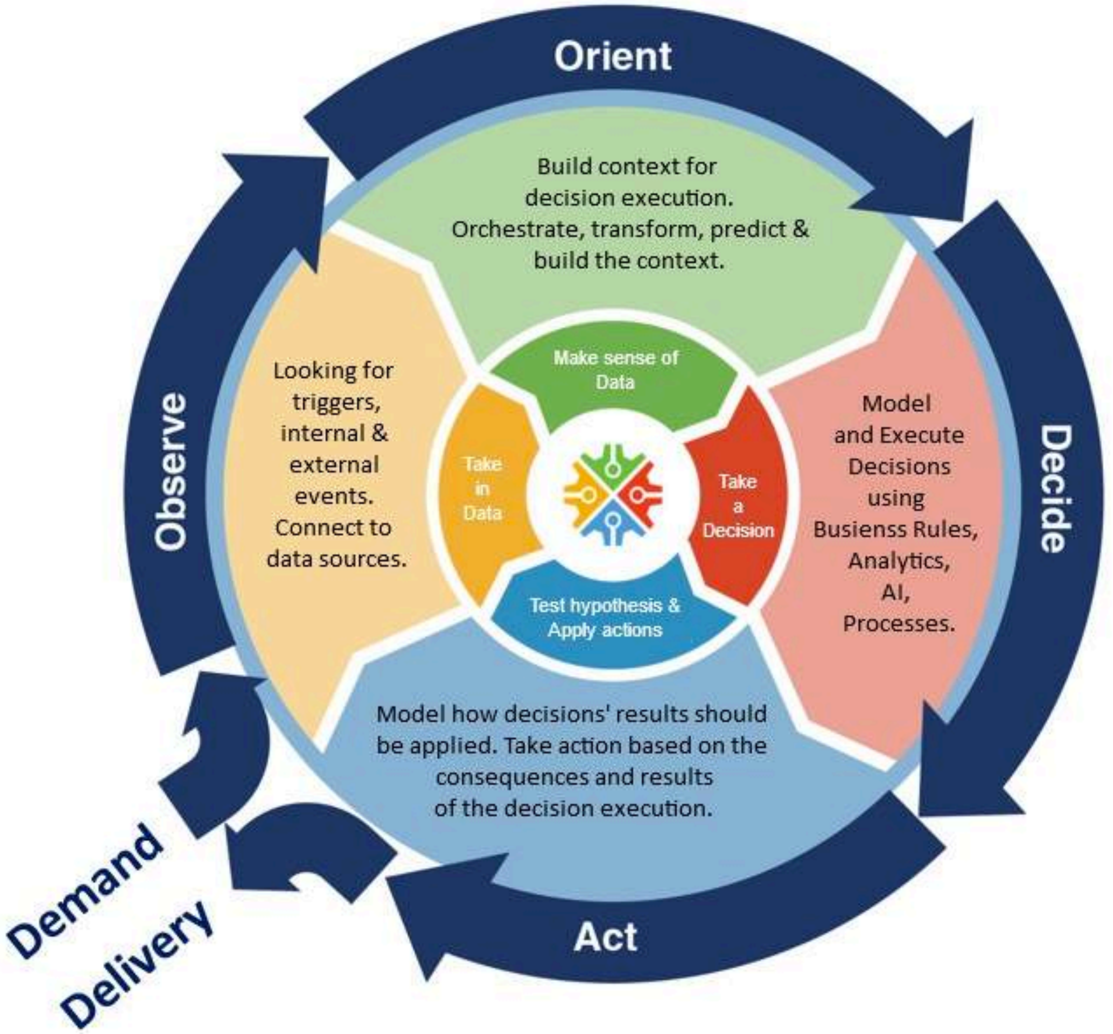
Richard A. De Millo
Georgia Institute of Technology

Richard J. Lipton and Alan J. Perlis
Yale University

***"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."*



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!

