MILCOM 2017

MILITARY COMMUNICATIONS AND INNOVATION - PRIORITIES FOR THE MODERN WARFIGHT

Learn to analyze and verify large software for cybersecurity and safety

Suresh C. Kothari

Richardson Professor, Department of Electrical and Computer Engineering

President & Founder EnSoft Corp.

Ben Holland, a graduate student and a security analyst on the DARPA APAC and STAC projects

Materials: https://github.com/benjholla/MILCOM2017

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

BALTIMORE, MD • OCTOBER 23–25, 2017

Cybersecurity attacks/disasters

- WannaCry – May 12, 2017
- Ukraine power grid attacks – December 23, 2015 and December 17, 2016
- Jeep remotely hijacked – July 21, 2015
- HeartBleed – April 1, 2014
- HP printers remotely set on fire – November 29, 2011
- Stuxnet – identified in 2010, deployed in 2005?
- Ariane 5 rocket launch explosion – June 4, 1996

What do these attacks/disasters have in common? All rooted in Software

Atiane 5 disaster: https://www.youtube.com/watch?v=PK_yguLapgA

Blurred distinction: Bug or malware?

```
-
-                if ((err = ReadyHash(&SSLHashMD5, &hashCtx, ctx)) != 0)
600 +
601 +                if ((err = ReadyHash(&SSLHashMD5, &hashCtx)) != 0)
602                          goto fail;
603                  if ((err = SSLHashMD5.update(&hashCtx, &clientRandom)) != 0)
604                          goto fail;
... @@ -616,10 +617,10 @@ OSStatus FindSigAlg(SSLContext *ctx,
617
618          hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
619      hashOut.length = SSL_SHA1_DIGEST_LEN;
-     if ((err = SSLFreeBuffer(&hashCtx, ctx)) != 0)
620 +     if ((err = SSLFreeBuffer(&hashCtx)) != 0)
621          goto fail;
622
-     if ((err = ReadyHash(&SSLHashSHA1, &hashCtx, ctx)) != 0)
623 +     if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
624          goto fail;
625      if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
626          goto fail;
... @@ -627,6 +628,7 @@ OSStatus FindSigAlg(SSLContext *ctx,
628          goto fail;
629      if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
630          goto fail;
631 +        goto fail;
632      if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
633          goto fail;
634
```

Authenticity check

Apple SSL problem survived a year, affected OSX and iOS.

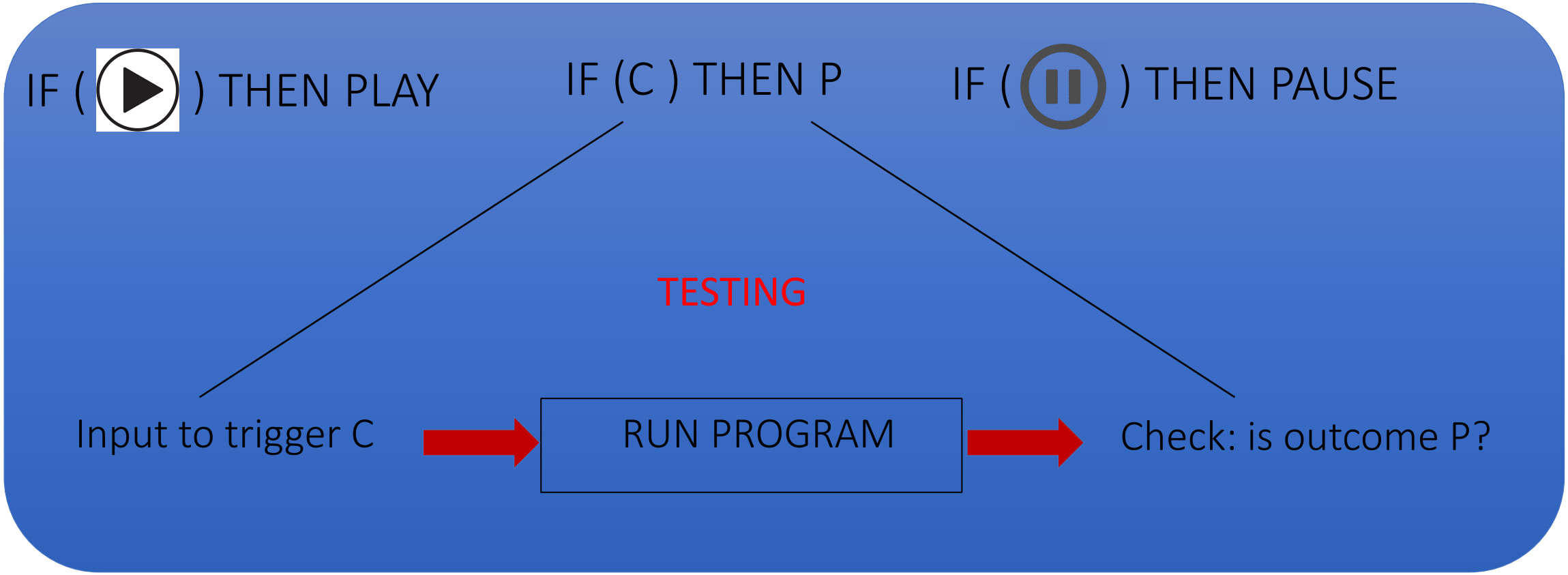
A crucial server authenticity check is skipped because of the unconditional goto – an inadvertent mistake or malicious?

```
1   /* a1, a2 are constants, and X, Y, Z, and d are variables */
2   /* Other variables not shown here define the conditions C1, C2,
3   and C3 for the B1, B2, B3 branch nodes in the CFG */
4       X = a1+a2
5       d = a1;
6       if (C1) {
7           X = a1;
8       }
9       else {
10          Y = a2;
11      }
12      if (C2) {
13          if (C3) {
14              Y = a1;
15          }
16          else {
17              d = d-a1;
18          }
19      }
20      else {
21          d = d+1;
22      }
23      Z = X/d;
```

1. What is a potential vulnerability?
2. Why will the vulnerability be not noticed every time the program runs?
3. What probability would you assign to predict how often the vulnerability will be noticed during multiple program runs?

Software Testing

- It has been the predominant software assurance solution.



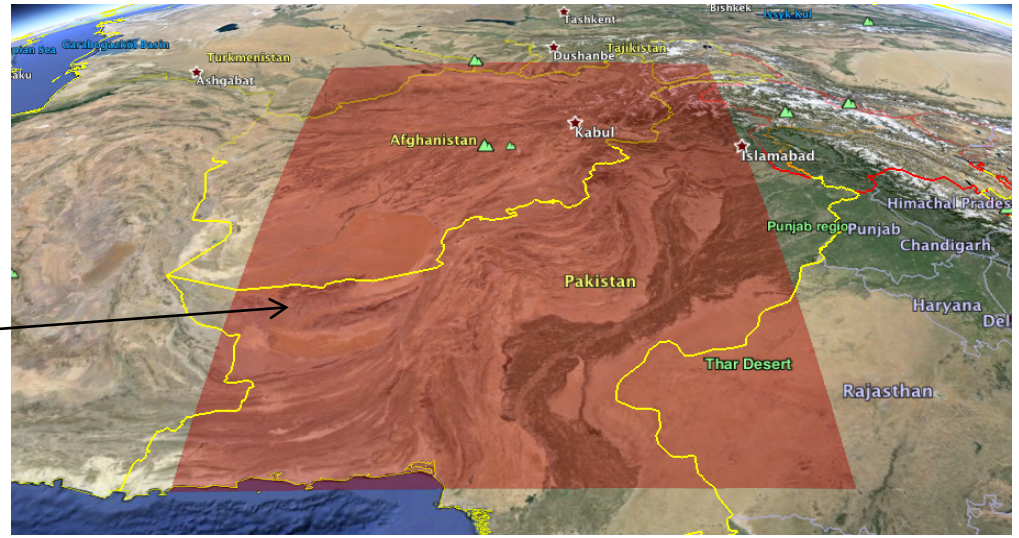
Cybersecurity: Testing is not enough

- Software Testing is based on requirements: IF (C) THEN P
- Cybersecurity does not have a specified input C
- Cybersecurity does not have a specified outcome P
- Fundamental challenge: cybersecurity does not have specified *requirements*

Cybersecurity: Look for a needle in the haystack, not knowing what the needle looks like!

Catastrophic malware with an obscure trigger

```
@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);
   }
        ...
}
```



Small malicious code in a large Android app.
This is an example of *integrity* breach.

Where is this malware triggered?

The obscure trigger makes it inordinately difficult to detect malware through testing.

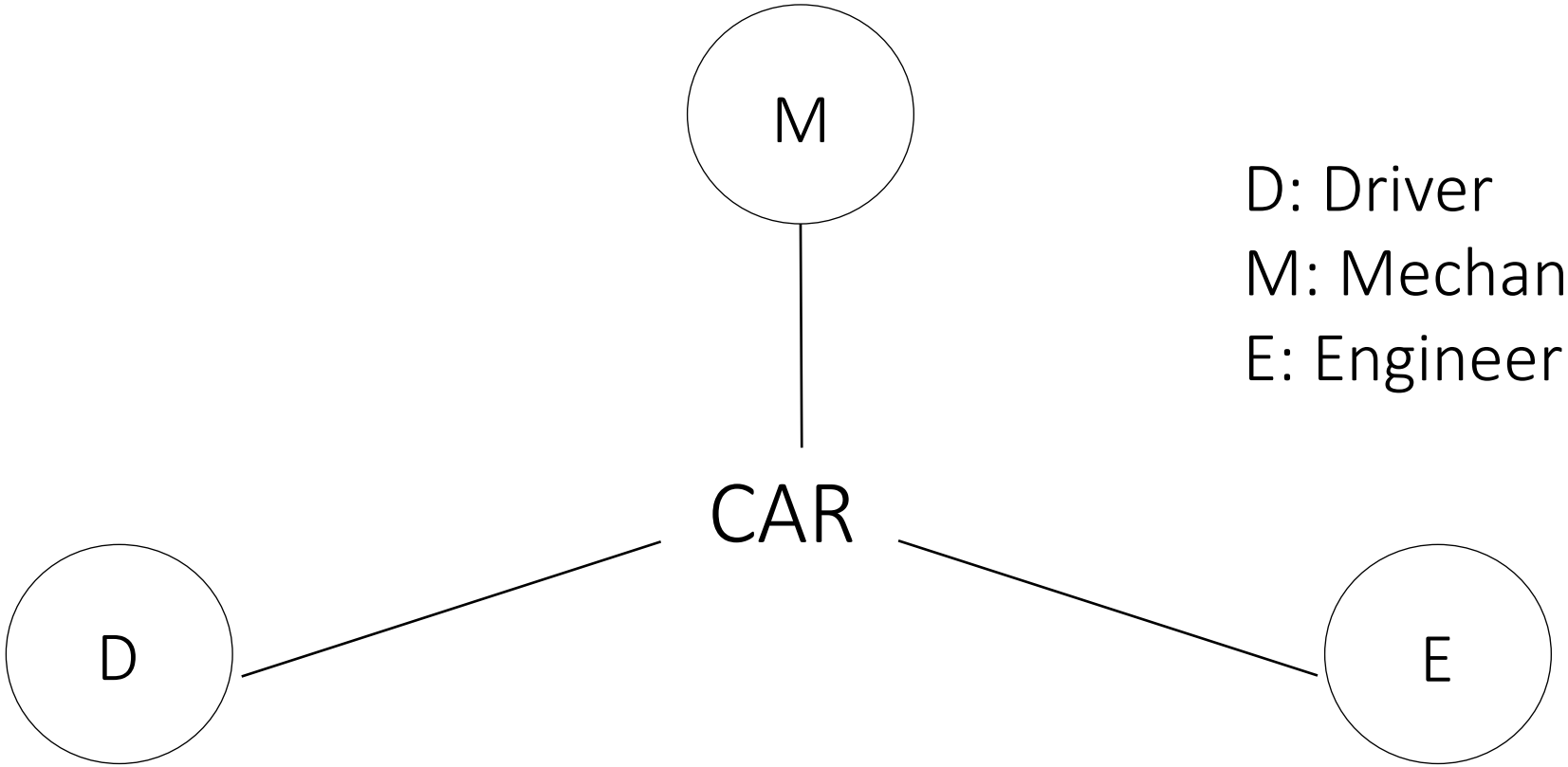
Two aspects of hard problems

- Hypothesis
  - First, come up with a specification of what the potential vulnerability is – *not knowing what the needle looks like*
- Verification
  - Then, verify the vulnerability using the specification – *find the needle in the haystack*

Multiple views of cybersecurity

- We will bring out the differences between multiple views of cybersecurity by drawing a parallel:
  - Car vs. Cybersecurity
  - Three categories of people

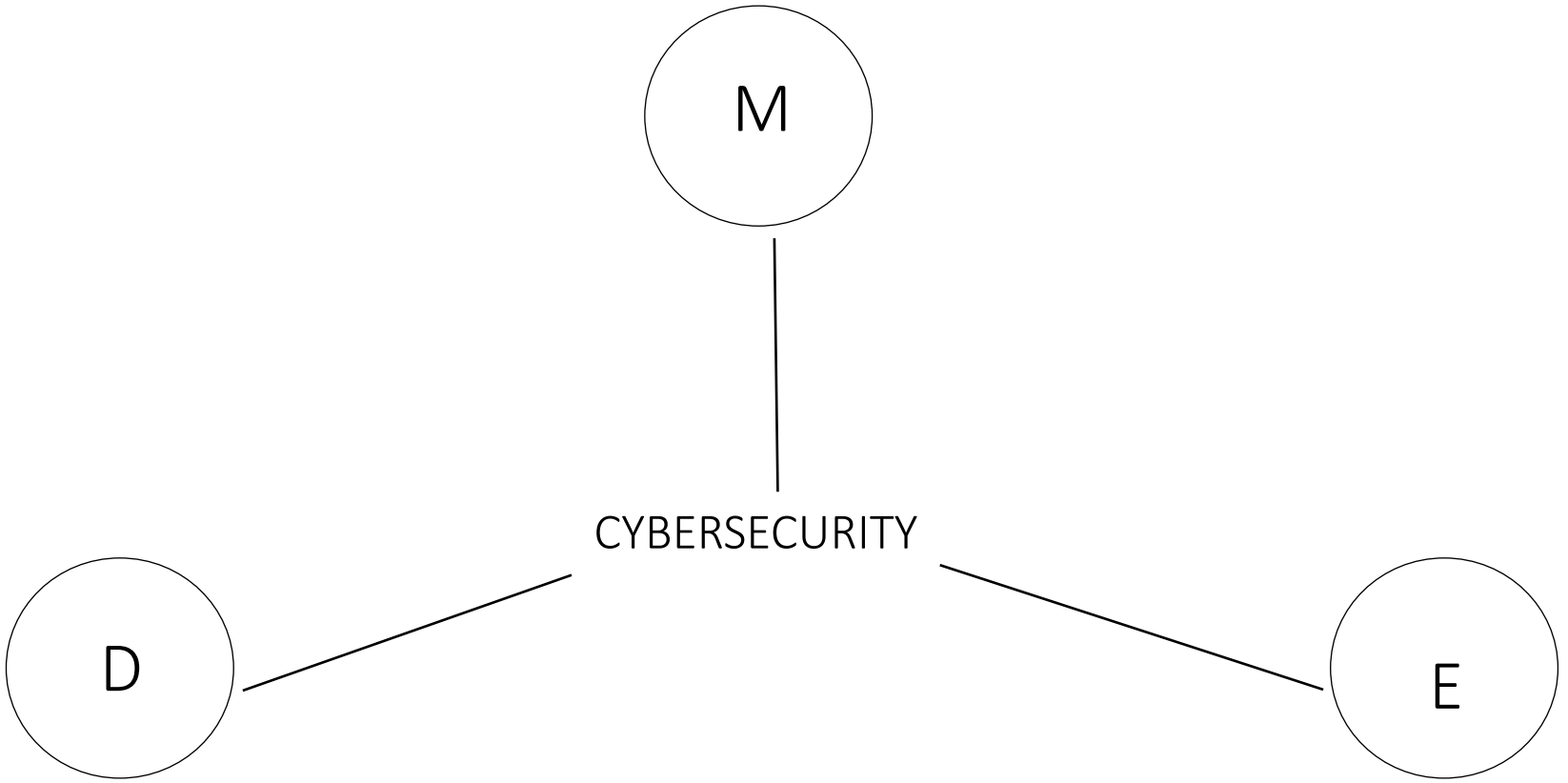
Three categories of car people



D: Driver
M: Mechanic
E: Engineer

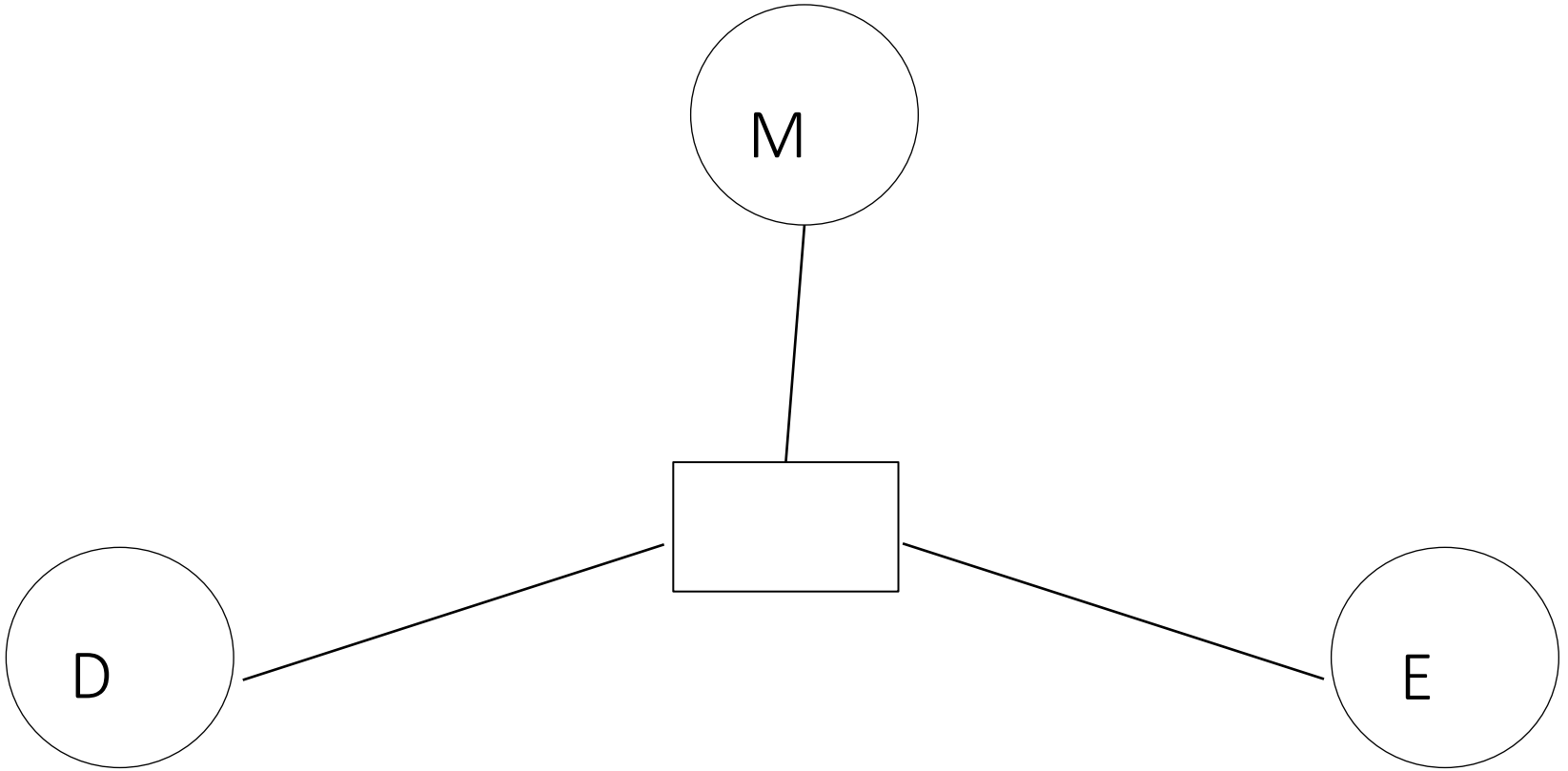
All interested in cars but for different reasons

Three categories of car cybersecurity people



All interested in cybersecurity but for different reasons

Knowledge filtration and encapsulation



Knowledge filtered out or encapsulated in going from E ⟶ M ⟶ D

D & M Type Cybersecurity

- Many D Type Cybersecurity jobs: Learn how to use a tool. The tool is automatic; it produces a report. The job requires the knowledge of running the tool and refining the report produced by the tool.
- Relatively fewer M Type Cyber jobs: Learn how to assemble a tool that D personnel uses. The job requires the knowledge of known vulnerabilities, the available components to build the tool and the knowledge of assembling the components.