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

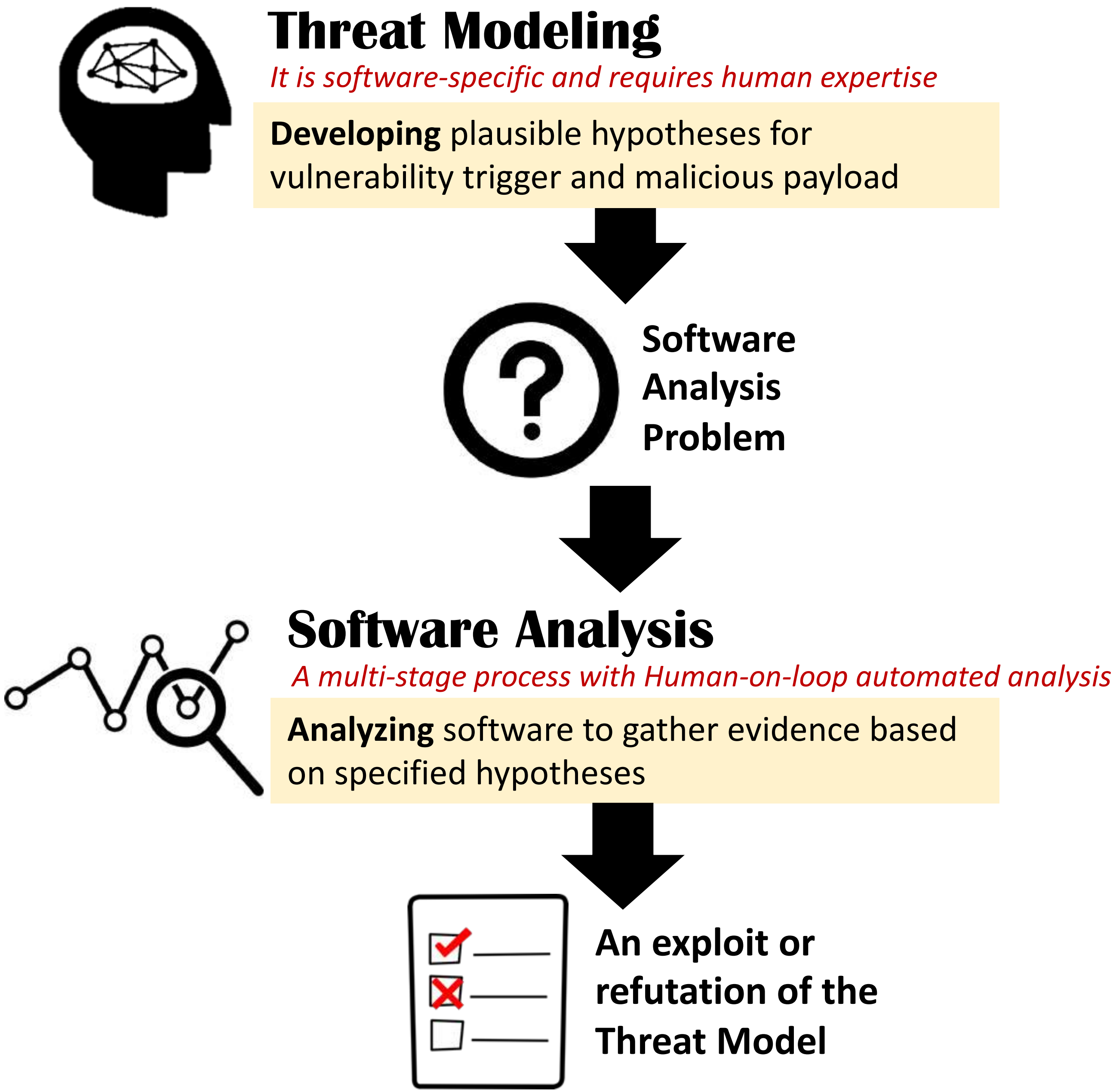
$$e^{i\pi} + 1 = 0$$

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

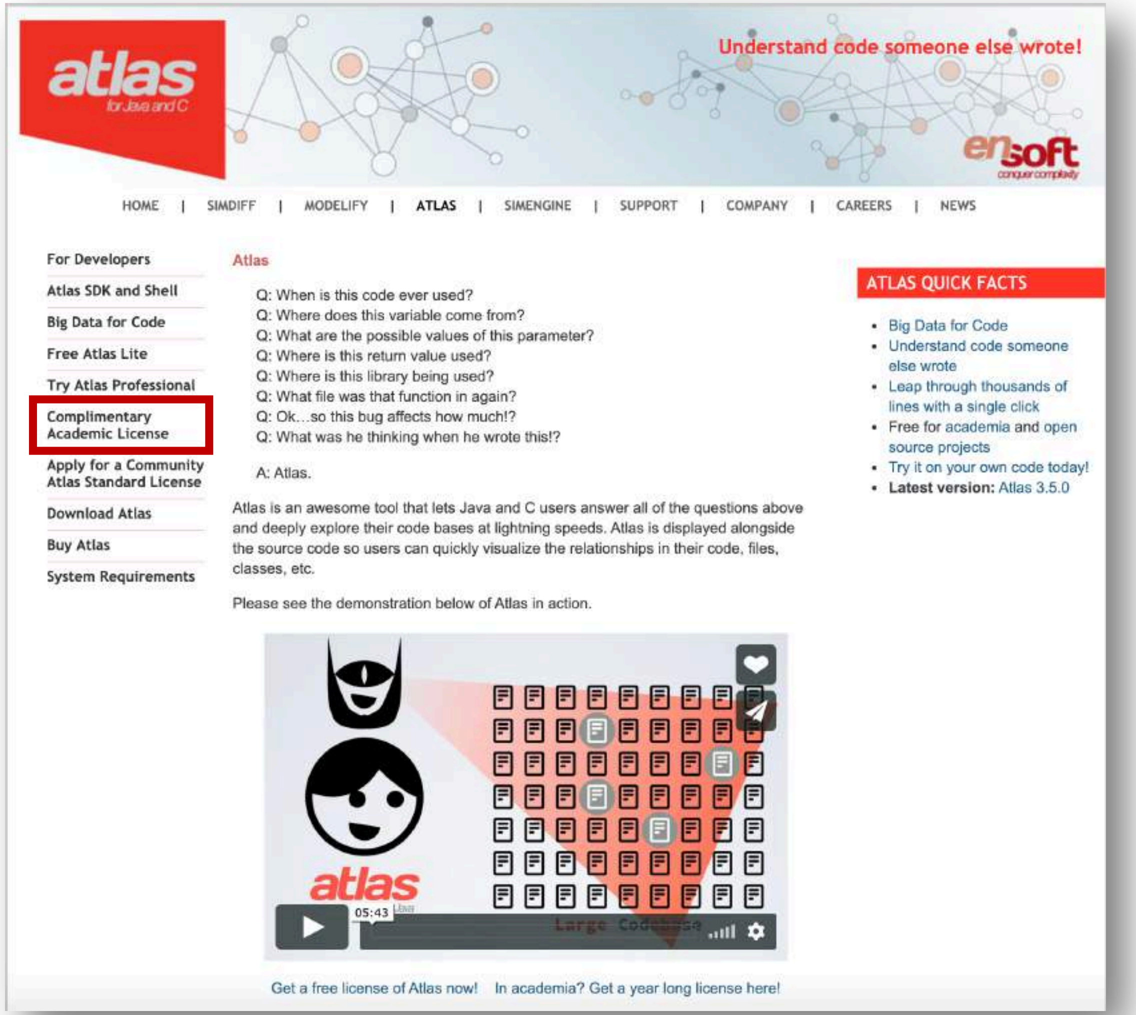
New technological advances are crucial for using the Euler's method with software of gigantic proportions!


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

Human-In-The-Loop approach to *Detecting Sophisticated Vulnerabilities*



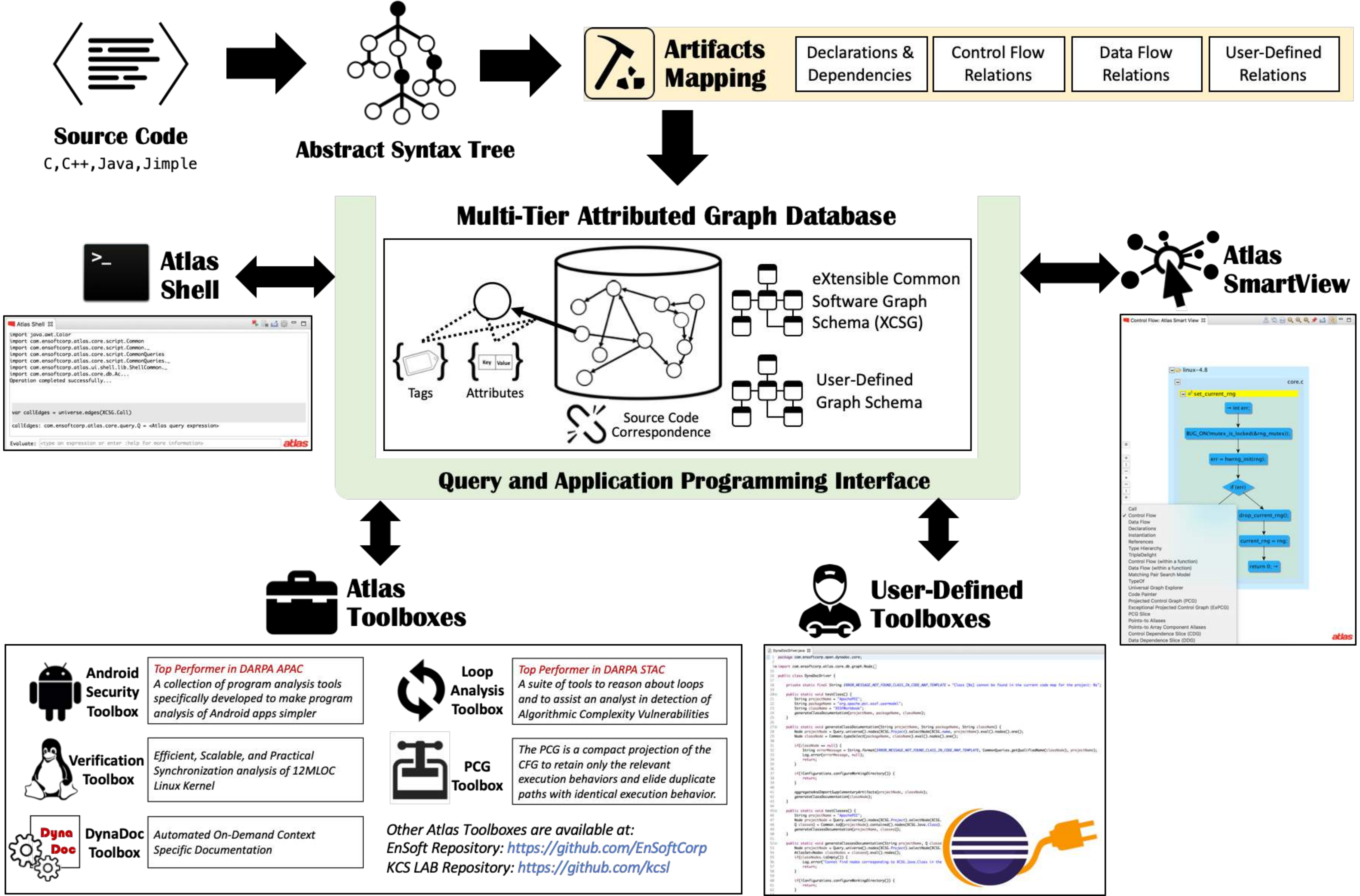
Our goal is to build an **intelligence amplifying framework** that **mines and connects various software artifacts** and **enables human-machine interaction** to solve complex software problems



http://www.ensoftcorp.com/atlas/

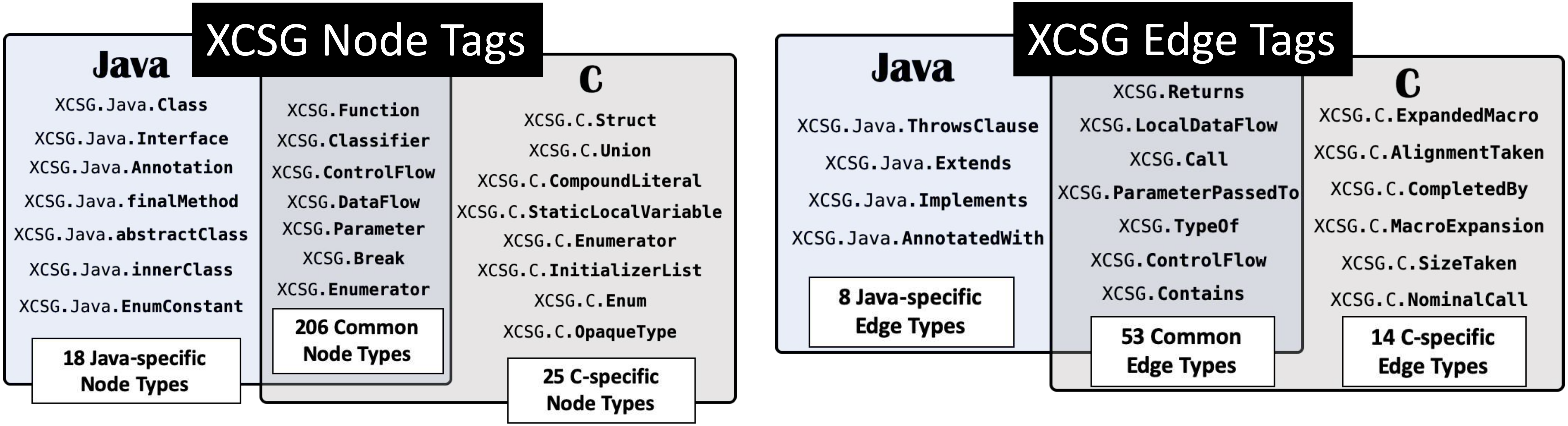
Atlas is an intelligence amplifying framework that **provides a new way to interactively explore software artifacts** and **enables analysts to write analysis scripts to tackle complex software problems**!

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


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
Tags
Key
Value
Attributes
Source Code Correspondence
eXtensible Common Software Graph Schema (XCSG)
User-Defined Graph Schema
Query and Application Programming Interface
Atlas Shell
Atlas SmartView
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.
Dyna Doc
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/kcsl

e**X**tensible **C**ommon **S**oftware **G**raph (XCSG)

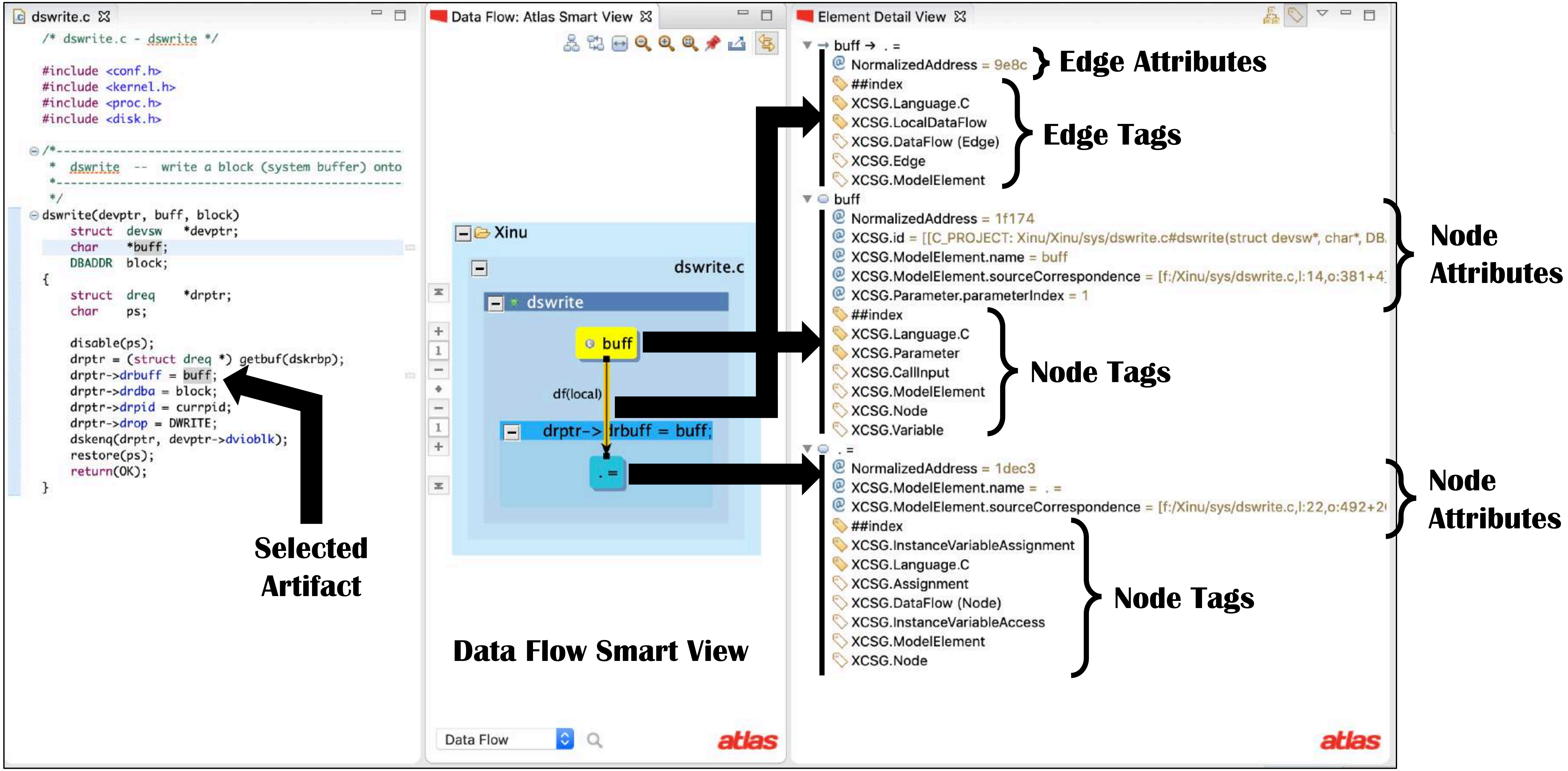
a harmonious representation of software written in different languages



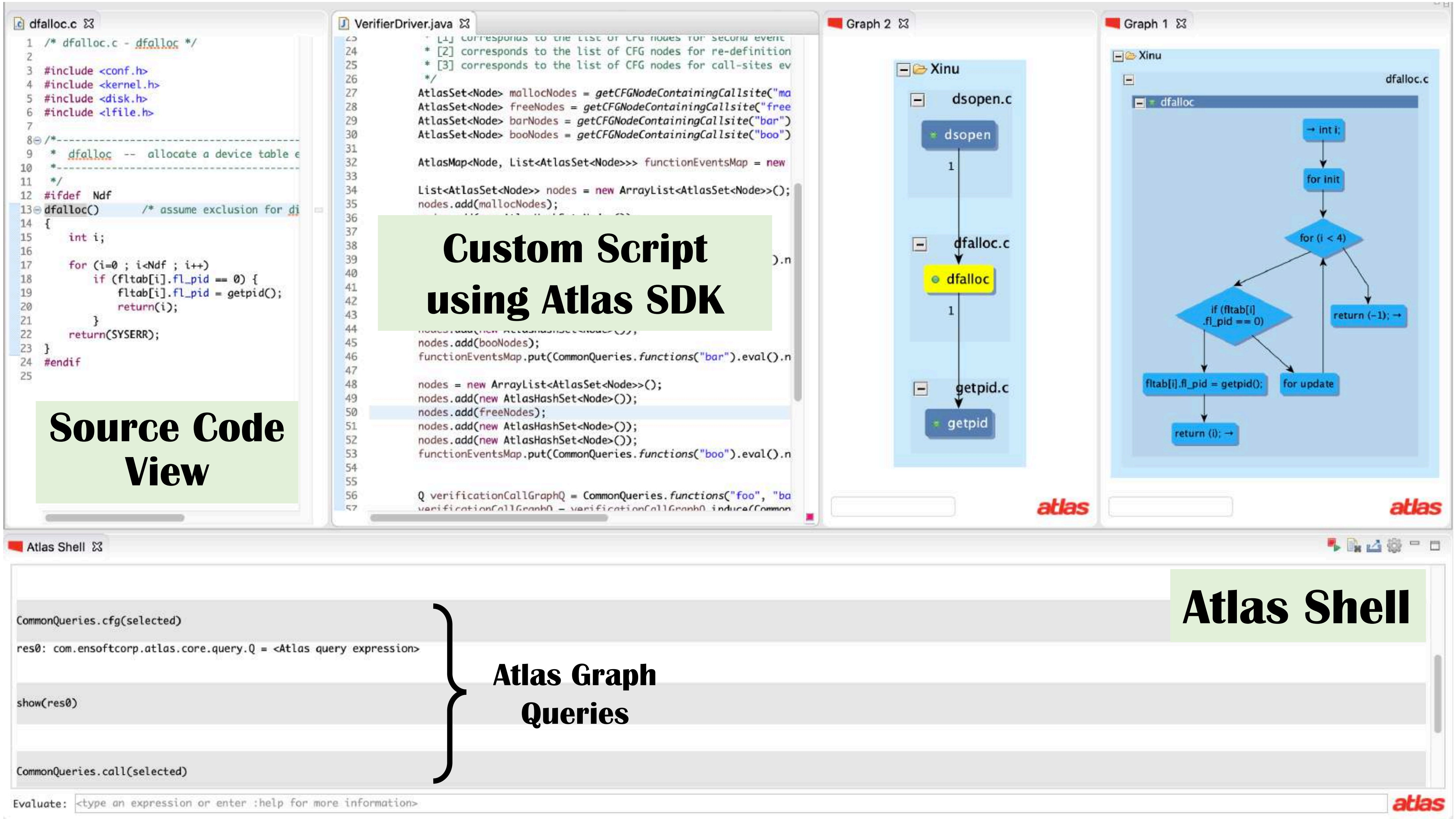
XCSG defines a variety of **program artifacts (nodes)** and **relations (edges)** to *capture the semantics of programming languages*

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

Atlas Smart View and Atlas Element Detail View

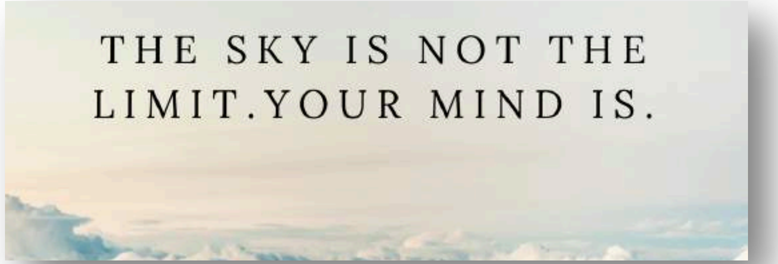


Atlas Shell, Custom Scripts, and Atlas SDK



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!*



Atlas Query Language Examples

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

```java
// We first find the function that we want to reason about.
Q fooFunction = CommonQueries.functions("foo");

// Let us find all the loops in function "foo".
Q fooCFG = CommonQueries.cfg(fooFunction);

// display the control flow graph of function "foo".
DisplayUtil.displayGraph(fooCFG.eval(), null, "foo CFG");

// print out the number of loops
Q fooLoops = fooCFG.nodes(XCSG.Loop);
System.out.println(fooLoops.eval().nodes().size());

// Let us find all the functions that directly call "foo".
Q callEdges = Query.universe().edges(XCSG.Call);
Q fooCallers = fooFunction.reverseStepOn(callEdges);

// Let us find all the functions that have call chains to "foo".
Q fooAncestors = fooFunction.reverseOn(callEdges);
```

Custom script written in Java using Atlas SDK

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?