Fragility of automated tools

CVE-2012-4681: “Allows remote attackers to execute arbitrary code via a crafted applet that bypasses SecurityManager restrictions…”

A classroom experiment two years after the CVE:

Sample	Notes	Positive detection
Original Sample	http://pastie.org/4594319	30/55
Technique A	Changed Class/Method names	28/55
Techniques A-B	Obfuscate strings	16/55
Techniques A-C	Change Control Flow	16/55
Techniques A-D	Reflective invocations (on sensitive APIs)	3/55
Techniques A-E	Simple XOR Packer	0/55

E Type Cybersecurity

- E-Type is important because:
  - New cybersecurity problems with catastrophic consequences
  - These problems cannot be solved using the current technology of automated tools.
  - Terrorist organizations and nations engage in cyberwarfare.
- Government and industry desperately need E-Types but they cannot find them
- Becoming E-Type is challenging
  - Science and engineering for E-type has a long way to go from its current primitive state.
  - Prevalent misconceptions
  - E-Type requires deep knowledge of software engineering, compilers, operating systems, and mathematics.

A broad categorization of research and practice

- Cybersecurity research and practice can be broadly categorized as:
  1. Most cybersecurity problems are rooted in software – *analyze and verify software*.
  2. Most cybersecurity problems propagate through computer networks – *analyze and monitor networks*.
  3. Software systems and computer networks must be properly administered – *administer and protect systems and networks*.

Overarching aspects of cybersecurity vulnerabilities

- **Damage** – categorized using the CIA model
  - ***C****onfidentiality* breach: secret information is leaked, e.g., photos are leaked by a smartphone app
  - ***I****ntegrity breach*: a critical function is compromised, e.g., GPS gives wrong location information.
  - ***A****vailability breach*: a critical resource becomes unavailable, e.g., the battery is drained quickly by a smartphone app
- **Trigger** – something that activates the damage causing functionality.
  - Environment, e.g., the region or time triggers the GPS malfunction
  - User interaction, e.g., the user click triggers leak.
- **Program construct** – Mechanism that enacts the damage
  - A modification of the GPS information governed by a trigger condition

DARPA APAC Project

- **Automated Program Analysis for Cybersecurity (APAC):** Detect sophisticated vulnerabilities in Android apps.
- Requirement: Analyze Java code, the resource and GUI files, and the Android APIs used by the app.
- Six Blue teams and two Red teams.
- This project ended in February 2015: ISU-EnSoft the top performing Blue team in Phase I, and among the top 3 teams in Phase II.

DARPA STAC Project

- **Space/Time Analysis for Cybersecurity (STAC)**: attacks use the knowledge of variations in space-time complexities along different execution paths to design denial of service or side channel attacks.
- Requirement: Analyze Java byte code to detect *algorithmic complexity* (AC) and *side channel* (SC) vulnerabilities.

People
Tools
Atlas
Atlas Toolboxes
FlowMiner
LSAP

Publications

- Papers (11)
- Short Courses (1)
- Talks (3)
- Tutorials (7)
- Upcoming (4)

Monthly Activity

- October 2016 (3)
- September 2016 (3)
- 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
- Jeremías Sauceda
- Jon Mathews
- Nikhil Ranade
- Payas Awadhutkar
- Sandeep Krishnan
- Suresh Kothari

People

The Knowledge-Centric Software Laboratory at the Department of Electrical and Computer Engineering consists of a group of researchers interested in solving hard problems in software program analysis. The laboratory is led and directed by professor and entrepreneur Dr. Suresh Kothari.

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

Director

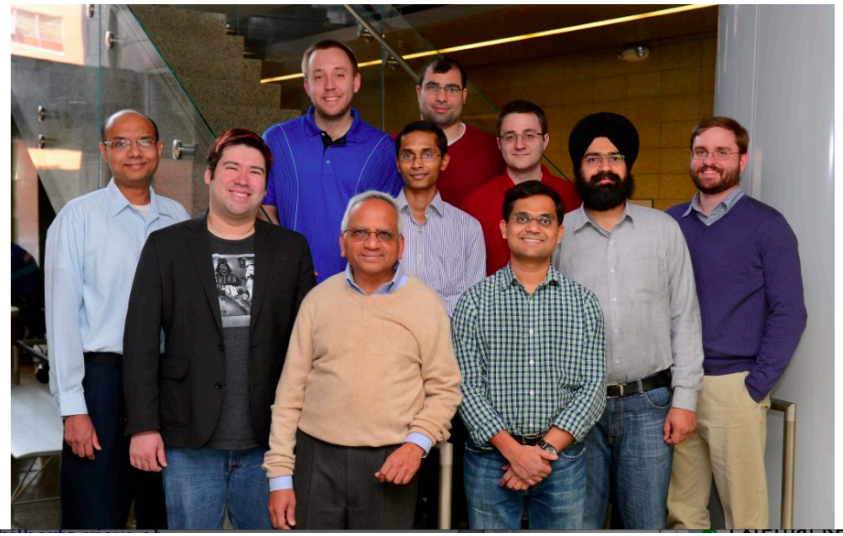
- Suresh Kothari (Richardson Professor)

Current Members

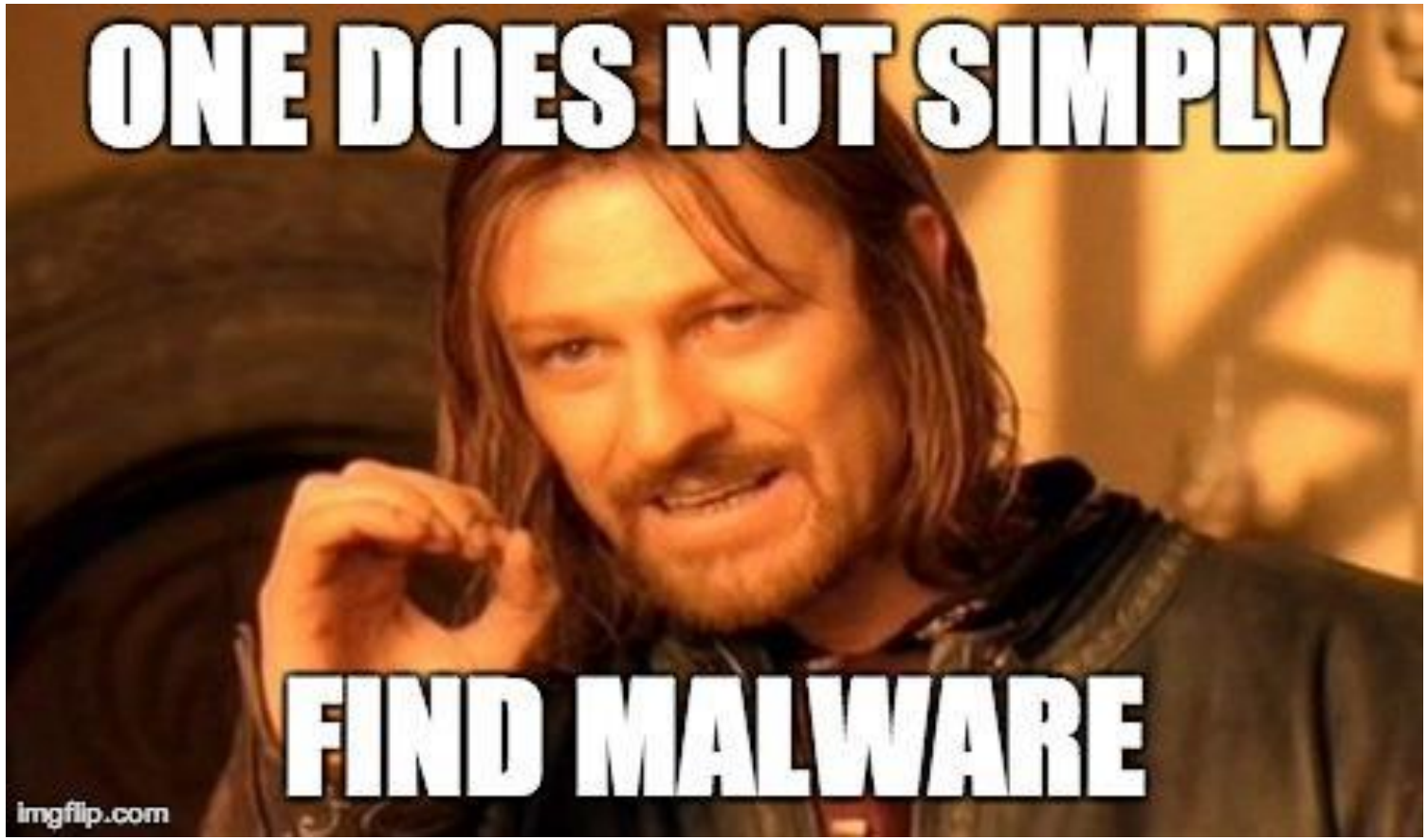
- Benjamin Holland (Graduate Student)
- Ganesh Ram Santhanam (Associate Scientist)
- Payas Awadhutkar (Graduate Student)

Past Members

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



Knowledge-Centric Software Engineering Lab:
http://www.ece.iastate.edu/kcsl/



Key Question: How do we reason systematically about software?

Problem: Find and fix vulnerabilities in software.

This is too general. We need specifics to get started.

Buffer overflow Problem

- Actions that cause the problem:
  - User gives large input
  - The input is written to the buffer
- What happens
  - Input is bigger than the size of the buffer
  - Memory outside the buffer is overwritten
  - The overwriting causes a crash
  - Input could be malicious code which executes to cause even serious damage

Confidentiality problem

- Actions that cause the problem:
  - User opens the camera app
  - User taps the screen to focus
- What happens
  - The previewed scene is captured
  - The captured scene is sent to attacker's website

Integrity problem

- Actions that cause the problem:
  - User enter a particular region of Afghanistan
  - User reads the location from the the GPS
- What happens
  - The location information is corrupted by the malicious software whenever the GPS is in a certain geographic region

Availability problem

- Actions that cause the problem:
  - User sends email on Friday the 23th at 12 noon
- What happens
  - A malicious loop runs to drain the battery

How do we specify cybersecurity vulnerabilities?

How do we describe what the *needle looks like*?

- Endless specificity is not helpful either – we cannot analyze and verify software with an endless list of cybersecurity problems.
- The challenge is to model the cybersecurity problems with good abstractions that cover the problem space without getting into an endless listing of problems.

A reminder from algebra

Problem 1: John is 3 years older than Jill. Together their ages add up to their mother's age. Their mother is 45 years old. How old are John and Jill?

Problem 2: A tank holds 3 gallons more water than another tank. Together the two tanks hold 45 gallons of water. What is the capacity of each water tank?

Are these different problems?

Without knowledge of algebra, the solution will be by trial and error

$$X - Y = 3$$
$$X + Y = 45$$

Reminder: Use an abstract model to express the problem.

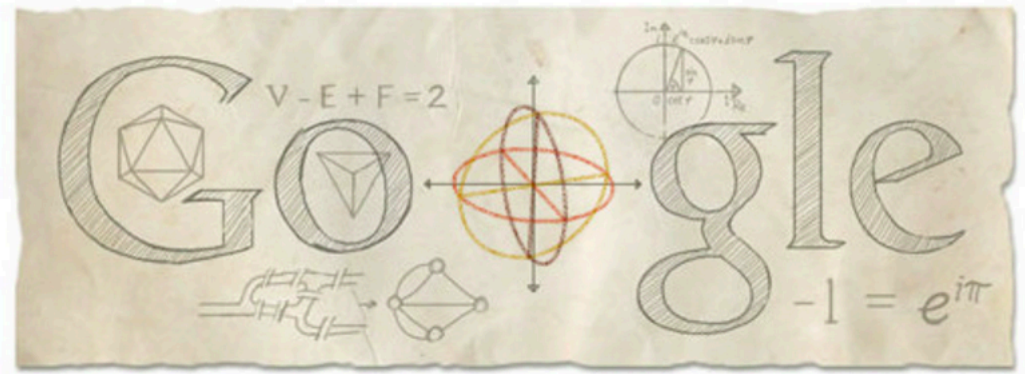
Modeling paradigm for cybersecurity?

- The algebra captures the commonality of an endless variety of problems
- It uses variables and equations.
- Cybersecurity
  - What is the commonality?
  - Should it be variables and equations or something else?

Modeling is the key! Without it, we are left with endless problems.

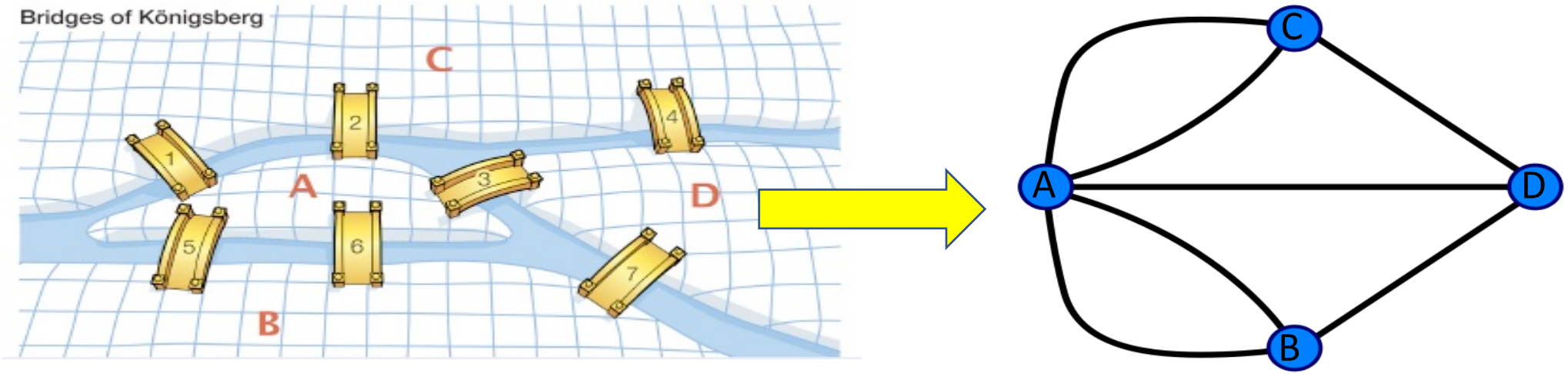
Leonhard Euler April 15, 1707 - September 18, 1783
Portrait by Johann Georg Brucker (1756) from Wikipedia

Born in Basel, Switzerland he spent most of his adult life in St. Petersburg, Russia, and in Berlin, Prussia. He made important discoveries in fields as diverse as infinitesimal calculus and graph theory. He also introduced much of the modern mathematical terminology and notation, particularly for mathematical analysis, such as the notion of a mathematical function.



The Google Doodle recalls several of his legacies. Perhaps the most significant is the Euler Equation which reveal a relationship between e, pi and i and has been described as "the most beautifual equation in all mathematics.

Euler introduced the graph concept (1735)



A loop that goes through all edges exactly once is called an *Euler loop.*

Theorem: A Euler loop exists if and only if each vertex has even degree.

This problem of bridges can be solved as a graph problem.

Aryabhata (476-550CE)

- Used place value system and zero
- Approximated π as 3.1416 and noted that it is irrational
- Introduced the notion of variables used equations with variables
- Came up with formulas: $1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$

$$1^3 + 2^3 + \cdots + n^3 = (1 + 2 + \cdots + n)^2$$

Nālandā. According to the Tang Dynasty Chinese pilgrim, Xuanzang (630-643 CE), it comes from *Na al,lllam dā* meaning *no end in gifts* or *charity without intermission*.

Let us take a concrete problem and begin to learn how to reason systematically. We shall learn important concepts and use them to articulate reasoning

What would be your answer if asked to name the two most fundamental concepts about software?

```
2   /* Other variables not shown here define the conditions C1, C2,
3   and C3 for the B1, B2, B3 branch nodes in the CFG */
4       X = a1+a2
5       d = a1;
6       if (C1) {
7           X = a1;
8       }
9       else {
10          Y = a2;
11      }
12      if (C2) {
13          if (C3) {
14              Y = a1;
15          }
16          else {
17              d = d-a1;
18          }
19      }
20      else {
21          d = d+1;
22      }
23      Z = X/d;
```

Relevant lines of code (events): 5, 17, 21, 23

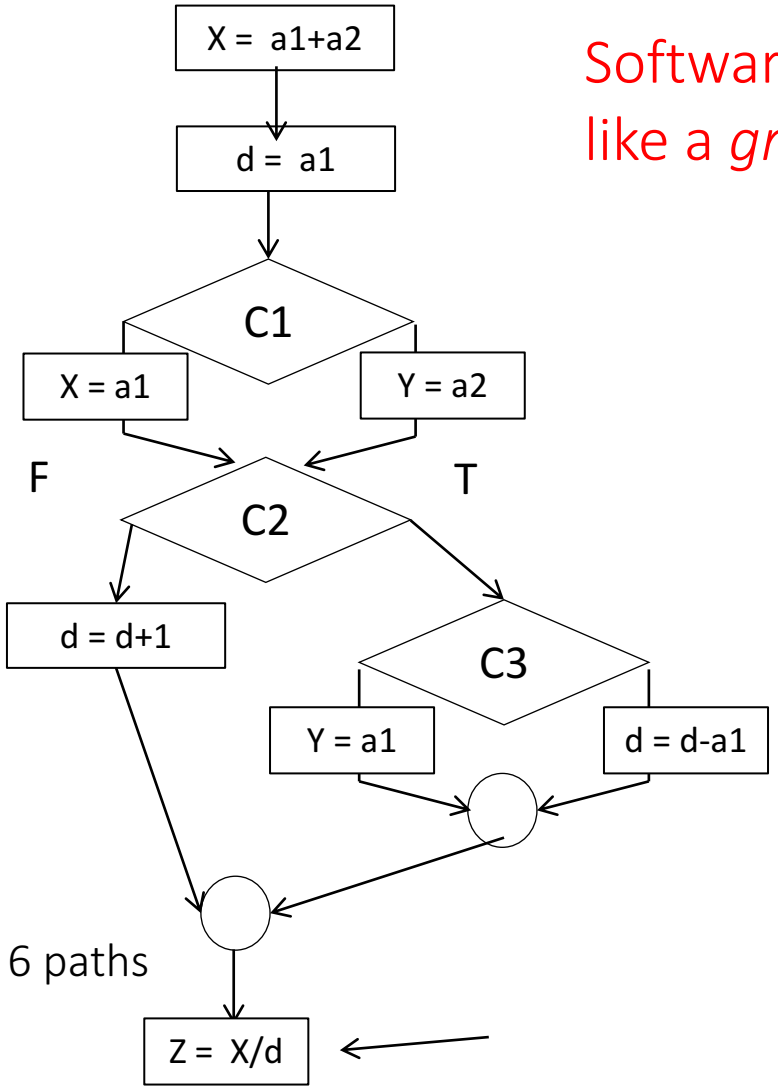
Traces:
#1: 5, 17, 23
#2: 5, 21, 23
#3: 5, 23

#1: 5, 17, 23 is vulnerable

Check if C2==T AND C3==F is satisfiable

← Division by zero vulnerability

```
1  /* a1, a2 are constants, and X, Y, Z, and d are variables */
2  /* Other variables not shown here define the conditions C1, C2,
3  and C3 for the B1, B2, B3 branch nodes in the CFG */
4      X = a1+a2
5      d = a1;
6      if (C1) {
7          X = a1;
8      }
9      else {
10         Y = a2;
11     }
12     if (C2) {
13         if (C3) {
14             Y = a1;
15         }
16         else {
17             d = d-a1;
18         }
19     }
20     else {
21         d = d+1;
22     }
23     Z = X/d;
```



6 paths

Software looks like a *string* but behaves like a *graph*!