https://kcsl.github.io/L-SAP

Scalable, *Efficient*, and *Practical* **Linux Verification** against Synchronization Problems

https://kcsl.github.io/L-SAP

Verification Results against Top Performing Tool

4 total hours to complete — **Explainable Verification (EV)** tool based on Atlas platform

173 total hours to complete — **Linux Driver Verification (LDV)** tool the top performers in the SV-COMP '14, 15' and '16

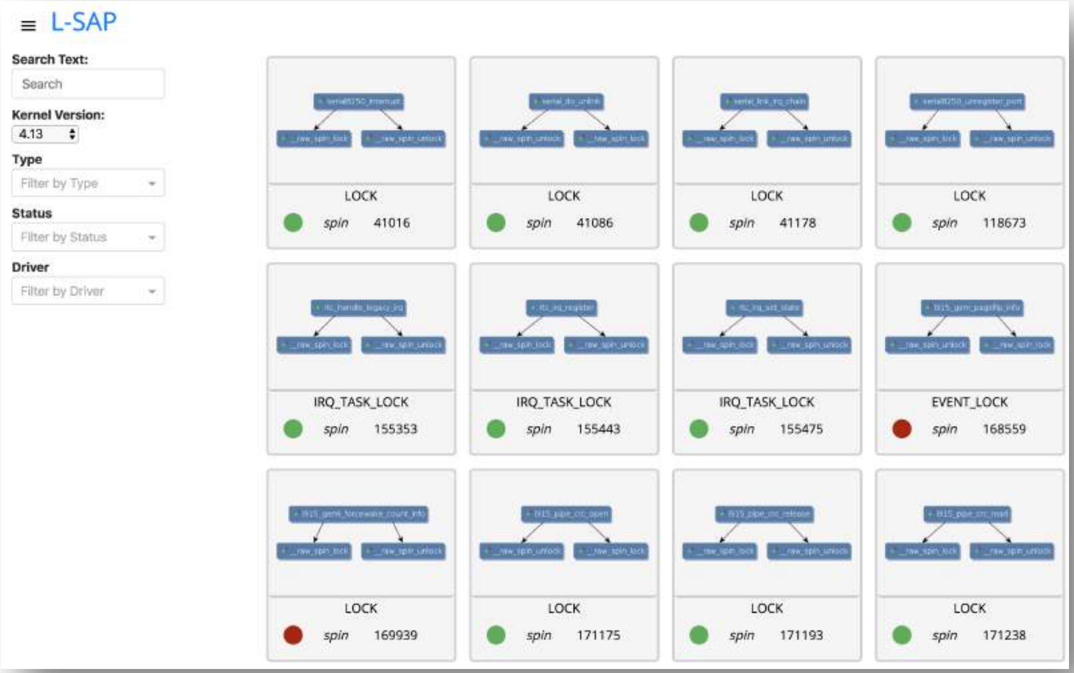
Kernel	Lock Type	Lock Instances	EV: Safe	EV: Unsafe	EV: Incomplete Verification	LDV: Safe	LDV: Unsafe	LDV: Potential Bug	LDV: Incomplete Verification		
									Crash	Timeout	Total
3.17-rc1	Mutex	7887	7813	1	73	5494	0	91	1200	1102	2302
	Spin	14180	14097	1	82	8962	0	366	2188	2664	4852
3.18-rc1	Mutex	7893	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%**

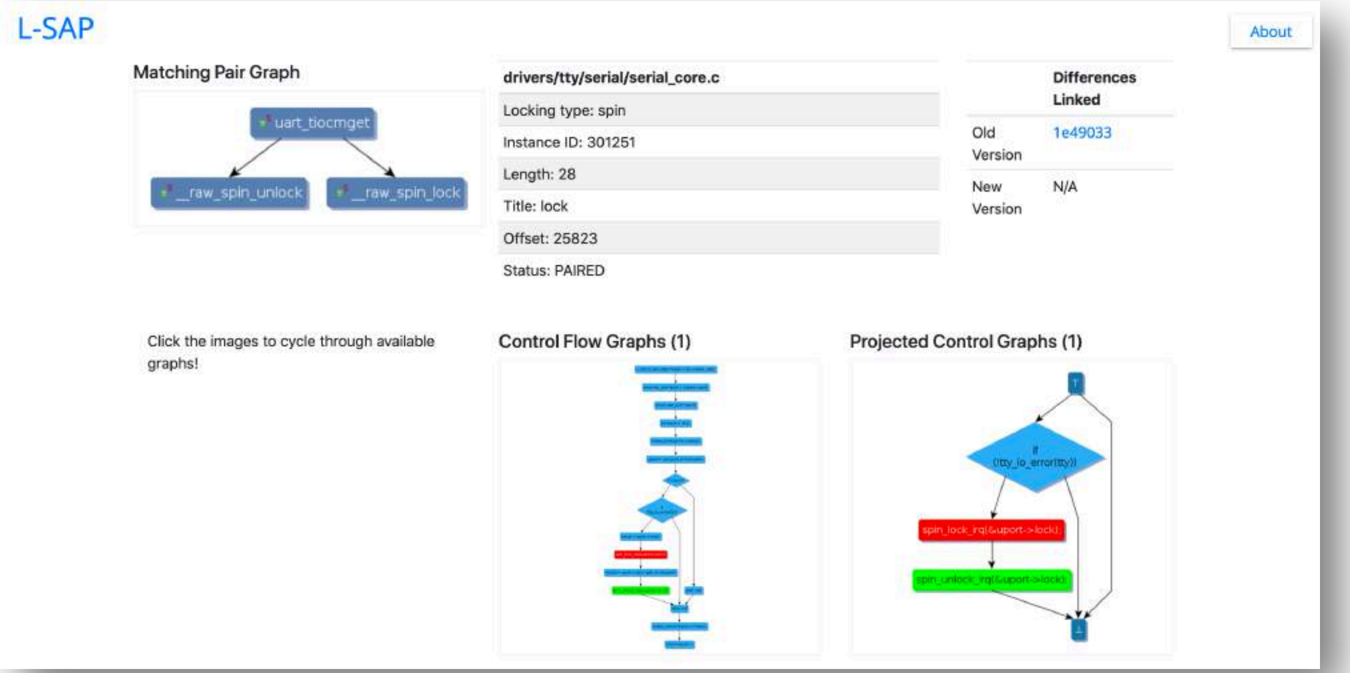
Importing Linux Kernel in Atlas

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





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***

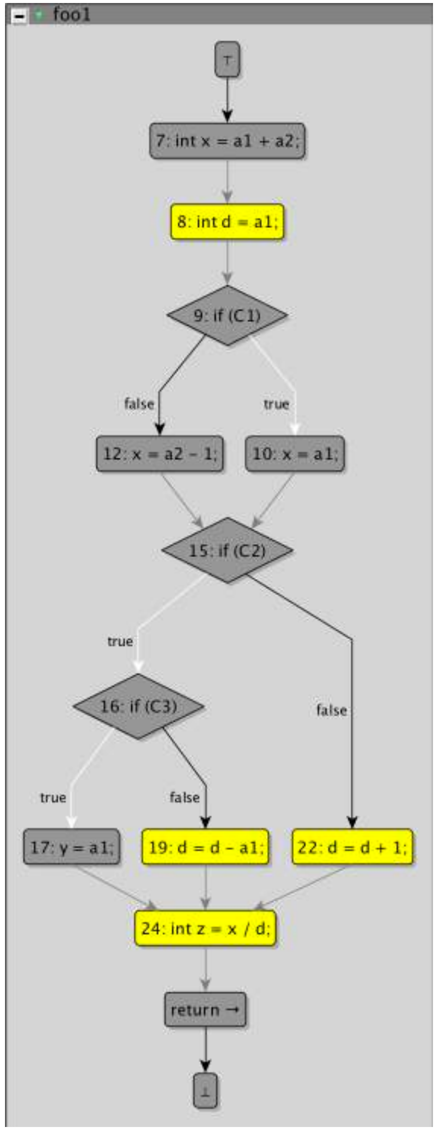
Algorithmic Challenge*:* Compute the distinct relevant behaviors without going through each path!

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)

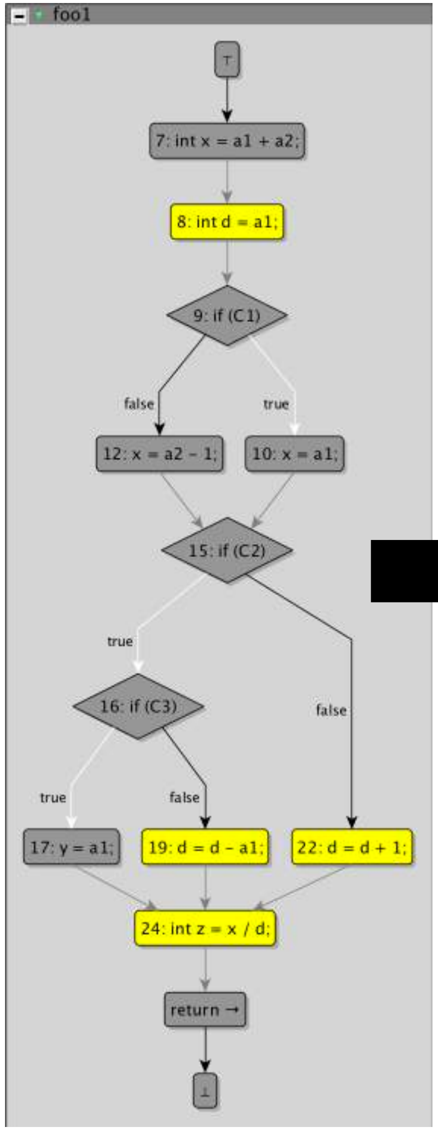
Six Possible Execution Paths

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

Division-By-Zero (DBZ) Vulnerability?

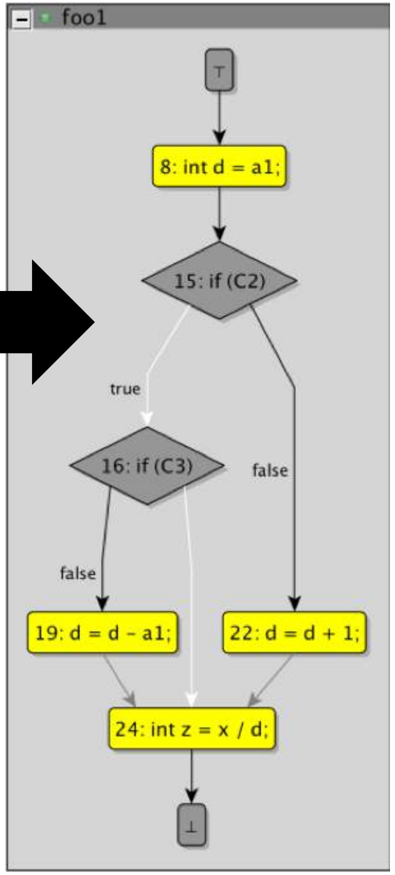
```
1  int a1 = 1, a2 = 2;
2  int y = 2;
3  bool C1 = true;
4  bool C2 = false;
5  bool C3 = true;
6  void foo1() {
7      int x = a1 + a2;
8      int d = a1;
9      if(C1){
10         x = a1;
11     }else{
12         x = a2 - 1;
13     }
14
15     if(C2){
16         if(C3){
17             y = a1;
18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

Function `foo1`



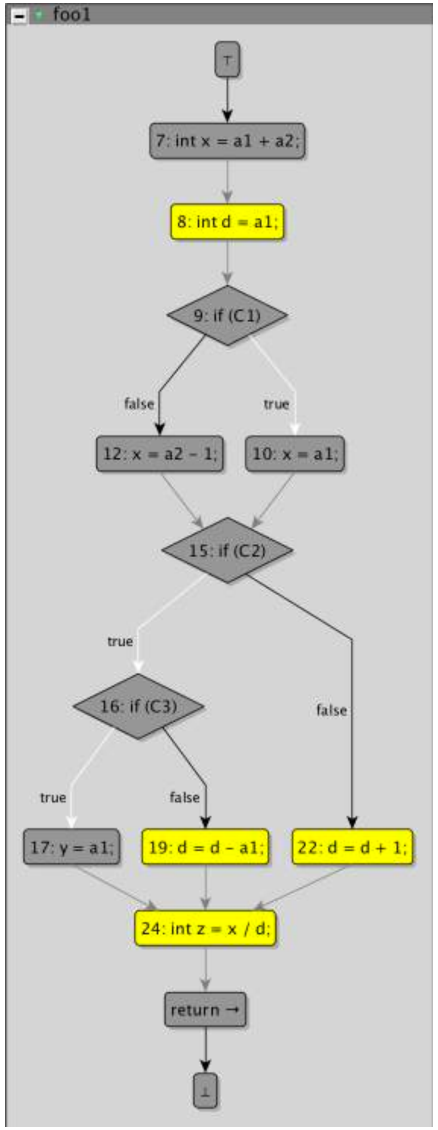
Control Flow Graph

Efficient Graph Transformations based on famous Tarjan's Algorithm



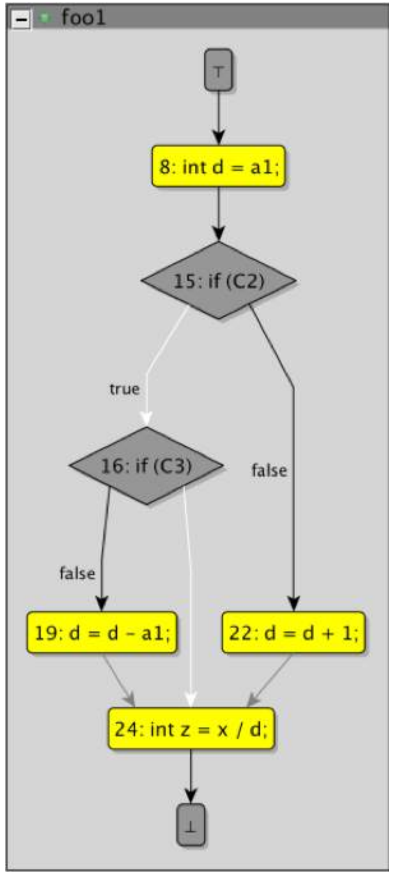
Projected Control Graph

Division-By-Zero (DBZ) Vulnerability?



Control Flow Graph

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



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
ptlrpc_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_writepages_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