Execution behaviors

- Each execution produces one behavior
- A behavior is a sequence of executed statements
- The power of software resides into the multitude of behaviors it can produce.
- Difficult problems involve reasoning about all possible behaviors
- Compilers can reason automatically to find certain types of errors, however, they can only reason about relatively simple problems

Division by zero problem and its generalization

- It is a difficult problem because it involve reasoning about all possible behaviors
- The problem has a single signature event – the statement with a division operation
- We shall extend our reasoning multi-event problems
- Many cybersecurity and safety vulnerabilities can be modeled as *two signature event* problems

Avoid getting lost into programming language details

1. Difficulty of reasoning = core difficulties + peripheral difficulties created by the programming languages
2. It is important to focus on the core difficulties, and not to get lost in the peripheral difficulties
3. The power of programing and so also the core difficulties boil down two simple but very powerful programming constructs.
4. One such construct is the IF statement
5. IF statement enables a very compact encoding of multitude of behaviors
6. 2^n behaviors can be encoded in a program using n IF statements

Let us first learn a key algorithm to reason about software – counting the number of behaviors in a program

Brief History

- Famous paper: “A Complexity Measure” by McCabe, 1976
- The paper introduced the widely used cyclomatic complexity measure
- McCabe announces his approach: *The complexity measure approach we will take is to measure the number of paths through a program*
- McCabe immediately changes the approach by saying: *This approach, however, immediately raises the following nasty problem ..*
- Instead of his original goal the number of paths as the complexity measure, McCabe defines the following measure: e – n + p where e is the number of edges, n the number of vertices, and p the number of components
- Had McCabe known what we shall learn shortly, his paper would probably be very different

How many behaviors in this simple case?

1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. Print t;

```
1  /* a1, a2 are constants, and X, Y, Z, and d are variables */
2  /* Other variables not shown here define the conditions C1, C2,
3  and C3 for the B1, B2, B3 branch nodes in the CFG */
4      X = a1+a2
5      d = a1;
6      if (C1) {
7          X = a1;
8      }
9      else {
10         Y = a2;
11     }
12     if (C2) {
13         if (C3) {
14             Y = a1;
15         }
16         else {
17             d = d-a1;
18         }
19     }
20     else {
21         d = d+1;
22     }
23     Z = X/d;
```

How many behaviors?

C1	C2	C3	Behavior
T	T	T	4, 5, 7, 14, 23

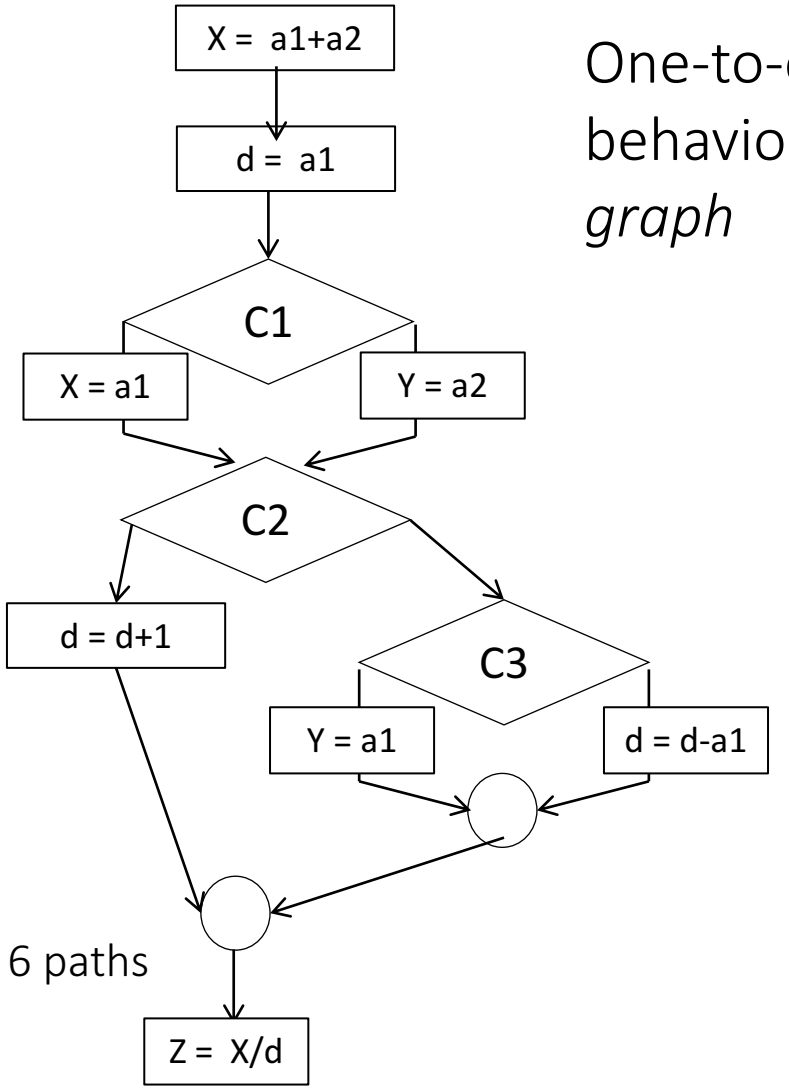
```
1  /* a1, a2 are constants, and X, Y, Z, and d are variables */
2  /* Other variables not shown here define the conditions C1, C2,
3  and C3 for the B1, B2, B3 branch nodes in the CFG */
4      X = a1+a2
5      d = a1;
6      if (C1) {
7          X = a1;
8      }
9      else {
10         Y = a2;
11     }
12     if (C2) {
13         if (C3) {
14             Y = a1;
15         }
16         else {
17             d = d-a1;
18         }
19     }
20     else {
21         d = d+1;
22     }
23     Z = X/d;
```

C1	C2	C3	Behavior
T	T	T	4, 5, 7, 14, 23
F	T	T	
T	T	F	
F	T	F	
T	F	*	
F	F	*	

```
1  /* a1, a2 are constants, and X, Y, Z, and d are variables */
2  /* Other variables not shown here define the conditions C1, C2,
3  and C3 for the B1, B2, B3 branch nodes in the CFG */
4      X = a1+a2
5      d = a1;
6      if (C1) {
7          X = a1;
8      }
9      else {
10         Y = a2;
11     }
12     if (C2) {
13         if (C3) {
14             Y = a1;
15         }
16         else {
17             d = d-a1;
18         }
19     }
20     else {
21         d = d+1;
22     }
23     Z = X/d;
```

C1	C2	C3	Behavior
T	T	T	4, 5, 7, 14, 23
F	T	T	4, 5, 10, 14, 23
T	T	F	4, 5, 7, 17, 23
F	T	F	4, 5, 10, 17, 23
T	F	*	4, 5, 7, 21, 23
F	F	*	4, 5, 10, 21, 23

```
1  /* a1, a2 are constants, and X, Y, Z, and d are variables */
2  /* Other variables not shown here define the conditions C1, C2,
3  and C3 for the B1, B2, B3 branch nodes in the CFG */
4      X = a1+a2
5      d = a1;
6      if (C1) {
7          X = a1;
8      }
9      else {
10         Y = a2;
11     }
12     if (C2) {
13         if (C3) {
14             Y = a1;
15         }
16         else {
17             d = d-a1;
18         }
19     }
20     else {
21         d = d+1;
22     }
23     Z = X/d;
```



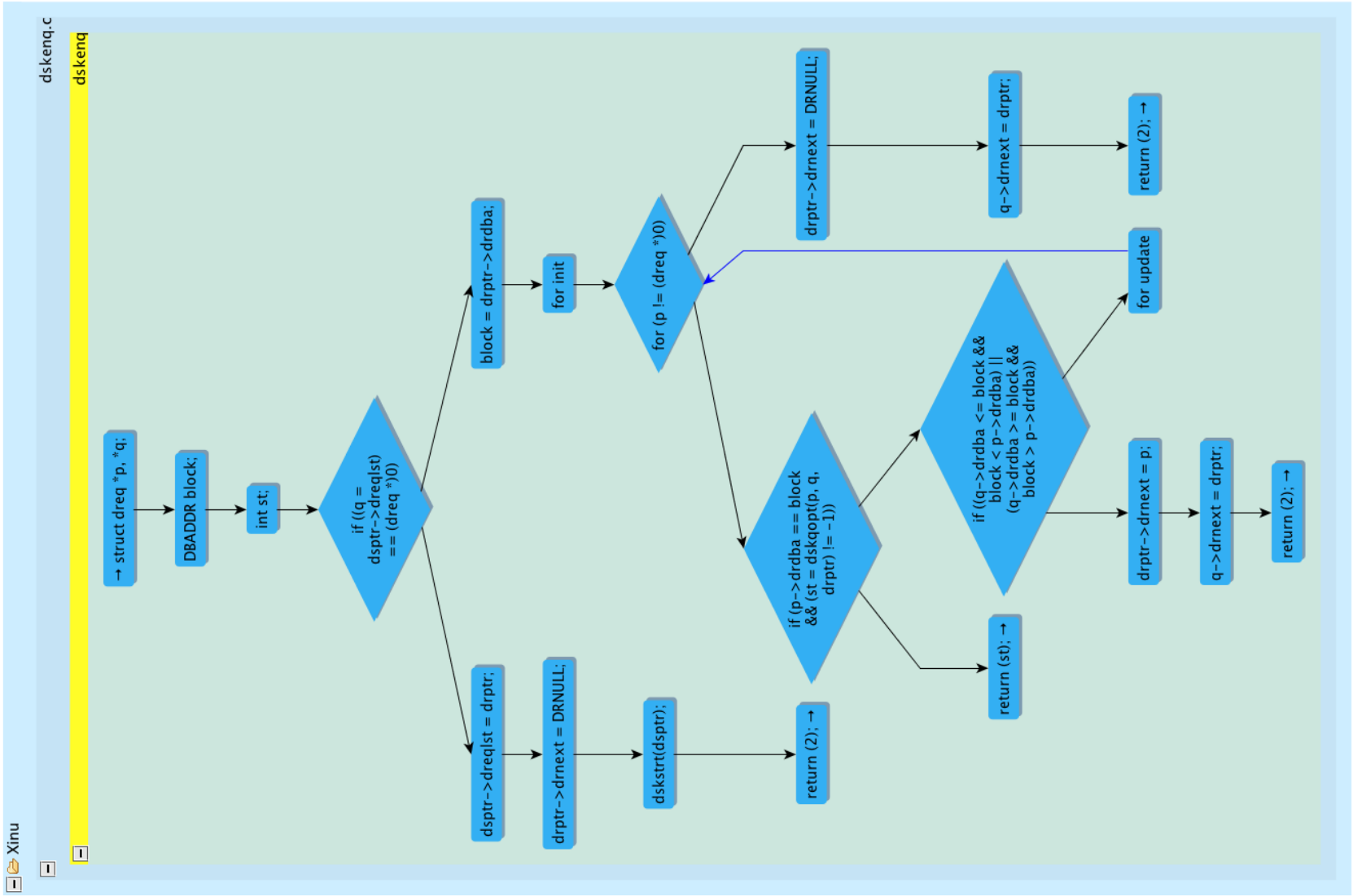
6 paths

One-to-one correspondence between behaviors and the paths in the control flow *graph*

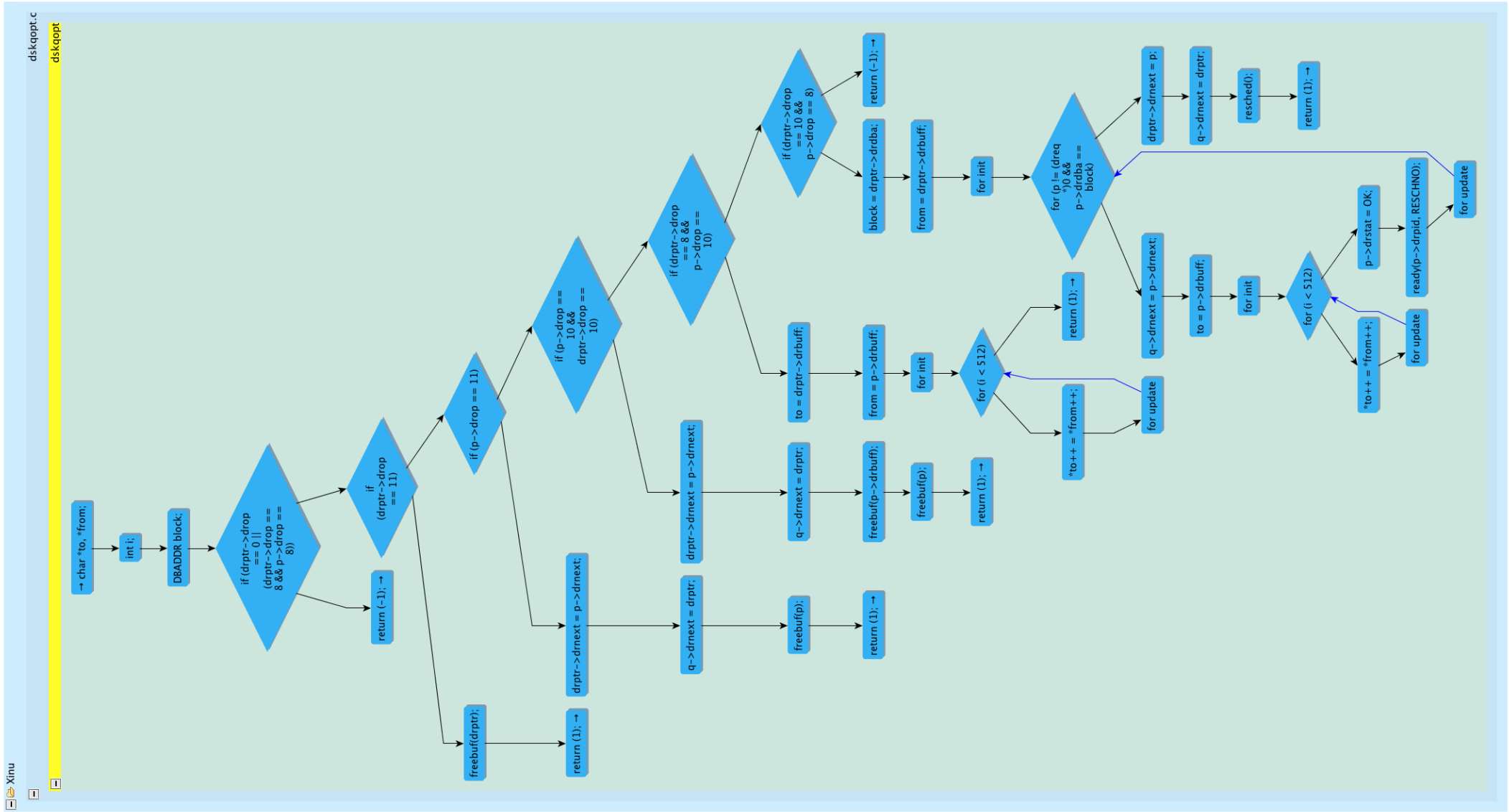
Counting the number of paths for loops

- The reasoning we can do without knowing how may times a loop would iterate:
  - Two possibilities: (1) loop is skipped, or (2) it iterates one or more times
  - Count a loop as two paths corresponding to these two possibilities
- A loop with one break point:
  - Three possibilities: (1) loop is skipped, or (2) it iterates one or more times without the break, or (3) it iterates one or more times and exits with the break
  - Count a loop as three paths corresponding to these possibilities
- Generalization: Count a loop with n breakpoints as n+2 paths
- We did experiments on open source software to assess that it is useful in practice to count and differentiate loop behaviors as suggested.

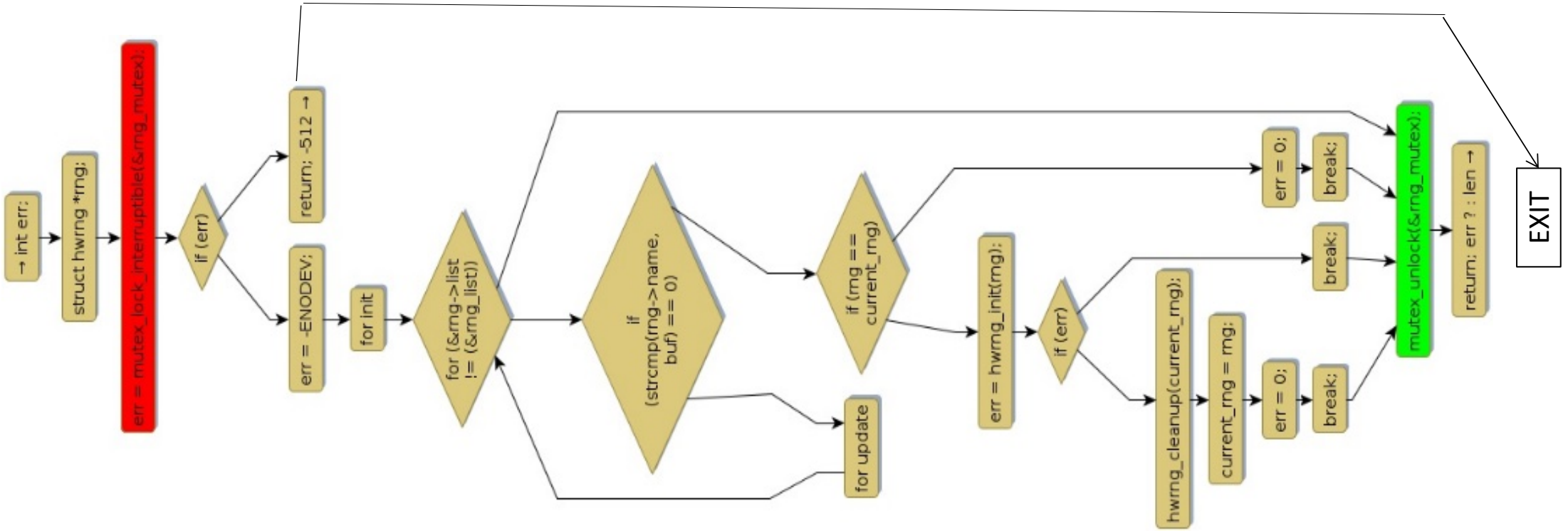
How many behaviors?