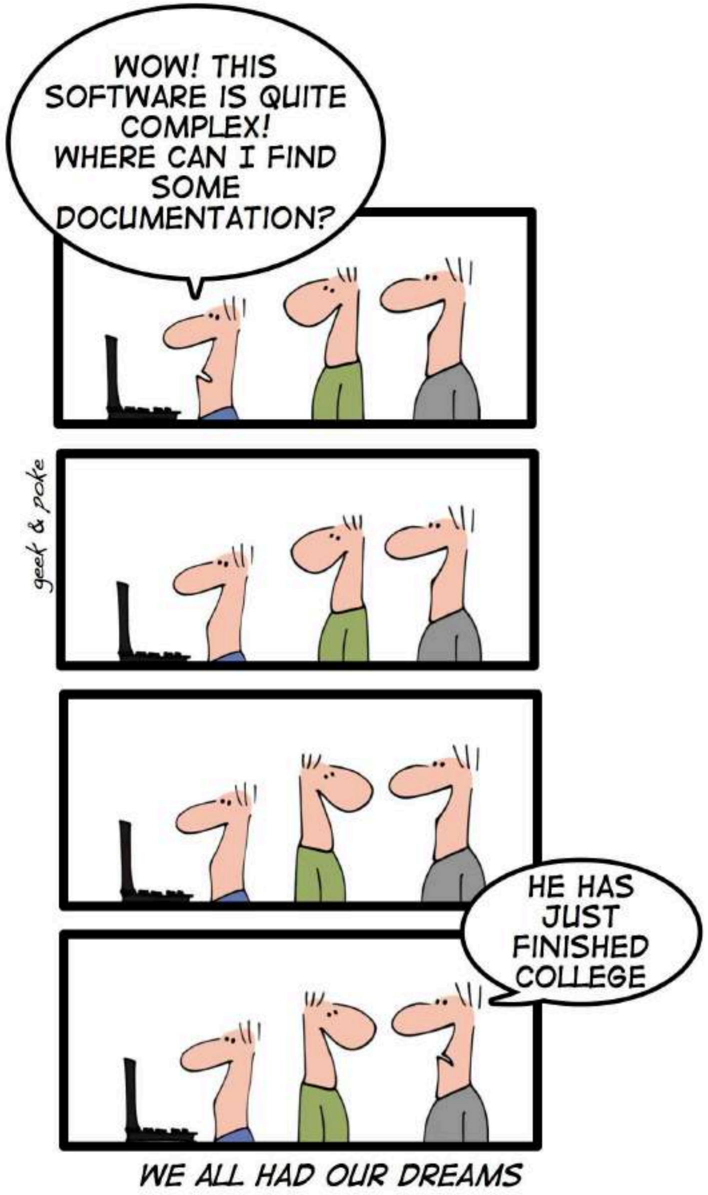
Control Flow Graph (CFG) captures the entire semantics!

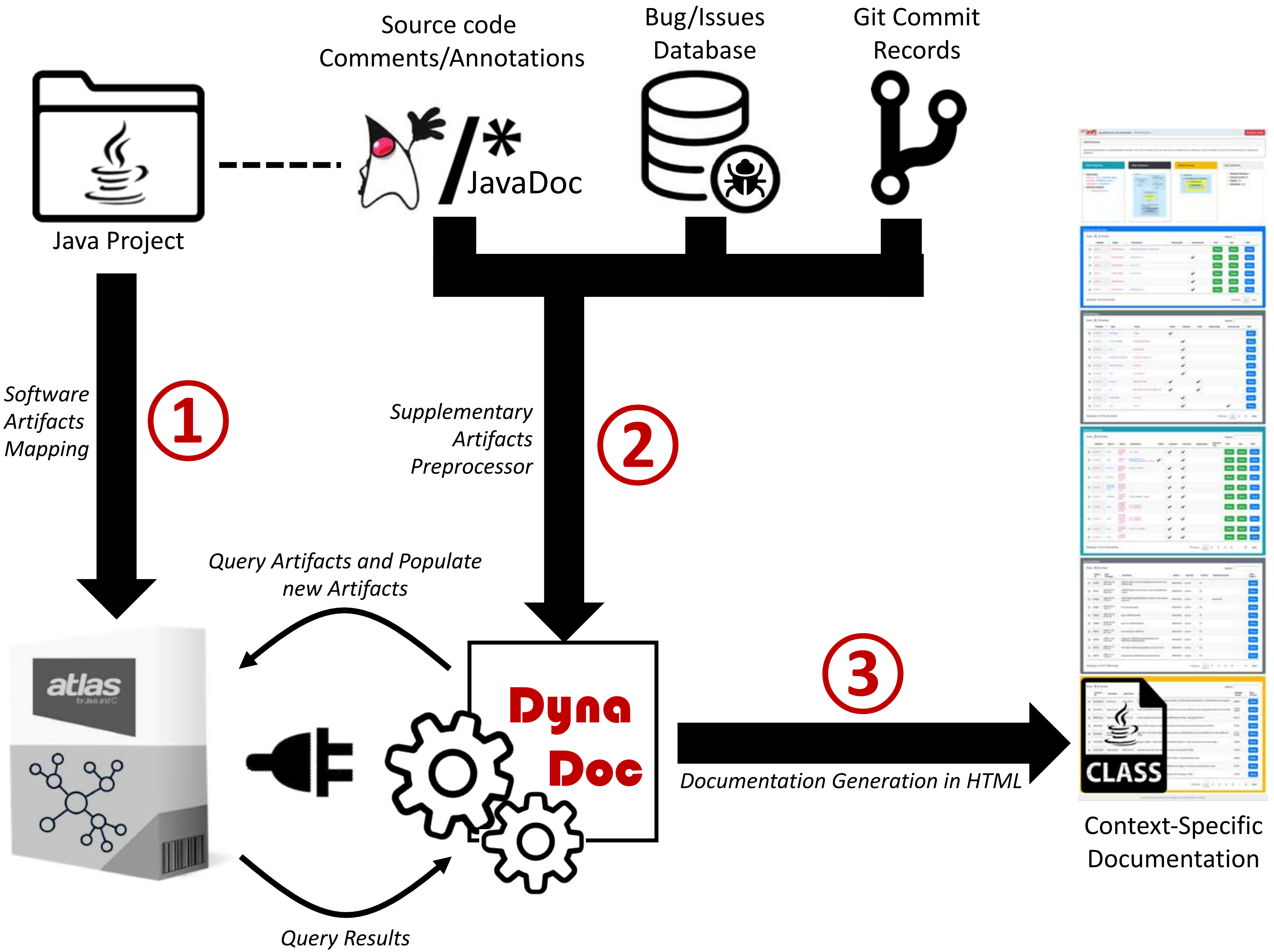
Exponentially many paths but **only a small number of relevant behaviors!**

https://github.com/EnSoftCorp/DynaDoc

Dyna
Doc

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





Source code
Comments/Annotations
Bug/Issues
Database
Git Commit
Records
Java Project
JavaDoc
Software
Artifacts
Mapping
1
Supplementary
Artifacts
Preprocessor
2
Query Artifacts and Populate
new Artifacts
atlas
Dyna
Doc
3
Documentation Generation in HTML
CLASS
Context-Specific
Documentation
Query Results

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

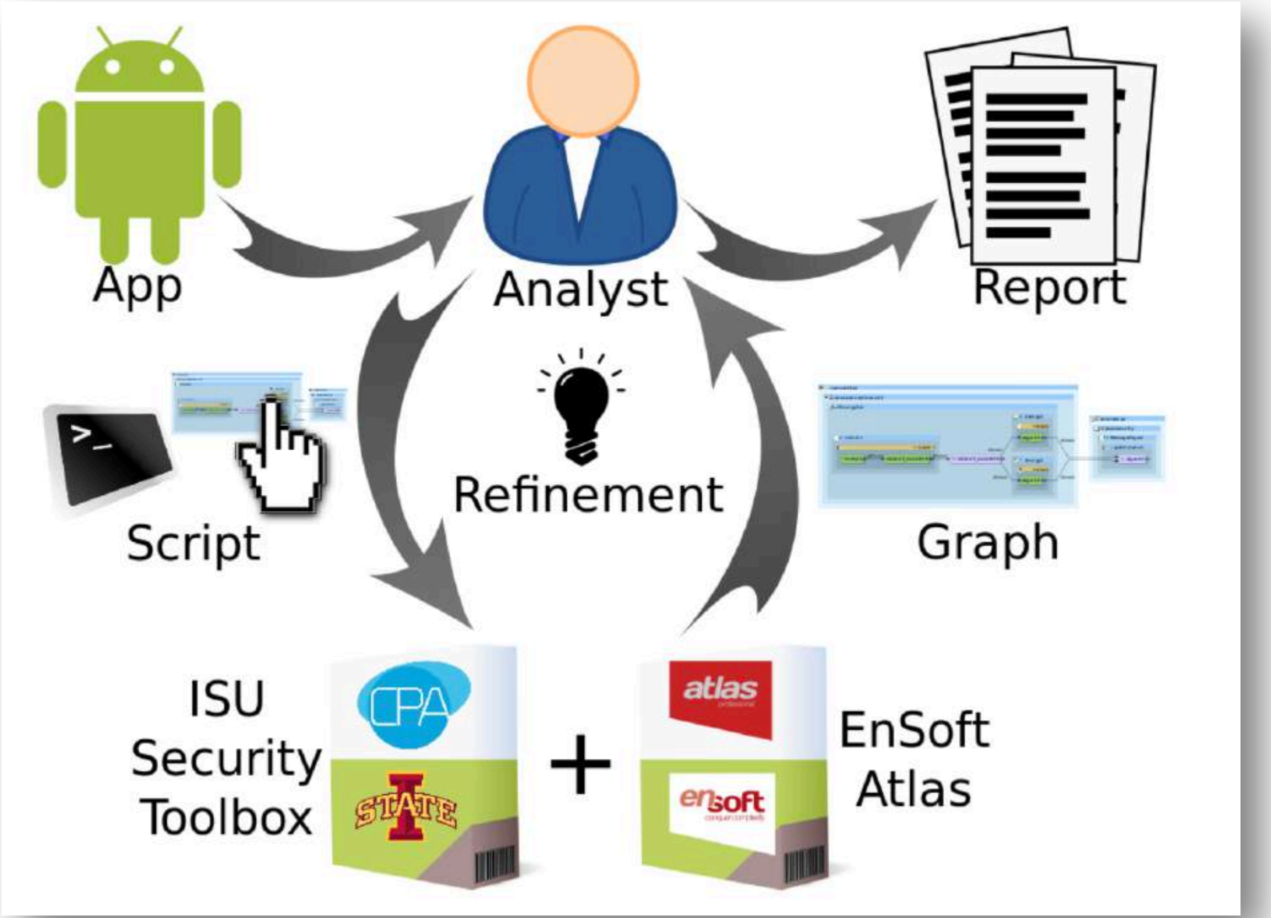
Automated Program Analysis for Cybersecurity

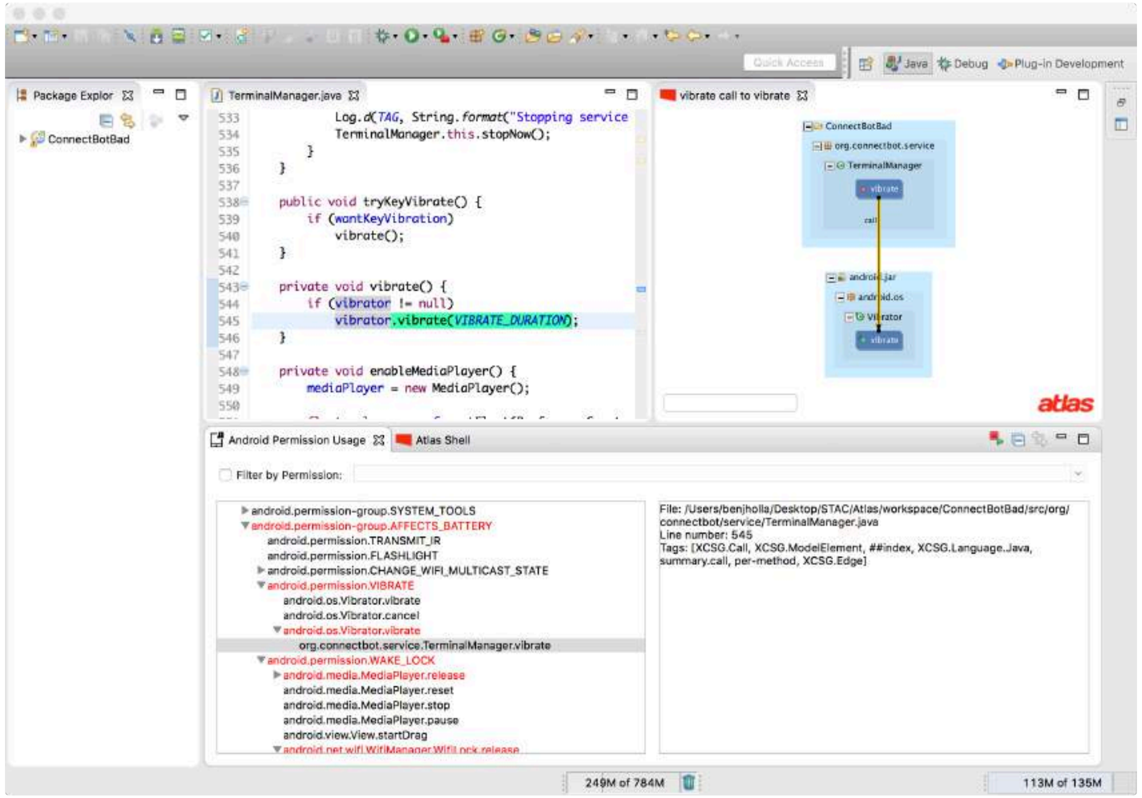