How many behaviors?



An example from the Linux kernel



- How many paths in total?
- On how many paths Lock is followed by Unlock?
- On how many paths Lock is *not* followed by Unlock?
-

See if you can find any article on Google on how to count the number of paths in a graph

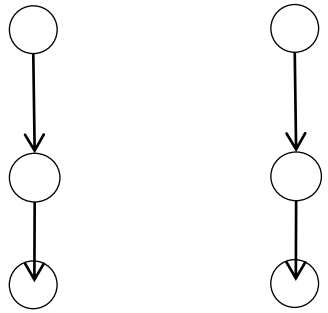
We shall learn a key algorithm to reason about software – counting the number of behaviors in a program

Watch – how we can learn by performing experiments

Counting the number of paths - Experiment 1

Imagine a graph G such that every node has, at the most, one incoming edge and one out going edge.
Let V and E be the number of vertices and edges.

How many paths in G?



Assume: Every vertex has at least one edge.

Lesson from the experiment: Efficient answer requires new concepts

Concept: connected components of a graph

A series of increasingly difficult experiments

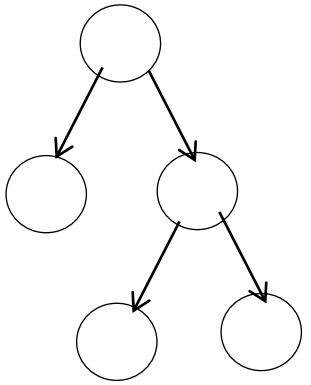
How many paths in G?

Assume: G such that every vertex has at least one edge.

1. Every node has, at the most, 1 incoming and 1 out going edge.
2. Every node has, at the most, 1 incoming and 2 out going edges.
3. Every node has, at the most, 2 incoming and 2 out going edges.

Answer: #paths = number of leaves

#1 can be solved as a special case of #2



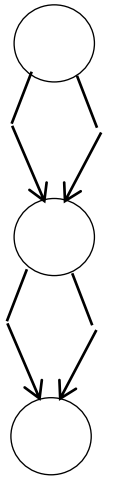
A series of increasingly difficult exercises

How many paths in G?

Assume: G such that every vertex has at least one edge.

1. Every node has, at the most, 1 incoming and 1 out going edge.
2. Every node has, at the most, 1 incoming and 2 out going edges.
3. Every node has, at the most, 2 incoming and 2 out going edges.

#3 is a quantum jump; the number of paths grows exponentially w.r.t. (V+E)

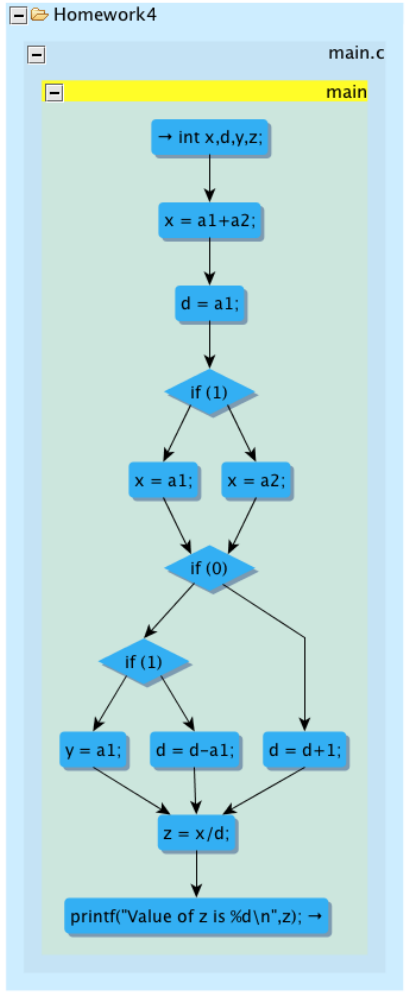


Structured program is a fundamental notion

What does it really mean?
Does structuredity mean less complexity?
Can we analyze and compute *structuredity* of programs?

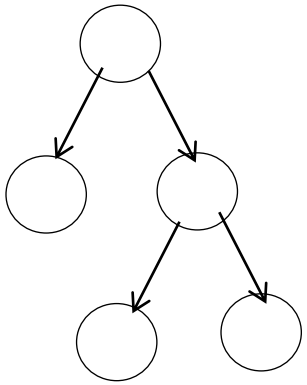
A good notion of structuredity: structuredity makes it easier to reason about programs

Structural programming to manage software complexity



- Structural programming has been advocated as a key to manage software complexity
- Languages are designed to facilitate/enforce structural programming.
- What is a structured program?
  - A lot of research since 60's
  - goto debate
- Bottom line: structured program should be easy to reason about

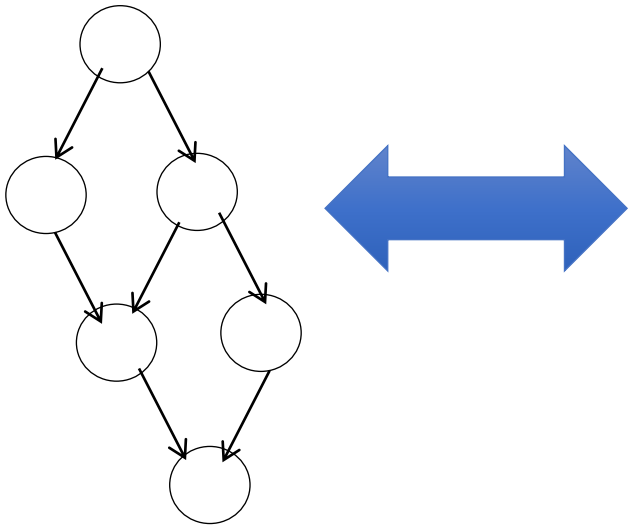
Can it be a graph of a structured program?



Program?

Answer: Place RETURN at the leaf nodes.

Can it be a graph of a structured program?



Software graph?

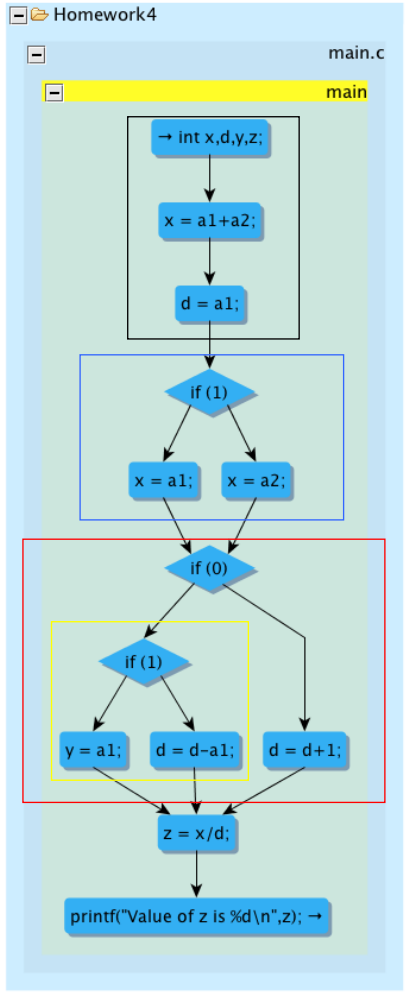
Structured program?

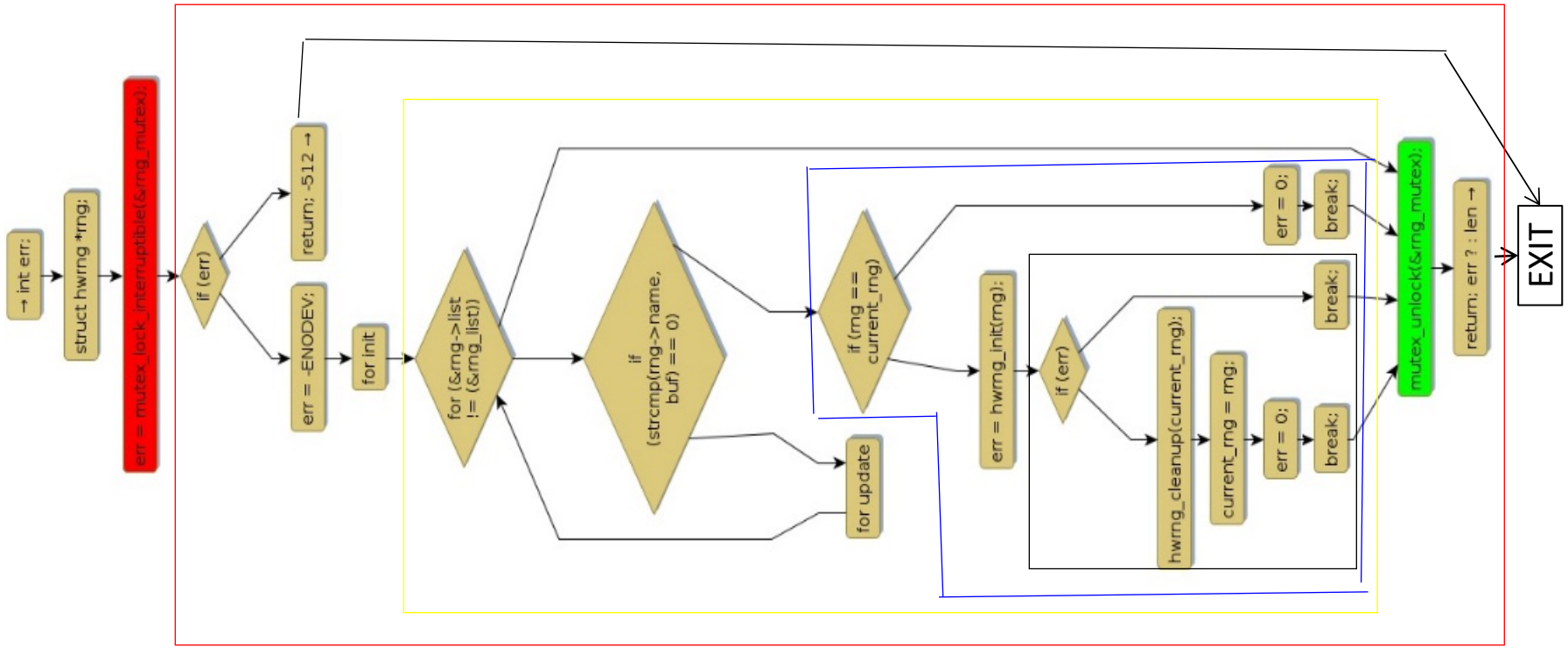
Can it be a program without goto?

Structured program defined by graph abstraction

- A program is structured if and only if the corresponding software graph consists of subgraphs with following properties:
  - The software graphs has nodes corresponding to the branch, loop, or function. The other nodes have a linear control flow.
  - The graph has one entry and one exit points
  - Each function has its own software graph with one node as the unique entry and another node as the unique exit.
  - Each branch or loop node B has a unique minimal associated subgraph with a unique entry point and a unique exit point
  - Every path begins at the unique entry point and ends at the unique exit point
  - Nodes outside the set of nodes branch, loop, and function nodes and their
  - Every node is reachable from the entry point

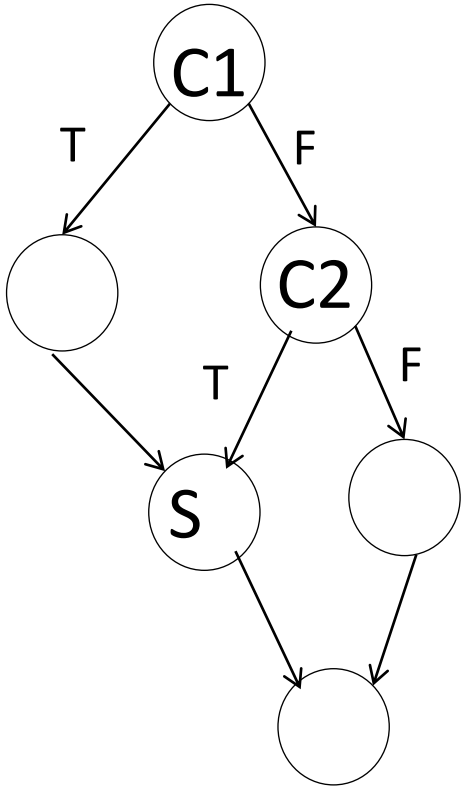
Two examples of structured program graphs





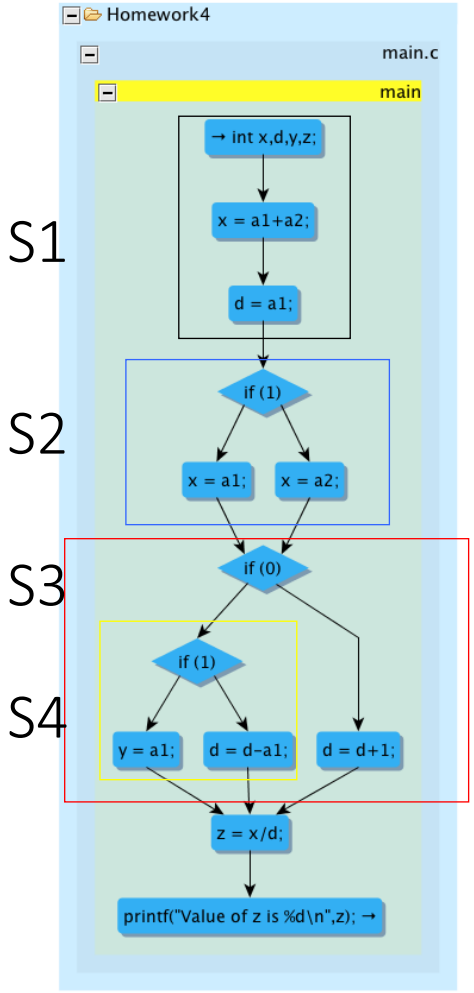
The second graph from the Linux kernel includes a loop with break points. Break points are allowed

A non-structured software graph

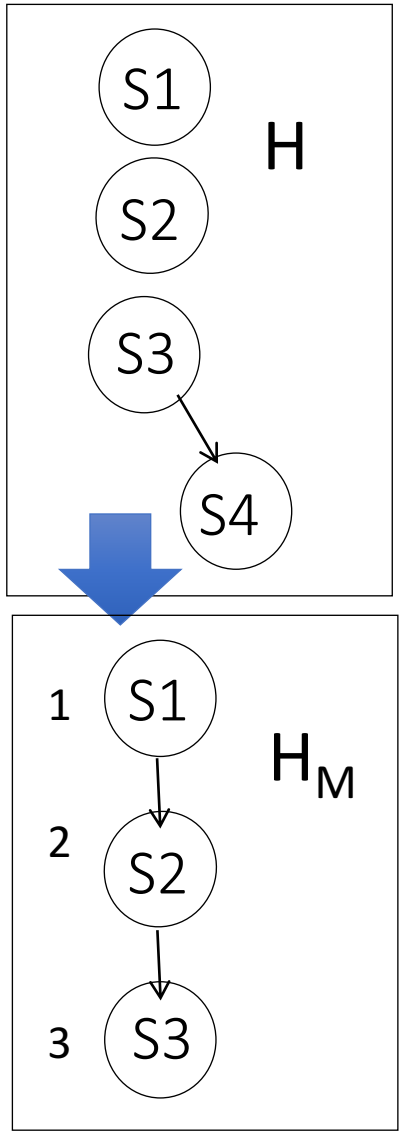


- The branch node C2 does not have a subgraph with a single entry point
- The program corresponding to this graph must have a goto
- It is difficult to reason about this program because the IF logic for C1 is messed up as S is always executed when C1 is TRUE but it is also sometimes executed when C1 is FALSE

Structured software graph enables efficient reasoning

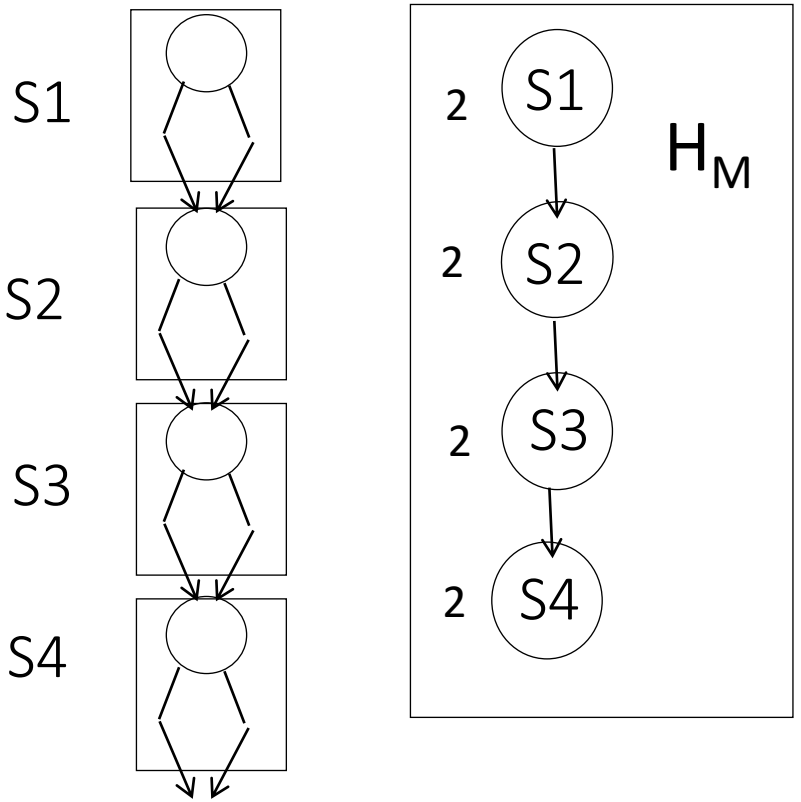


Ans: 1*2*3 =6



- A structured software graph enables an efficient algorithm to compute the number of paths in time V+E
- The crux of the efficient algorithm:
  - Define H: a graph of subgraphs – each subgraph is a node, an edge from S1 to S2 if S1 contains S2.
  - H is a lattice with min and max subgraphs
  - Compute the multiplicity for the entry node for each max subgraph
  - Take the subgraph of H_M of only the max nodes with edges induced from the original graph
  - Multiply the multiplicities along each path and add those numbers to get the total number of paths

Another illustration of the efficient algorithm



Answer: 16

Graph + Algebra: Problem 1

Bring variables into graphs! – we have a new variety of difficult problems.

A graph G with variables: A sparse subset of nodes are marked, each with one variable. Suppose we have two variables A and B.

Trace of a path: the sequence of variables along the path

Problem: Count the number of distinct traces

More difficult than counting the number of paths, because it involves the content of each path.

Graph + Algebra: Problem 2

Bring variables into graphs + prescribe rules!

A graph G with variables: A sparse subset of nodes are marked, each with one variable. Suppose we have two variables A and B.

A-Vulnerability Rule: There is a path on which A is not followed by B.

Problem: Verify that G does not have A-vulnerabilities.

Is the problem different if instead of A and B we use the letter L and U?

Graph + Algebra: Problem 3

Bring equations into graphs – a new variety of problems

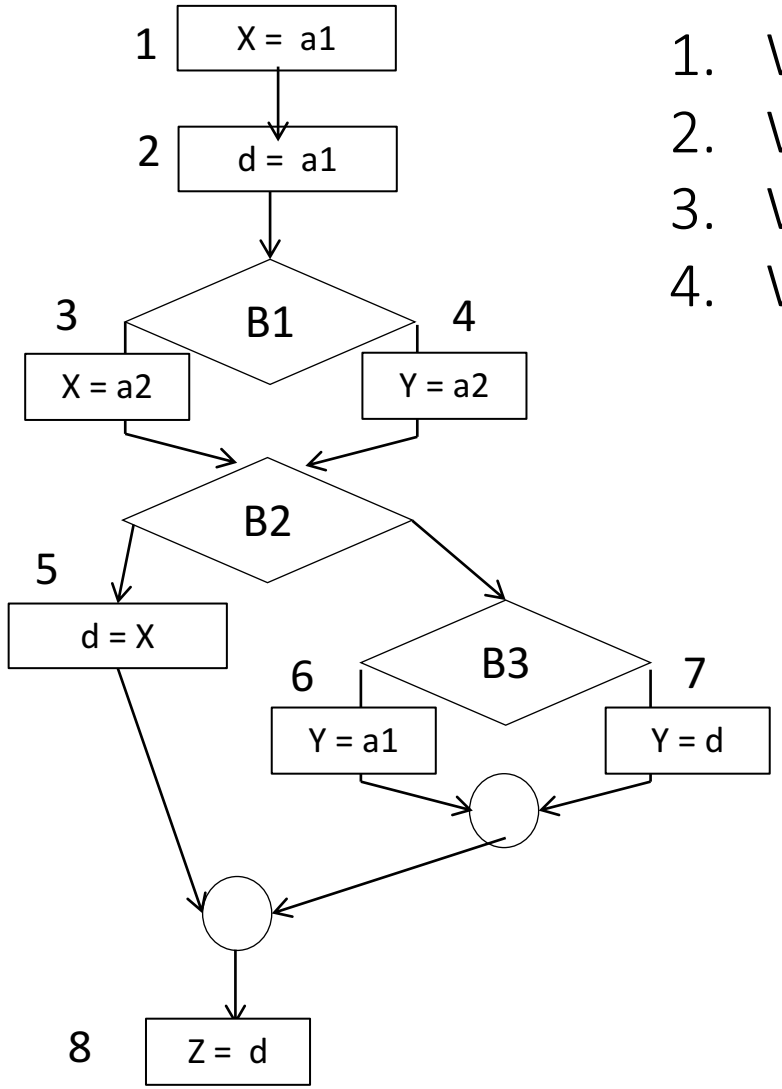
A graph G with equations: A sparse subset of nodes are marked, each with one simple equation $X_i = X_j$. Suppose we have variables X_1, X_2, .., X_k.

Think of $X_i = X_j$ as a *store-and-recall* operation. It stores X_i and recalls Xj

Problems:

1. Where does *recalled* value comes from?
2. Where does a *stored* value go?

Example: Store-Recall problem



1. What are the vertices where d is stored?
2. What are the vertices where d is recalled?
3. What are the vertices from which d is recalled at 8?
4. What are the vertices to which d reaches?

Graph + Algebra: Problem 4

Graph + Variables + Equations + Rules – a new variety of problems

A sparse subset of nodes each one marked as $A(X_i)$, $B(X_j)$ or with an equation. Let $X_1, X_2, X_3, \ldots, X_k$ be variables.

$A(X_i)$ Vulnerability Rule: There is path on which $A(X_i)$ is not followed by $B(X_j)$ and X_i reaches X_j

Problem: Verify that G does not have $A(X_i)$ vulnerabilities.

Think of A and B as operators acting on variables X_i and X_j

Is the problem different if instead of $A(X_i)$ and $B(X_j)$ we have $L(X_i)$ and $U(X_j)$?

Universal graph in Atlas

- Atlas includes schemas for language semantics so that the original program as well as some *pre-computed relationships* can be stored as a graph – referred to as the universal graph.
- Steps to create the universal graph: (a) create an Eclipse project, (b) map the project to workspace
- One gigantic universal graph is created if multiple projects are mapped to the workspace
- Atlas is especially powerful because it can create the universal graph for a complex software system that includes multiple language (e.g. Android)

Pre-computed Relationships

- Reasoning about software amounts to reasoning about relationships between program artifacts
- Atlas is powerful platform to reason about software
- It provides many readymade relationships as building blocks
- These include many control flow and data flow relationships
- The relationships are stored as edges in the universal graph
- For domain-specific reasoning the domain knowledge (e.g. entry points, Intents, permissions, resources and other Android artifacts) are precomputed and stored in the universal graph

Graph Schema

- A comprehensive and extensible open source graph called the eXtensible Common Software Graph (XCSG) the key to support software semantics
- XCSG currently captures the complete semantics of several programing languages and the Android platform
- Node or edge tagging is a commonly used extensibility XCSG feature it comes in handy in almost any reasoning about software
- XCSG is the foundation for creating an highly expressive and easy-to-use query language
- Atlas Element View shows the XCSG artifacts for a selected node or edge in the Atlas graph

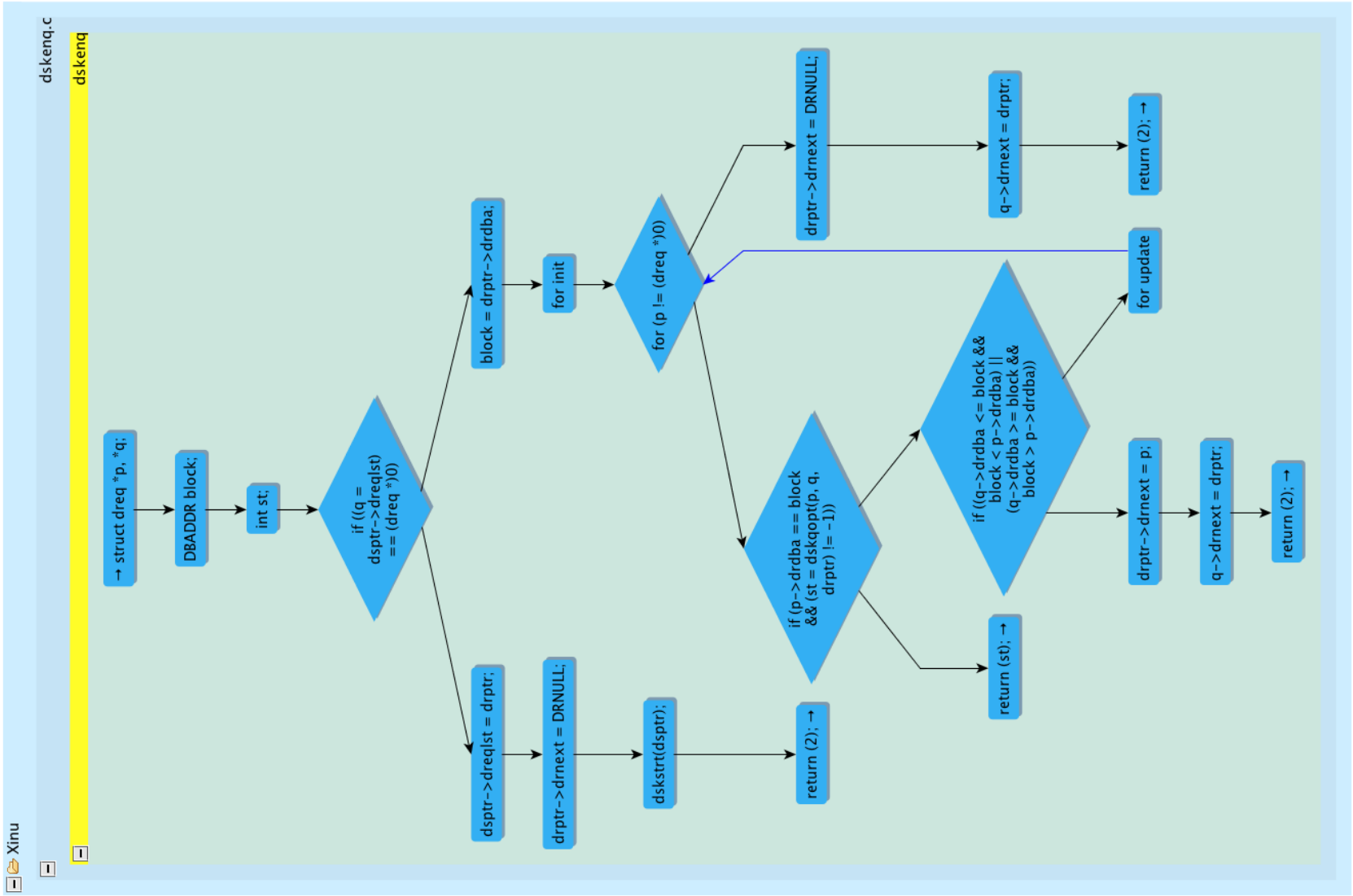
Query language

- Atlas provides a powerful query language
- A query can:
  - Specify constraints to select program artifacts (nodes) relationships (edges)
  - Traverse the graph
  - Add or remove nodes or edges
  - Add or remove attributes to nodes or edges in the universal graph
- Powerful analysis (dynamic or static) tools can be developed with just a few lines of code using the query language
- Atlas provides several readymade queries (e.g. the *cfg* query that specifies the constraints for the conntrol flow graph)

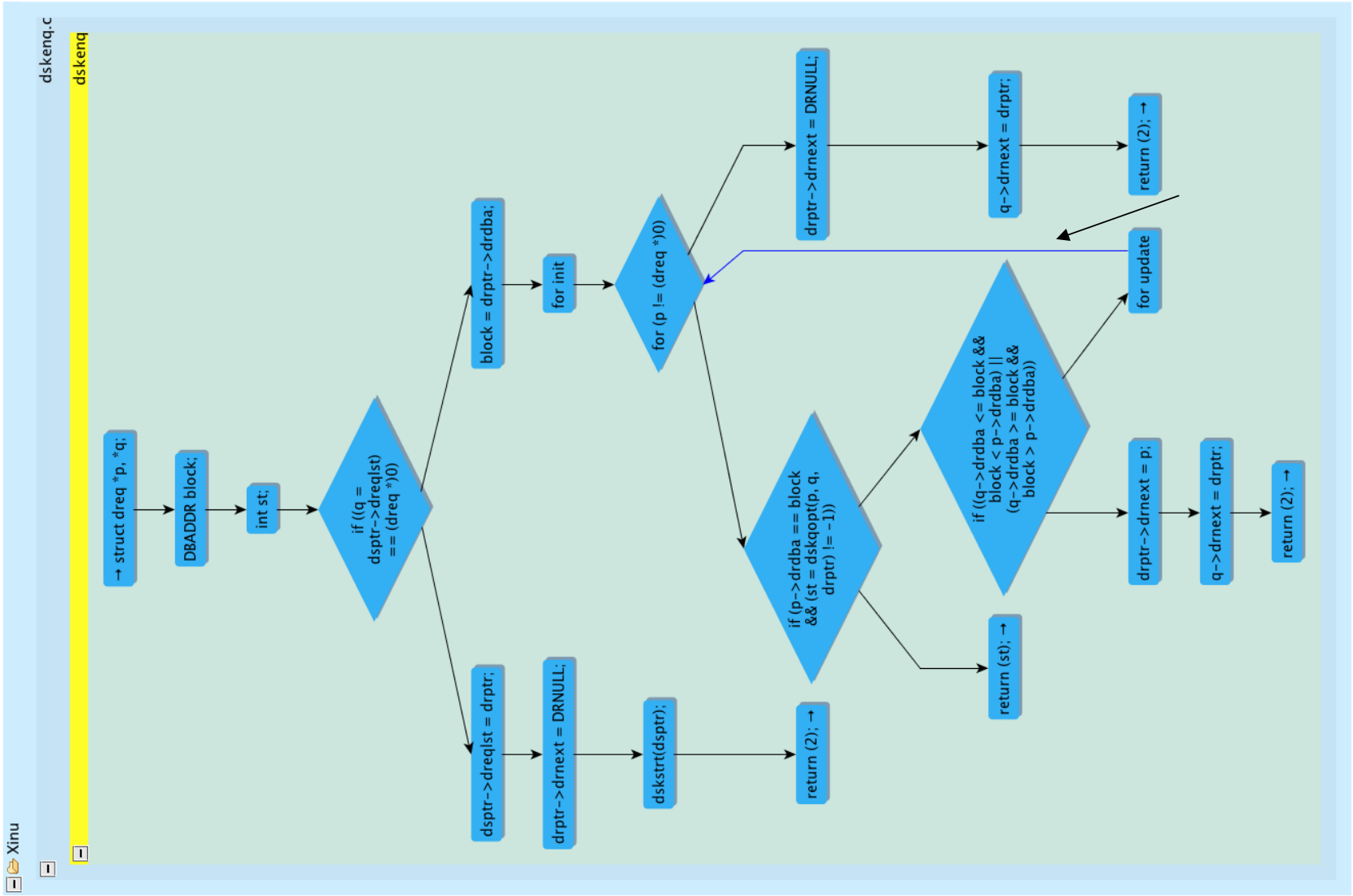
Performance optimization

- Performance optimization critical for working efficiently with millions of lines of code
- Atlas includes a highly optimized graph database engine
- For optimizing performance, Atlas incorporates two types objects: (a) graph objects, and (b) query objects
- A query object constitutes constraints to specify a graph object
- Evaluation of a query object results in a graph object
- The cost is incurred only when a query object is evaluated
- For optimized performance, continue to build the query object and evaluate it only when the actual graph is needed

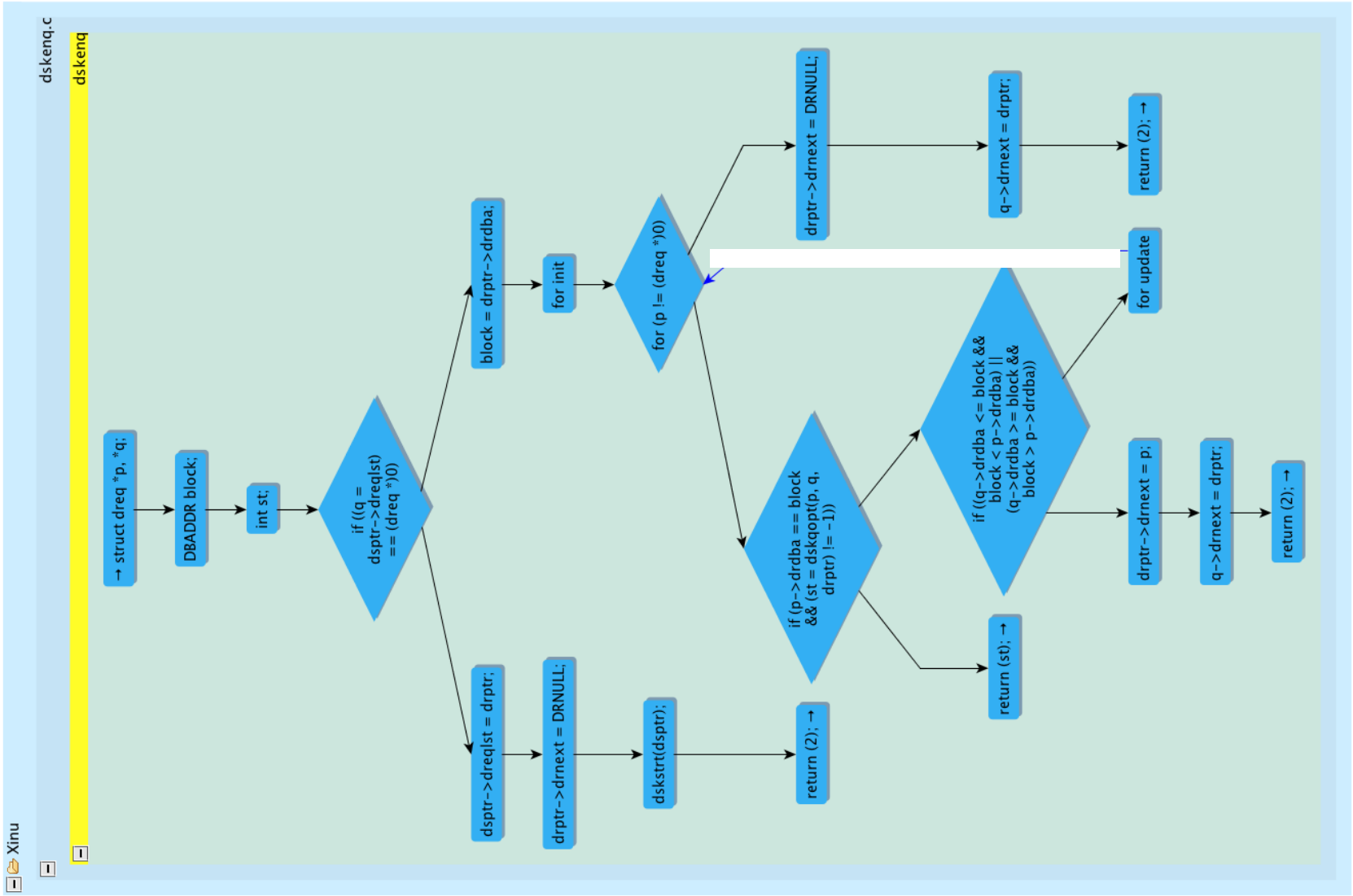
A graph created by evaluating the cfg query



Pre-computed relationship: *Back Edge*



Remove all the *back edges* by a query

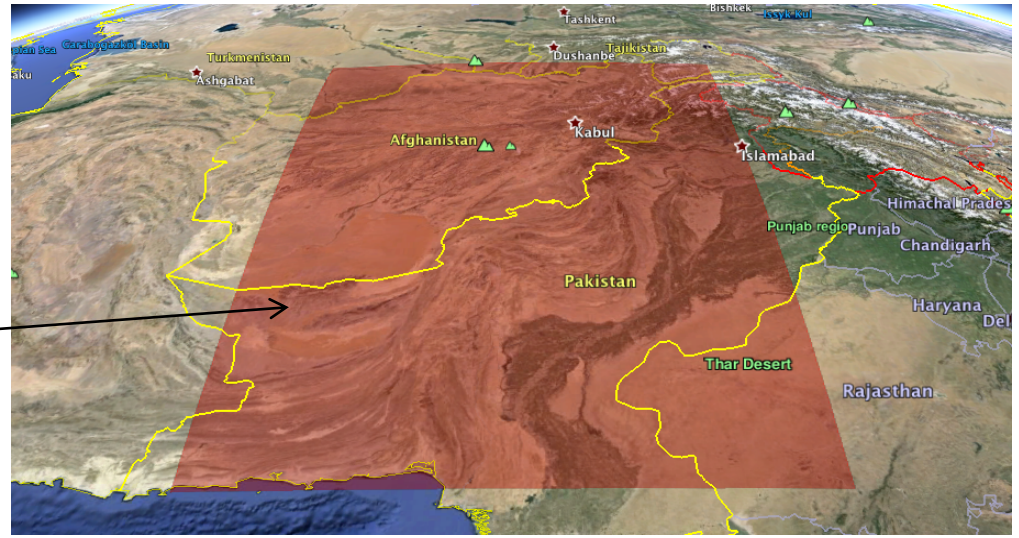


Basics of cybersecurity vulnerabilities

- **Effect** – CIA model categorizes the damage
  - ***C****onfidentiality* breach: e.g. photos are leaked by a smartphone app
  - ***I****ntegrity breach*: e.g. GPS gives wrong location
  - ***A****vailability breach*: e.g. the battery gets drained
- **Trigger** – something that activates the damage causing functionality.
  - Environment (e.g. the region or time triggers the GPS malfunction)
  - User interaction (e.g. the user click triggers leak)
- **Functional Core** – Program artifacts that enact the damaging effect

Catastrophic malware with an obscure trigger

```
@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);
   }
        ...
}
```



Small malicious code in a large Android app.
This is an example of *integrity* breach.