The program aims to address the challenge of **timely and robust security validation of mobile apps** by first defining security properties to be measured against and then developing automated tools to perform the measuring. The second challenge APAC aims to address is **producing practical, automated 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.*

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

DARPA APAC Program

Automated Program Analysis for Cybersecurity





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

Android Security Toolbox
https://ensoftcorp.github.io/android-essentials-toolbox/

DARPA STAC Program

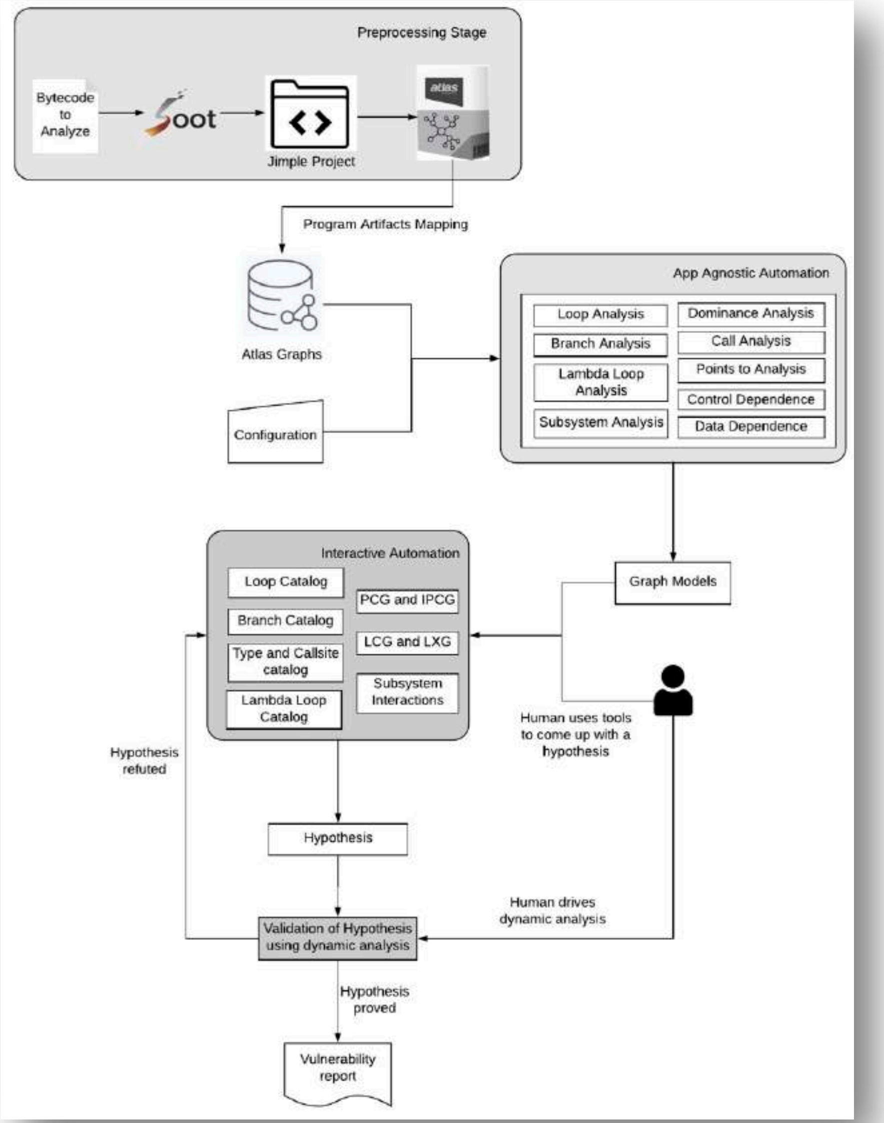
Space/Time Analysis for Cybersecurity

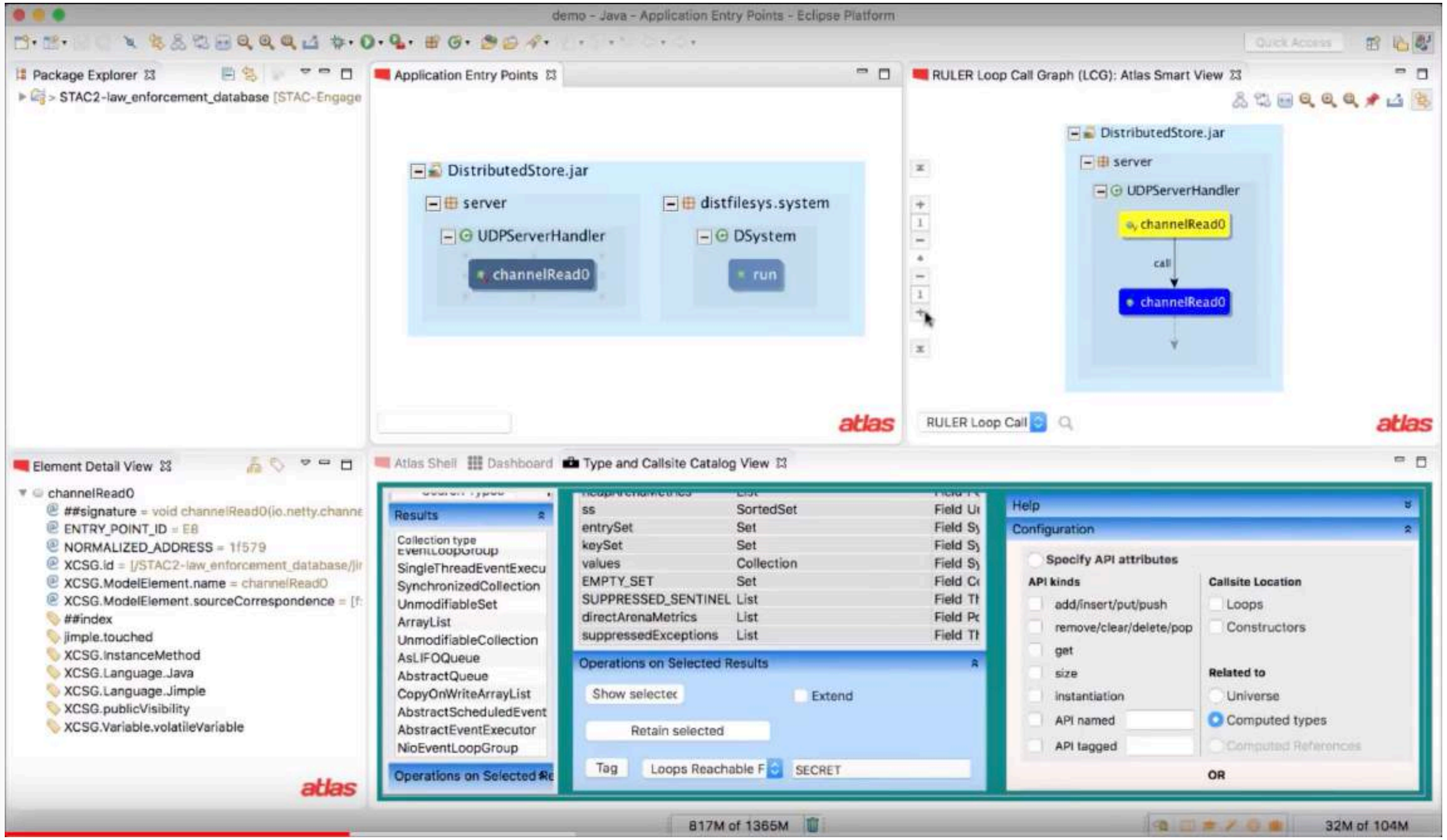