Where is this malware triggered?

The obscure trigger makes it inordinately difficult to detect malware through testing.

Buffer overflow vulnerability

- Actions that trigger the vulnerability:
  - User gives large input
  - The input is written to the buffer
- The malfunction:
  - Input is bigger than the size of the buffer
  - Memory outside the buffer is overwritten
  - The overwriting causes a crash
  - Input could be malicious code which can execute and cause even serious damage

Confidentiality vulnerability

- Actions that trigger the vulnerability:
  - User opens the camera app
  - User taps the screen to focus
- The malfunction:
  - The previewed scene is captured
  - The captured scene is sent to attacker's website

Integrity vulnerability

- Actions that trigger the vulnerability:
  - User enter a particular region of Afghanistan
  - User reads the location from the the GPS
- The malfunction:
  - The location information is corrupted by the malicious software whenever the GPS is in a certain geographic region

Availability vulnerability

- Actions that trigger the vulnerability:
  - User sends email on Friday the 23th at 12 noon
- The malfunction:
  - A malicious loop runs to drain the battery

How do we model cybersecurity vulnerabilities?

- The possibilities are endless for the triggers and the malfunctions of vulnerabilities
- Overarching goals:
  - A modeling paradigm that provides a compact representation to capture the endless possibilities
  - The modeling brings out the difficulty of detecting vulnerabilities

Buffer overflow verification

- Problem: Verify that each *buffer allocation* instance BA(x) (event E1) is followed by *buffer size check* BC(X) (event E2) on every *feasible* execution path
  - This is a 2-event problem.
  - Non-occurrence of the second event causes the malfunction!

Confidentiality verification

- Problem: Verify that each *sensitive source* instance SS(x) (event E1) is *not* followed by *malicious sink* MS(X) (event E2) on every *feasible* execution path
  - This is a 2-event problem.
  - Occurrence of the second event causes the malfunction!

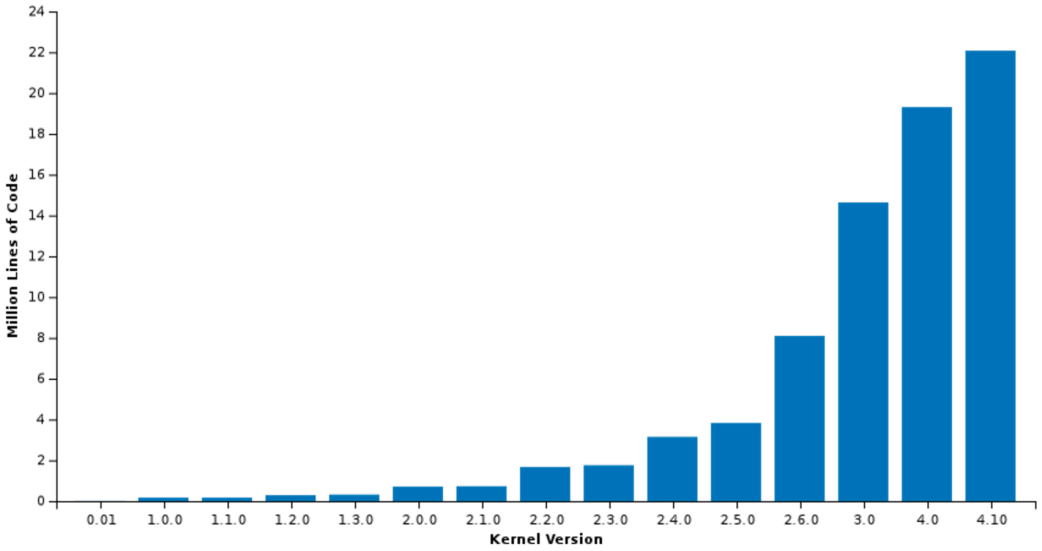
Integrity verification

- Problem: Verify that each *sensitive source* instance SS(x) (event E1) is *not* followed by *malicious modification* MM(X) (event E2) on every *feasible* execution path
  - This is a 2-event problem.
  - Occurrence of the second event causes the malfunction!

Software Verification for Cybersecurity

- Identify a set of events of interest: E_1, E_2, E_k
- Each event E is associated with a program statement: either O(X) or Y = X where O is an operator, X and Y are variables.
- The program is represented by a directed graph G and the events are a subset of G
- The *event trace of a path:* the sequence of events along the path
- *Path feasibility*: the conditions defined by the branch nodes along the path are satisfiable
- Verification:
  1. Compute the trace for each path
  2. Apply the vulnerability rule to the trace
  3. If vulnerable, check if the path is feasible

Enormous size and continuous change



Linux 1.0.0 – 176,250 LOC
Linux 4.10 – 21 M LOC

A printed version would be more than 200 feet stack of paper!

Added LOC/day	Deleted LOC/day	Modified LOC/day
12993	4958	2830

Formal verification: Lock/Unlock pairing in Linux using BLAST

Kernel	LOC	Type	Locks	Unlocks	BLAST			
					$C1$	$C2$	$C3$	Time
3.17-rc1	12.3 M	spin	14,180	16,817	8,962 (63.2%)	0	5,218	26h
		mutex	7,887	9,497	5,494 (69.7%)	0	2,393	27h
3.18-rc1	12.3 M	spin	14,265	16,917	9,152 (64.2%)	0	5,113	30h
		mutex	7,893	9,550	5,427 (68.8%)	0	2,466	30h
3.19-rc1	12.4 M	spin	14,393	17,026	9,204 (63.9%)	0	5,189	32h
		mutex	7,991	9,653	5,527 (69.2%)	0	2,464	29h
All Kernels			**66,609**	**79,460**	**43,766 (65.7%)**	**0**	**22,843**	**173h**

Two-event vulnerability: Event e1(O) must be followed by event e2(O) on all feasible execution paths.

C1: Verified Safe C2: Verified Unsafe – a feasible path with missing UNLOCK C3: Verification fails or times out
BLAST: Berkeley Lazy Abstraction Software Verification Tool

- De Millo, Lipton, and Perlis (the first recipient of the Turing Award) in “Social processes and proofs of theorems and programs”:

  - Should not take “formal” as an absolute guarantee of being correct.
  - Software verification, like “proofs” in mathematics, should provide *evidence* that humans can check to build trust into the correctness of the verification.
  - Without such evidence, it would be blind trust in software verification tools.

Cybersecurity: Formal verification is not enough

Kernel	LOC	Type	Locks	Unlocks	BLAST			
					$\mathcal{C}1$	$\mathcal{C}2$	$\mathcal{C}3$	Time
3.17-rc1	12.3 M	spin	14,180	16,817	8,962 (63.2%)	0	5,218	26h
		mutex	7,887	9,497	5,494 (69.7%)	0	2,393	27h
3.18-rc1	12.3 M	spin	14,265	16,917	9,152 (64.2%)	0	5,113	30h
		mutex	7,893	9,550	5,427 (68.8%)	0	2,466	30h
3.19-rc1	12.4 M	spin	14,393	17,026	9,204 (63.9%)	0	5,189	32h
		mutex	7,991	9,653	5,527 (69.2%)	0	2,464	29h
All Kernels			**66,609**	**79,460**	**43,766 (65.7%)**	**0**	**22,843**	**173h**

Q1 (*erroneous verification*): Among the 43766 instances verified as safe, are there unsafe instances?

Q2 (*incomplete verification*): Among the 22843 inconclusive instances, are there unsafe instances?

Q3 (*missing information*): What is so hard about the 22843 inconclusive instances that BLAST cannot verify them?

C1: Verified Safe C2: Verified Unsafe – a feasible path with missing UNLOCK C3: Verification fails or times out

BLAST: Berkeley Lazy Abstraction Software Verification Tool

Graph models as evidence

- **Macro Model**: Relevant functions and relationships between them
- **Micro Model**: For each relevant function, program statements and relationships between them

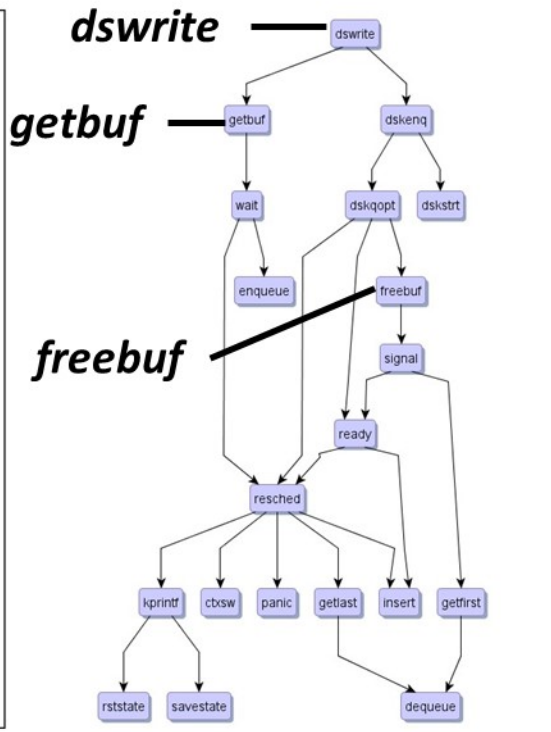
Macro Model: *Call Graph*?

```
1  dswrite(devptr, buff, block)
2     struct devsw *devptr;
3     char *buff;
4     DBADDR block;
5  {
6     struct dreq *drptr;
7     char ps;
8
9     disable(ps);
10 A  drptr = (struct dreq *) getbuf(dskrbp);
11    drptr->drbuff = buff;
12    drptr->drdba = block;
13    drptr->drpid = currpid;
14    drptr->drop = DWRITE;
15 B  dskenq(drptr, devptr->dvioblk);
16    restore(ps);
17    return(OK);
18 }
```

(a) Function dswrite

```
1  dskenq(struct dreq *drptr, struct dsblk *dsptr){
2     struct dreq *p, *q;
3     if ( (q=dsptr->dreqlst) == DRNULL ){
4        dsptr->dreqlst = drptr;
5  ①     dskstrt(dsptr);
6        return();
7     }
8     for (...){
9        if ((st = dskqopt(p, q, drptr) != SYSERR))
10    ②     return();
11       if (...){
12    ③     q->drnext = drptr;
13       return();
14       }
15    }
16 ④  q->drnext = drptr;
17    return();
18 }
```

(b) Function dskenq



(c) Call Graph of dswrite

No, it is neither necessary nor sufficient.

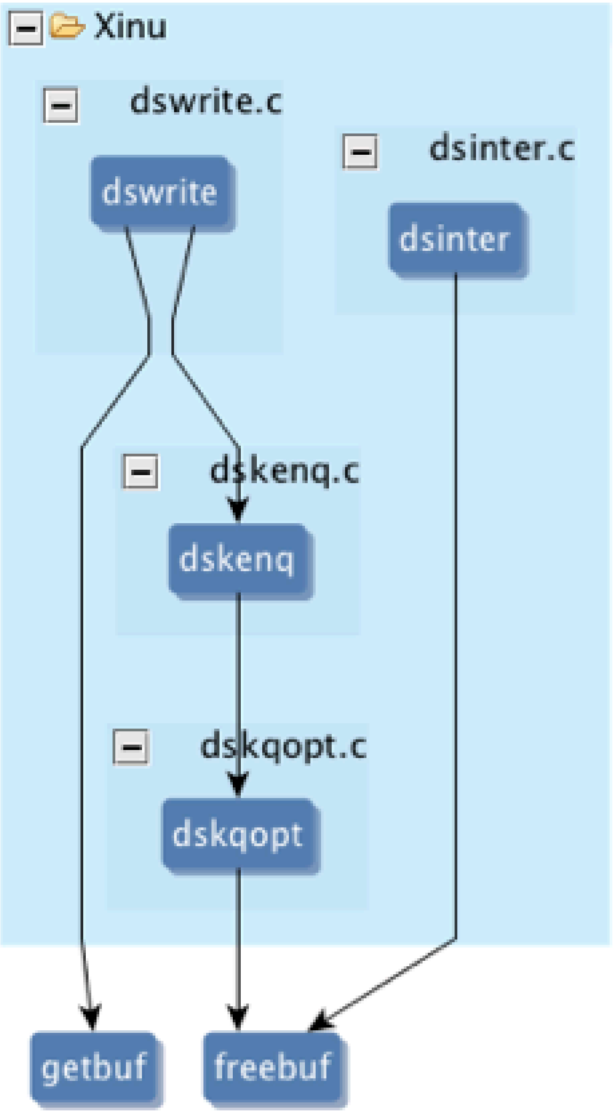
The right macro model for the example

```
1  dswrite(devptr, buff, block)
2     struct devsw *devptr;
3     char *buff;
4     DBADDR block;
5  {
6     struct dreq *drptr;
7     char ps;
8
9     disable(ps);
A    drptr = (struct dreq *) getbuf(dskrbp);
11    drptr->drbuff = buff;
12    drptr->drdba = block;
13    drptr->drpid = currpid;
14    drptr->drop = DWRITE;
B    dskenq(drptr, devptr->dvioblk);
16    restore(ps);
17    return(OK);
18 }
```

(a) Function `dswrite`

```
1  dskenq(struct dreq *drptr, struct dsblk *dsptr){
2     struct dreq *p, *q;
3     if ( (q=dsptr->dreqlst) == DRNULL ){
4        dsptr->dreqlst = drptr;
5  ①     dskstrt(dsptr);
6        return();
7     }
8     for (...){
9        if ((st = dskqopt(p, q, drptr) != SYSERR))
10    ②     return();
11       if (...){
12    ③     q->drnext = drptr;
13       return();
14       }
15    }
16 ④  q->drnext = drptr;
17    return();
18 }
```

(b) Function `dskenq`

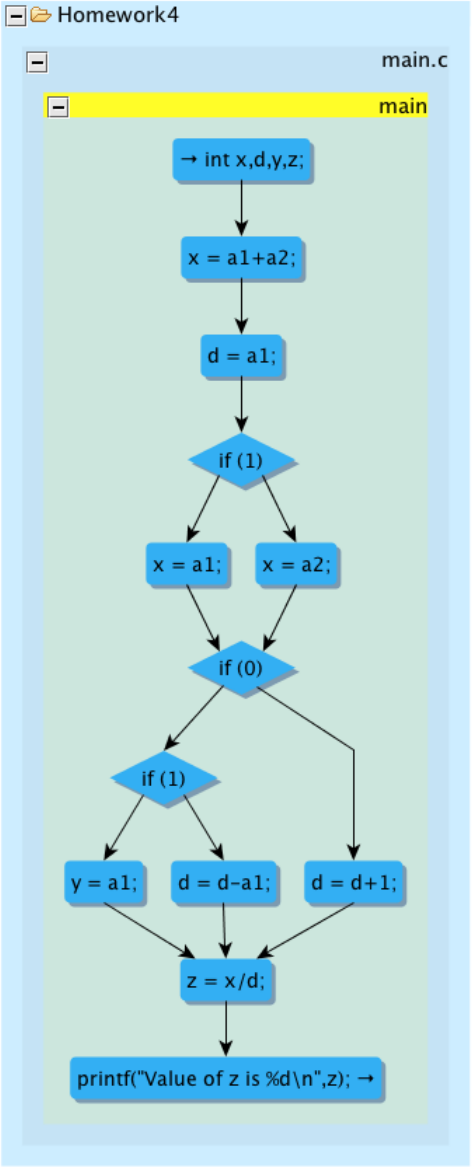


Micro Model: *Control Flow Graph*?

```
4       x = a1+a2
5       d = a1;
6       if (C1) {
7           X = a1;
8       }
9       else {
10          X = a2;
11      }
12      if (C2) {
13          if (C3) {
14              Y = a1;
15          }
16          else {
17              d = d-a1;
18          }
19      }
20      else {
21          d = d+1;
22      }
23      Z = X/d;
```

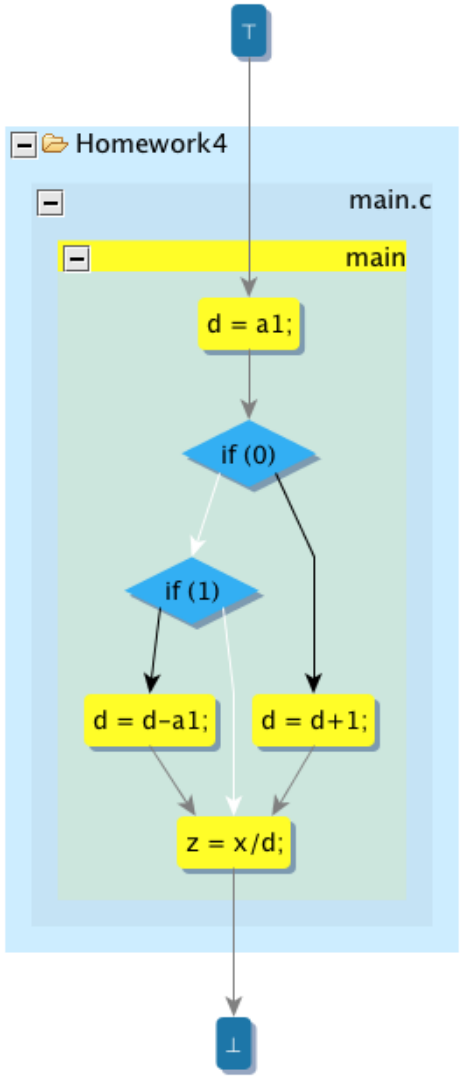
Sufficient, but not necessary.

In this example, 6 paths but only 3 distinct relevant behaviors.

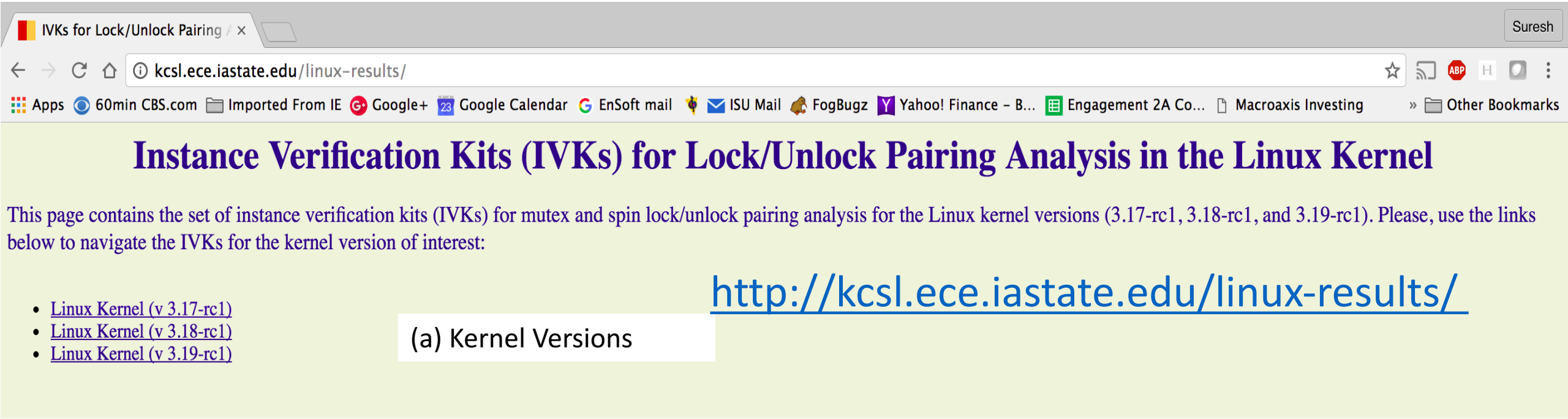


The right micro model for the example

```
4       x = a1+a2
5       d = a1;
6       if (C1) {
7           X = a1;
8       }
9       else {
10          X = a2;
11      }
12      if (C2) {
13          if (C3) {
14              Y = a1;
15          }
16          else {
17              d = d-a1;
18          }
19      }
20      else {
21          d = d+1;
22      }
23      Z = X/d;
```



Website with evidence for 66609 Lock instances



http://kcsl.ece.iastate.edu/linux-results/

(a) Kernel Versions

Used for teaching efficient proof strategies for verification.

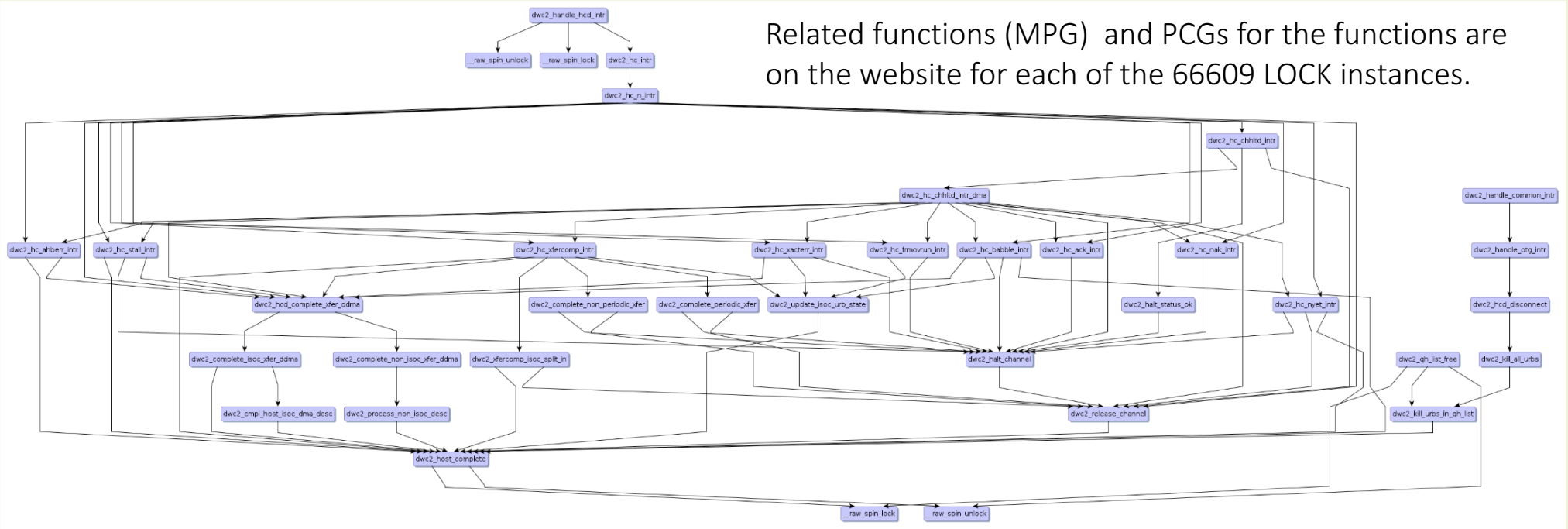
7312. [55219+47+/linux-3.17-rc1/drivers/infiniband/hw/ipath/ipath_driver.c]
7313. [40320+45+/linux-3.17-rc1/drivers/hid/hid-wiimote-modules.c]
7314. [104590+31+/linux-3.17-rc1/drivers/gpu/drm/i915/i915_gem.c]
7315. [32776+29+/linux-3.17-rc1/drivers/target/iscsi/iscsi_target_login.c]
7316. [11548+37+/linux-3.17-rc1/drivers/net/ethernet/mellanox/mlx4/eq.c]
7317. [35044+23+/linux-3.17-rc1/drivers/net/wan/sbni.c]
7318. [56597+51+/linux-3.17-rc1/drivers/infiniband/hw/ipath/ipath_driver.c]
7319. [53137+47+/linux-3.17-rc1/drivers/scsi/pm8001/pm8001_hwi.c]
7320. [25336+38+/linux-3.17-rc1/drivers/tty/serial/sh-sci.c]
7321. [3271+24+/linux-3.17-rc1/drivers/md/bcache/util.c]
7322. [30947+55+/linux-3.17-rc1/drivers/net/wireless/mwifiex/cmdevt.c]
7323. [10411+39+/linux-3.17-rc1/drivers/input/mouse/synaptics_i2c.c]
7324. [192818+34+/linux-3.17-rc1/drivers/md/raid5.c]
7325. [9037+38+/linux-3.17-rc1/drivers/usb/serial/oti6858.c]
7326. [2904+38+/linux-3.17-rc1/drivers/staging/rtl8821ae/ps.c]
7327. [53969+46+/linux-3.17-rc1/drivers/message/fusion/mptctl.c]
7328. [40290+40+/linux-3.17-rc1/drivers/net/wireless/rt2x00/rt2400pci.c]
7329. [5693+30+/linux-3.17-rc1/drivers/staging/lustre/lnet/selftest/timer.c]
7330. [212540+50+/linux-3.17-rc1/drivers/block/DAC960.c]
7331. [70804+47+/linux-3.17-rc1/drivers/gpu/drm/radeon/radeon.h]
7332. [10407+51+/linux-3.17-rc1/drivers/net/wireless/mwifiex/util.c]
7333. [18657+36+/linux-3.17-rc1/drivers/net/ethernet/realtek/8139cp.c]
7334. [66200+29+/linux-3.17-rc1/drivers/net/wireless/ath/ath10k/mac.c]
7335. [27979+31+/linux-3.17-rc1/drivers/net/wireless/ath/ath6kl/htc_mbox.c]

7312. [205003+31+/linux-3.17-rc1/drivers/platform/x86/thinkpad_acpi.c]
7313. [11027+28+/linux-3.17-rc1/drivers/net/wireless/ath/wcn36xx/smd.c]
7314. [59650+21+/linux-3.17-rc1/drivers/usb/atm/ueagle-atm.c]
7315. [4264+27+/linux-3.17-rc1/drivers/infiniband/hw/ocrdma/ocrdma_main.c]
7316. [2793+36+/linux-3.17-rc1/drivers/char/ttyprintk.c]
7317. [25126+36+/linux-3.17-rc1/drivers/scsi/qla2xxx/qla_target.c]
7318. [15722+26+/linux-3.17-rc1/drivers/net/ethernet/intel/e1000e/82571.c]
7319. [38140+30+/linux-3.17-rc1/drivers/net/wireless/rtl818x/rtl8187/dev.c]
7320. [35035+22+/linux-3.17-rc1/drivers/infiniband/hw/qib/qib_user_sdma.c]
7321. [24187+26+/linux-3.17-rc1/drivers/staging/android/ion/ion.c]
7322. [10508+24+/linux-3.17-rc1/drivers/gpu/drm/nouveau/nv50_display.c]
7323. [47208+31+/linux-3.17-rc1/drivers/hwmon/nct6775.c]
7324. [17470+42+/linux-3.17-rc1/drivers/media/usb/dvb-usb-v2/af9015.c]
7325. [3700+26+/linux-3.17-rc1/drivers/hwmon/tmp102.c]
7326. [17142+23+/linux-3.17-rc1/drivers/i2c/busses/i2c-designware-core.c]
7327. [53050+28+/linux-3.17-rc1/drivers/ide/ide-tape.c]
7328. [3576+31+/linux-3.17-rc1/drivers/hwmon/lm95241.c]
7329. [9107+23+/linux-3.17-rc1/drivers/net/wireless/ath/ath9k/channel.c]
7330. [10016+26+/linux-3.17-rc1/drivers/staging/android/ion/ion.c]
7331. [28529+34+/linux-3.17-rc1/drivers/acpi/scan.c]
7332. [1455+23+/linux-3.17-rc1/drivers/video/backlight/ld9040.c]
7333. [49102+27+/linux-3.17-rc1/drivers/memstick/core/ms_block.c]
7334. [19185+31+/linux-3.17-rc1/drivers/hwmon/lm87.c]
7335. [14422+25+/linux-3.17-rc1/drivers/mtd/lpddr/lpddr_cmds.c]

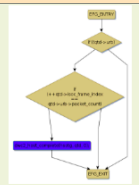
spin lock @ [62332+24+/linux-3.19-rc1/drivers/usb/dwc2/hcd_intr.c]

Instance Signature: lock

The Matching Pair Graph:



Related functions (MPG) and PCGs for the functions are on the website for each of the 66609 LOCK instances.

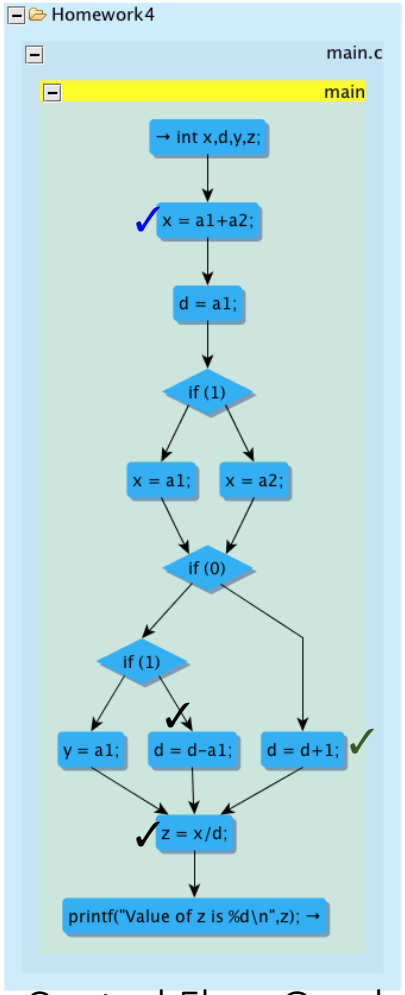
Function Name	Control Flow Graph (CFG)	Projected Control Graph (PCG)	Source Correspondence
dwc2_cmpl_host_isoc_dma_desc			**[22667+28+/linux-3.19-rc1/drivers/usb/dwc2/hcd_ddma.c]**

Automated verification results

Type	Locks	Unlocks	MBV				BLAST			
			C1	C2	C3	Analysis Time	C1	C2	C3	Analysis Time
`spin`	42838	50760	42599 (99.4%)	6	233	2h 40m 42s	27318 (63.8%)	0	15520	3d 16h 33m
`mutex`	23771	28700	23552 (99.1%)	1	218	43m 23s	16448 (69.2%)	0	7323	3d 13h 23m

A parameterized graph model

```
4       x = a1+a2
5   ✓   d = a1;
6       if (C1) {
7           X = a1;
8       }
9       else {
10          X = a2;
11      }
12      if (C2) {
13          if (C3) {
14              Y = a1;
15          }
16          else {
17  ✓           d = d-a1;
18          }
19      }
20      else {
21  ✓       d = d+1;
22      }
23  ✓   Z = X/d;
```

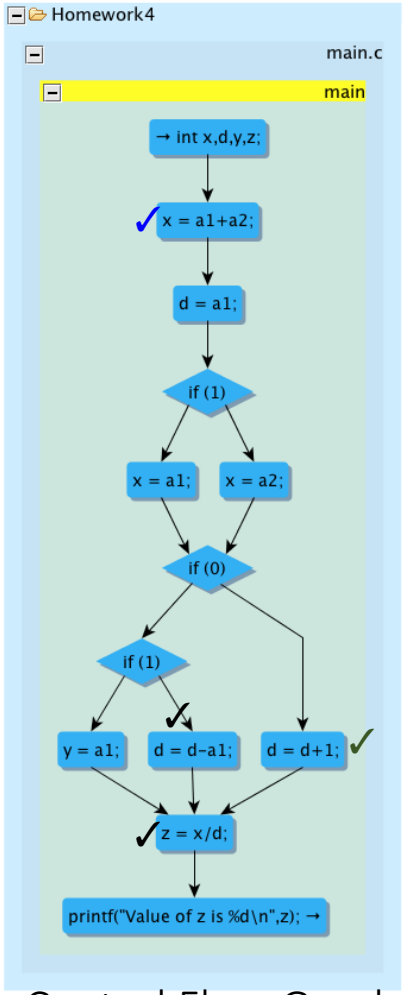


Control Flow Graph

- The model is called *Projected Control Model* (PCG).
- *Purpose*: Address the challenge of verifying all control paths.
- *Parameterized by*: a set of events that can create a vulnerability – selected by an expert or found by running an analyzer.
- *Benefit*: PCG is a compact model that captures the paths that correspond to distinct relevant behaviors.

A use of PCG

```
4       x = a1+a2
5   ✓   d = a1;
6       if (C1) {
7           X = a1;
8       }
9       else {
10          X = a2;
11      }
12      if (C2) {
13          if (C3) {
14              Y = a1;
15          }
16          else {
17  ✓           d = d-a1;
18          }
19      }
20      else {
21  ✓       d = d+1;
22      }
23  ✓   Z = X/d;
```

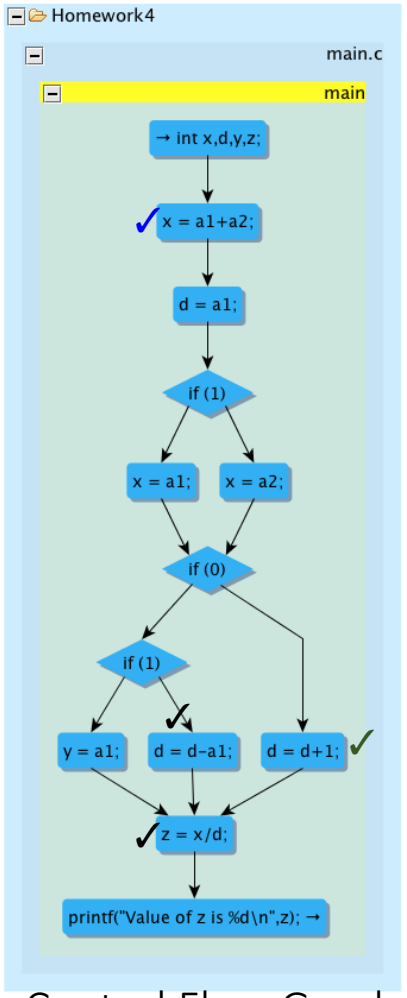


Control Flow Graph

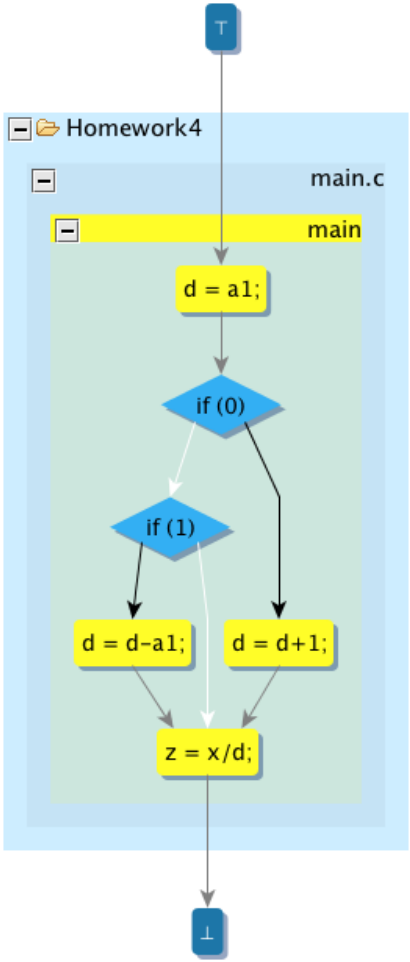
- Parameter: a set of four events – statements 5,17,21, and 25.
- Captures only the behaviors relevant to division-by-zero vulnerability at statement 25.

PCG: one path for each distinct behavior

```
4       x = a1+a2
5   ✓   d = a1;
6       if (C1) {
7           X = a1;
8       }
9       else {
10          X = a2;
11      }
12      if (C2) {
13          if (C3) {
14              Y = a1;
15          }
16          else {
17  ✓           d = d-a1;
18          }
19      }
20      else {
21  ✓       d = d+1;
22      }
23  ✓   Z = X/d;
```



Control Flow Graph



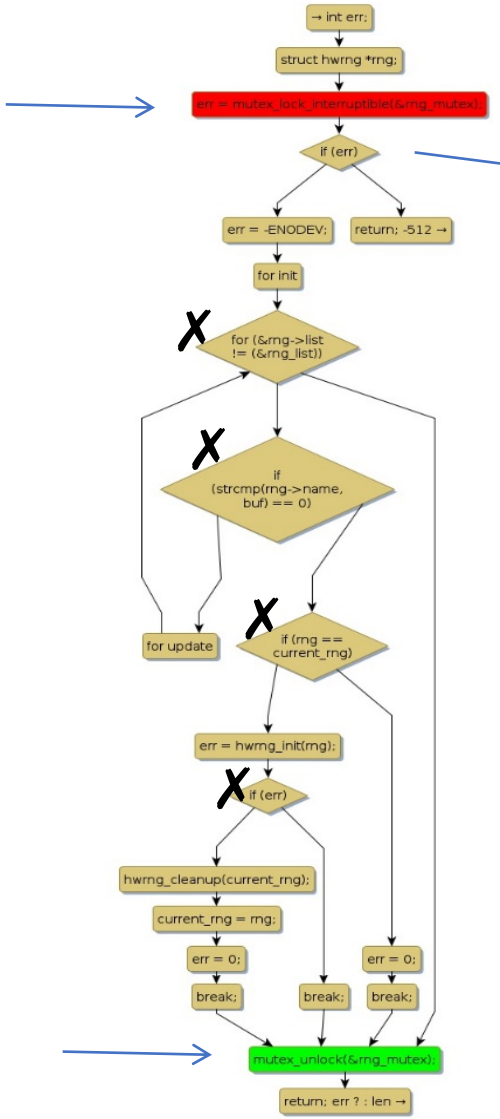
Parameterized Projected Control Graph (PCG)

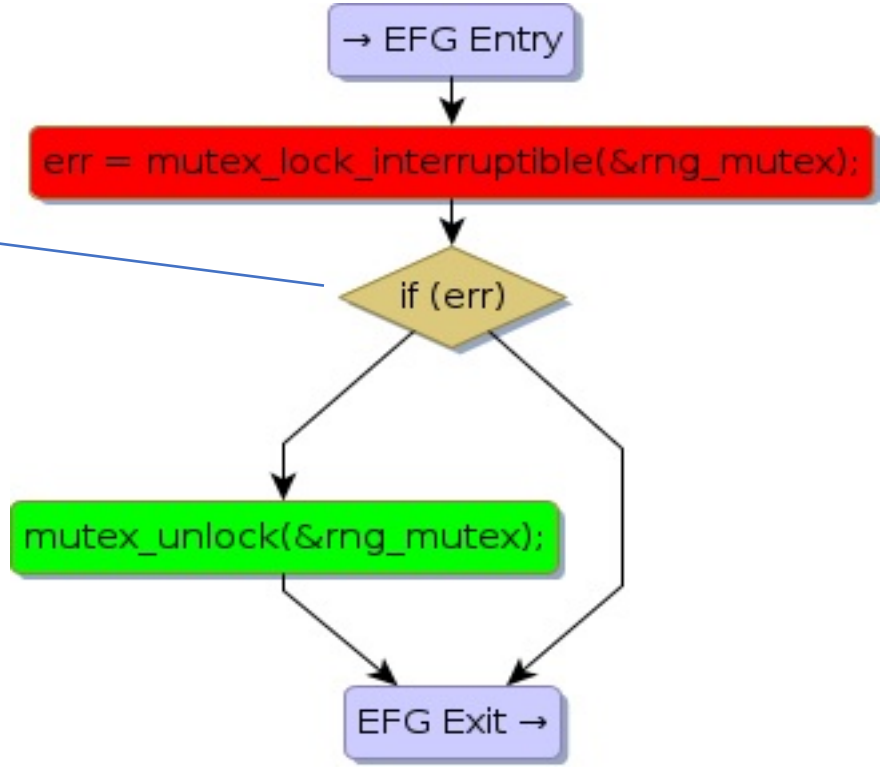
PCG as compact evidence

Function Name	Nodes		Edges	
	CFG	PCG	CFG	PCG
client_common_fill_super	1,101	15	1,179	28
kiblnd_create_conn	731	18	925	34
CopyBufferToControlPacket	392	20	559	39
kiblnd_cm_callback	662	38	831	56
kiblnd_passive_connect	622	22	784	44
dst_ca_ioctl	349	2	518	1
qib_make_ud_req	621	10	821	15
cfs_cpt_table_al	522	7	672	13
private_ioctl	569	16	732	24
vCommandTimer	490	47	623	75