The program aims to **develop new program analysis techniques and tools for identifying vulnerabilities related to the space and time resource usage behavior of algorithms**, specifically, vulnerabilities to *algorithmic complexity and side channel attacks*. STAC seeks to **enable analysts to identify 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.

www. 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

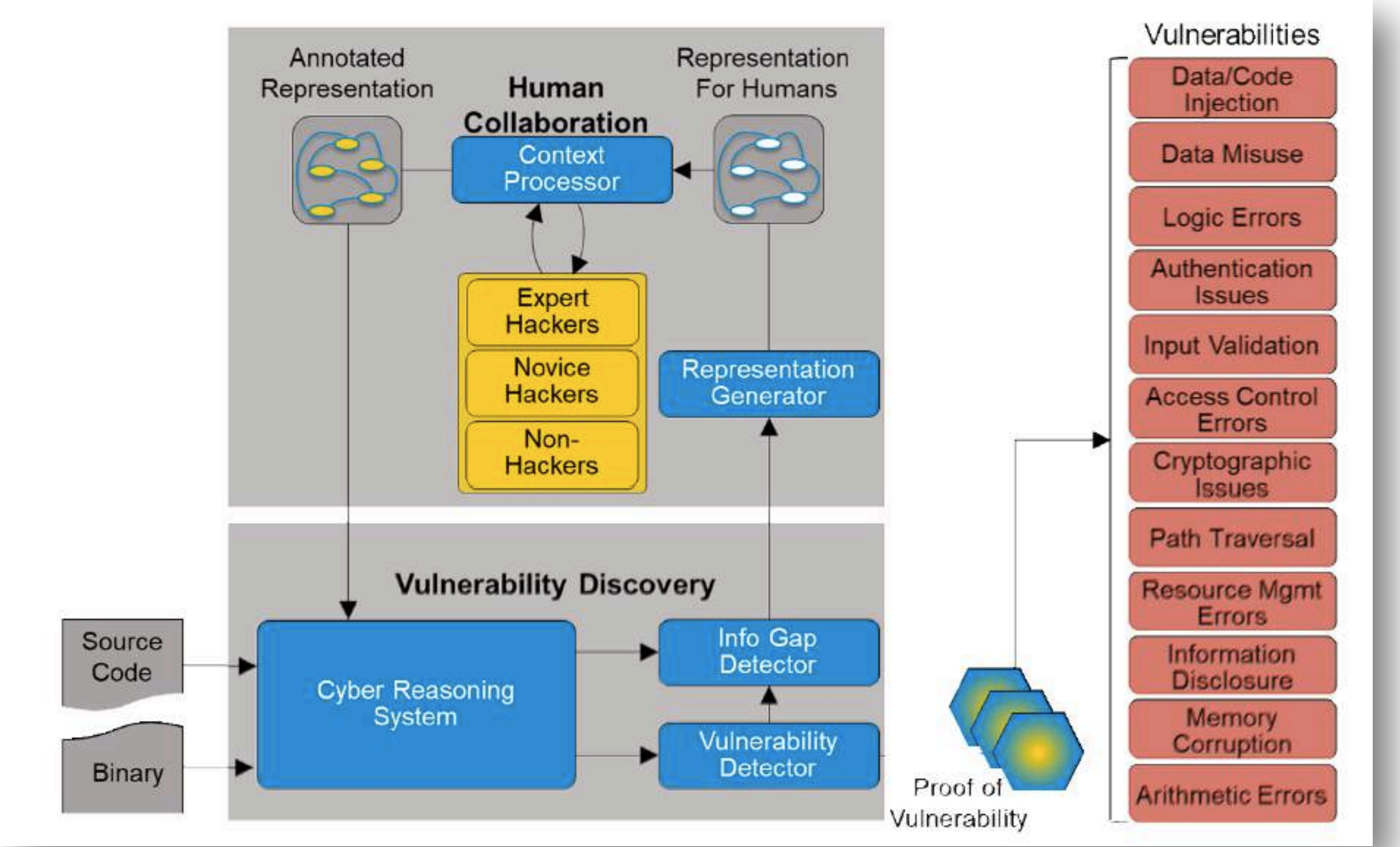
Computers and Humans Exploring Software Security

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.

www. 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

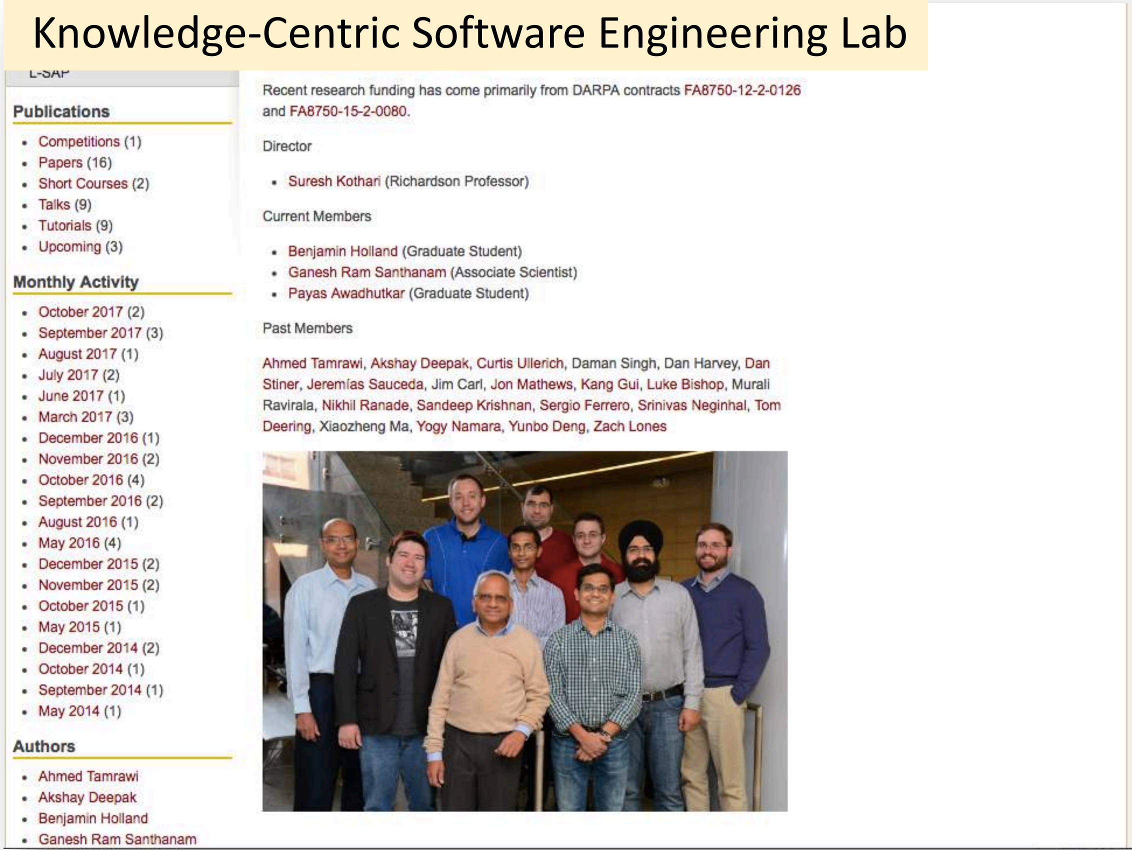
and many others ...

For further information and Resources:

atamrawi atamrawi.github.io ahmedtamrawi@gmail.com

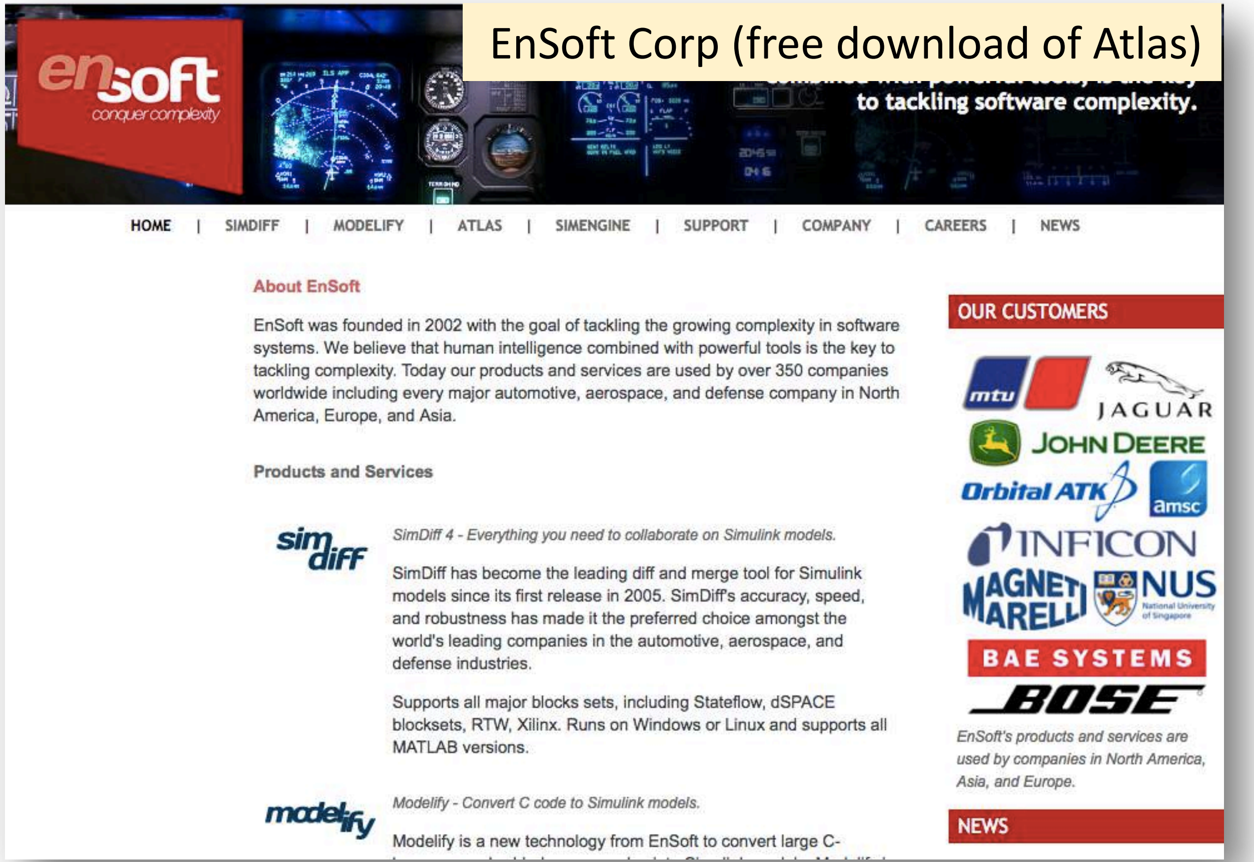
http://www.ece.iastate.edu/kcsl/

Knowledge-Centric Software Engineering Lab



https://www.ensoftcorp.com/

EnSoft Corp (free download of Atlas)



Thank you