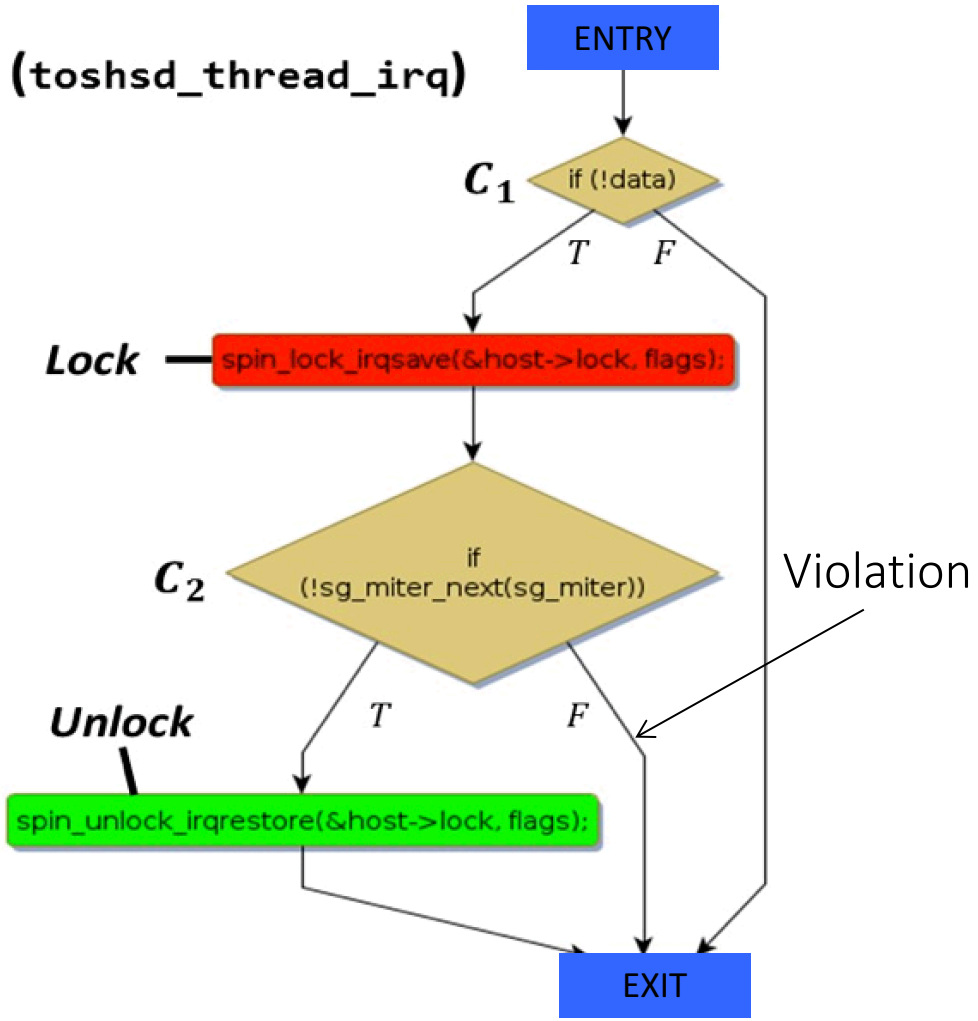
PCGs for Lock/UNLOCK pairing

A PCG example from Linux





Dissecting an unsafe instance missed by the LDV formal verification



LDV verdict: Inconclusive (category C3)

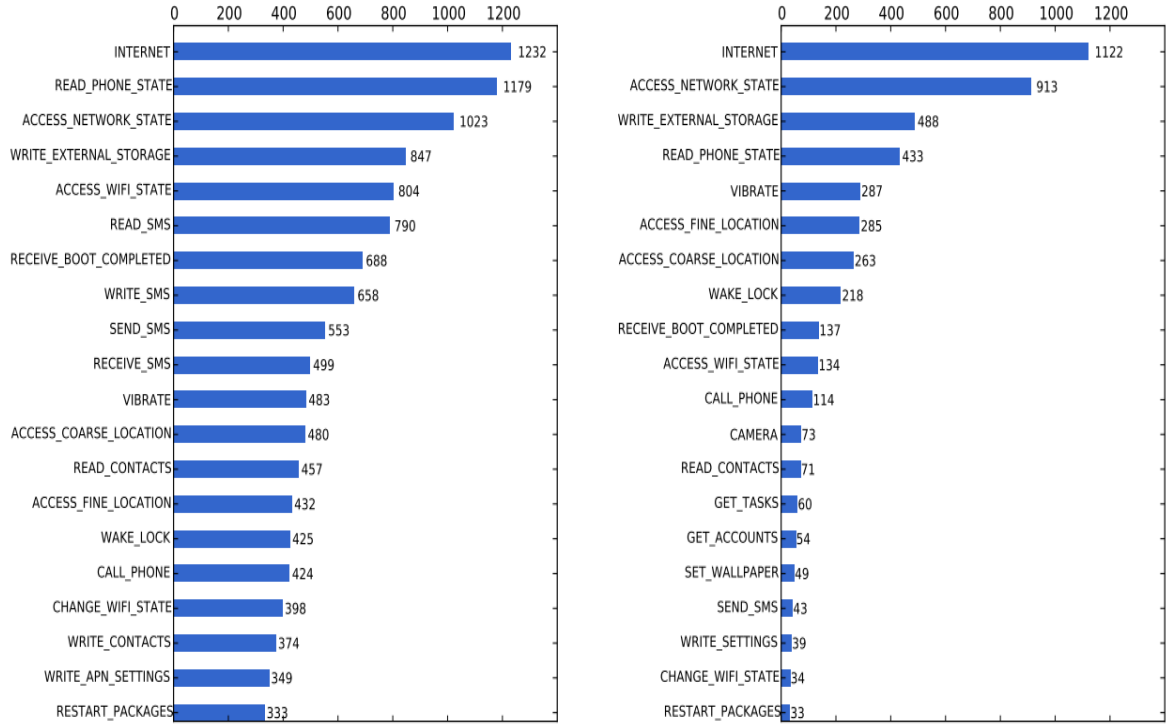
- LDV is inconclusive on this instance.
- The instance has **two challenges**: (a) *path feasibility check*, and (b) *path explosion*.
- Path explosion is suppressed in the figure by eliding spurious (not relevant to verification) branch nodes from the control flow graph to produce a compact graph that facilitates human comprehension.

Novel and Sophisticated Malware?

Finding “sophisticated” malware requires some thinking…

Case Studies in Android: Malware in the Wild

- Yajin Zhou, Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution.



(a) Top 20 Permissions Requested By 1260 Malware Samples

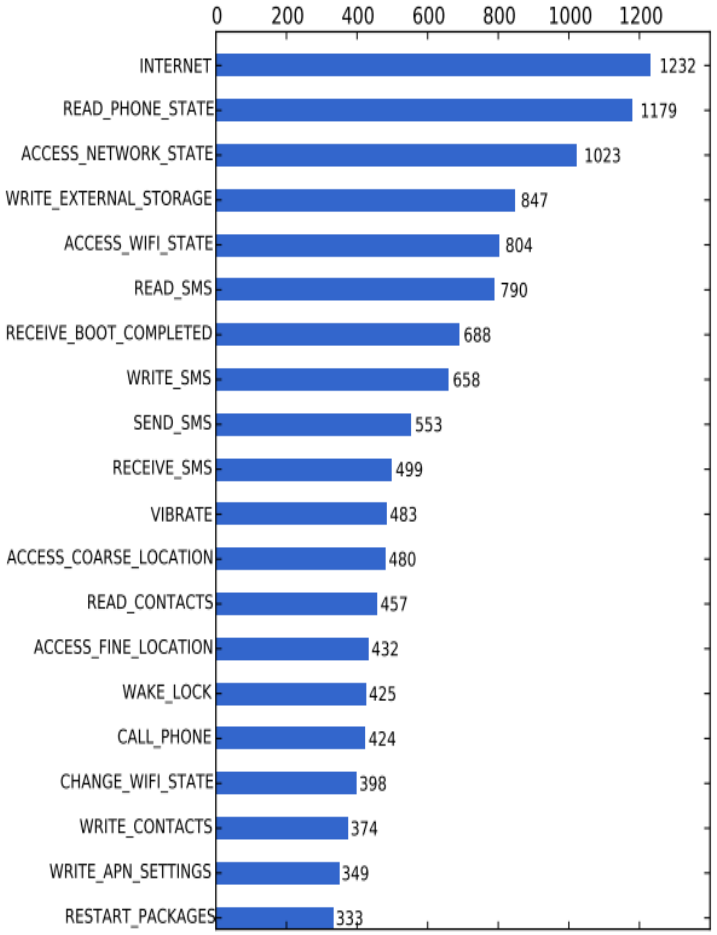
(b) Top 20 Permissions Requested by 1260 Top Free (Benign) Apps on the Offical Android Market

Figure 5. The Comparison of Top 20 Requested Permissions by Malicious and Benign Apps

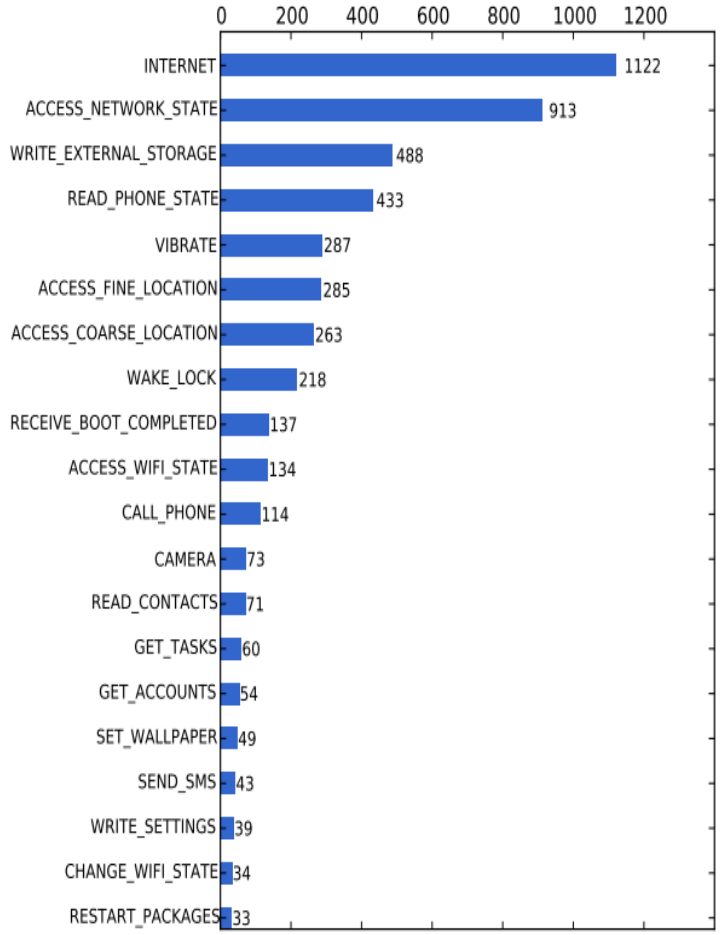
Table V
AN OVERVIEW OF EXISTING ANDROID MALWARE (PART II: MALICIOUS PAYLOADS)

	Privilege Escalation					Remote Control		Financial Charges			Personal Information Stealing		
	Exploid	RATC/ Zimperlich	Ginger Break	Asroot	Encrypted	NET	SMS	Phone Call	SMS	Block SMS	SMS	Phone Number	User Account
ADRD						√							
AnserverBot						√			√ †				
Asroot				√									
BaseBridge		√				√		√	√ †	√			
BeanBot						√		√	√ †	√		√	
BgServ						√			√ †	√		√	
CoinPirate						√			√ †	√	√		
Crusewin						√			√	√	√		
DogWars									√				
DroidCoupon		√				√							
DroidDeluxe		√											
DroidDream	√	√				√							
DroidDreamLight						√							√
DroidKungFu1	√	√			√	√						√	
DroidKungFu2	√	√			√	√						√	
DroidKungFu3	√	√			√	√						√	
DroidKungFu4						√							
DroidKungFu5	√	√			√	√						√	
DroidKungFuUpdate													
Endofday						√			√			√	
FakeNetflix													√
FakePlayer									√ ‡				
GamblerSMS											√		
Geinimi						√		√	√ †	√	√	√	
GGTracker									√ ‡	√	√	√	
GingerMaster			√			√						√	
GoldDream						√		√	√ †		√	√	
Gone60											√		
GPSSMSSpy									√				
HippoSMS									√ ‡	√			
Jifake									√ ‡				
jSMSHider						√			√ †	√		√	
KMin						√			√ †	√			
Lovetrap									√ †	√			
NickyBot							√		√		√		
Nickyspy						√			√		√		
Pjapps						√			√ †	√		√	
Plankton						√							
RogueLemon						√			√ †	√	√		
RogueSPPush									√ ‡	√			
SMSReplicator									√		√		
SndApps													√
Spitmo						√			√ †	√	√	√	
TapSnake													
Walkinwat									√				
YZHC						√			√ †	√		√	
zHash	√												
Zitmo											√		
Zsone									√ ‡	√			
number of families	*6*	*8*	*1*	*1*	*4*	*27*	*1*	*4*	*28*	*17*	*13*	*15*	*3*
number of samples	*389*	*440*	*4*	*8*	*363*	*1171*	*1*	*246*	*571*	*315*	*138*	*563*	*43*

Case Studies in Android: Malware in the Wild



(a) Top 20 Permissions Requested By 1260 Malware Samples



(b) Top 20 Permissions Requested by 1260 Top Free (Benign) Apps on the Offical Android Market

Figure 5. The Comparison of Top 20 Requested Permissions by Malicious and Benign Apps

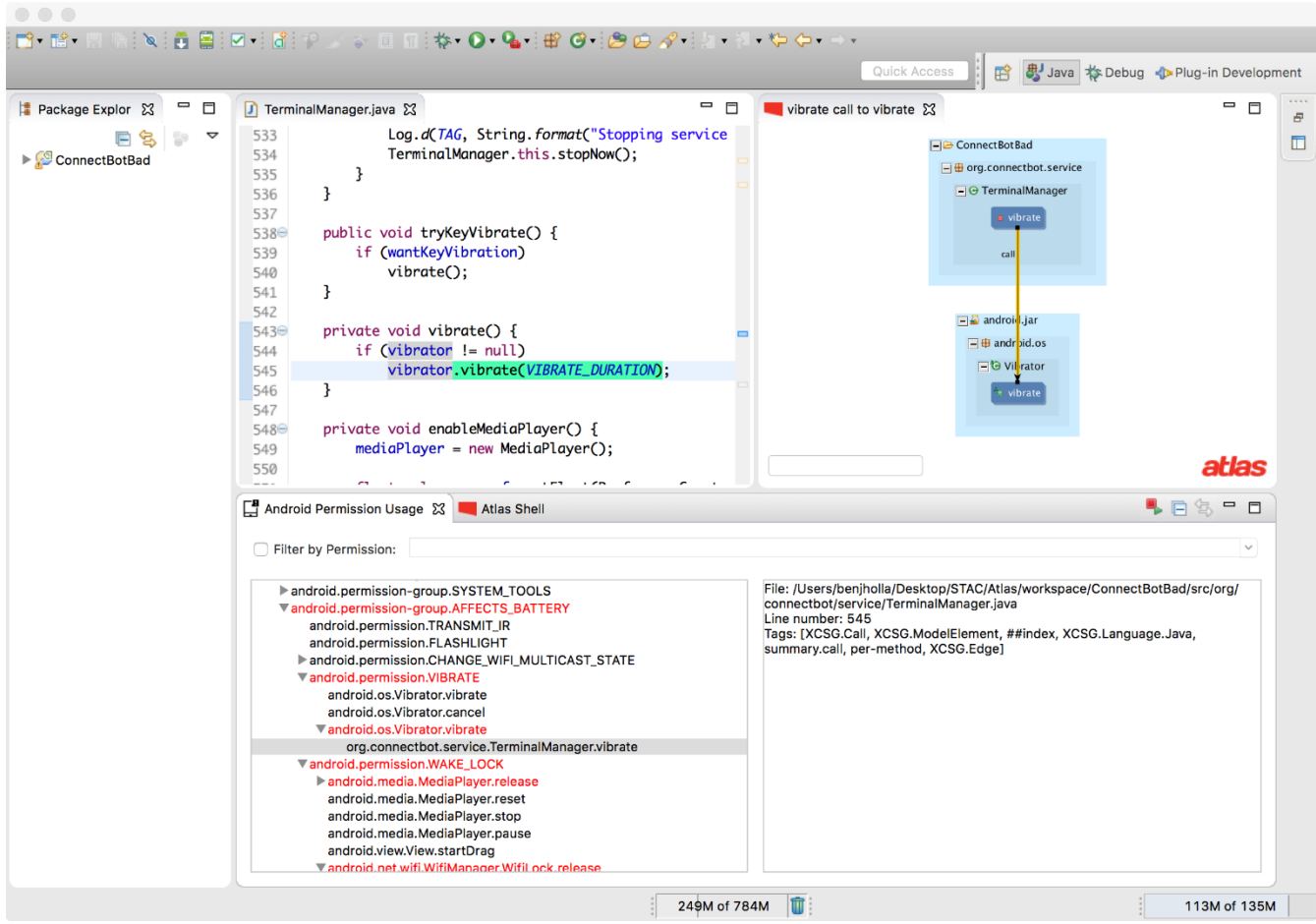
Case Studies in Android: Malware in the Wild

Table V
AN OVERVIEW OF EXISTING ANDROID MALWARE (PART II: MALICIOUS PAYLOADS)

	Privilege Escalation					Remote Control		Financial Charges			Personal Information Stealing		
	Exploid	RATC/ Zimperlich	Ginger Break	Asroot	Encrypted	NET	SMS	Phone Call	SMS	Block SMS	SMS	Phone Number	User Account
ADRD						√							
AnserverBot						√			√ †				
Asroot				√									
BaseBridge		√				√		√	√ †	√			
BeanBot						√		√	√ †	√		√	
BgServ						√			√ †	√		√	
CoinPirate						√			√ †	√	√		
Crusewin						√			√	√	√		
DogWars									√				
DroidCoupon		√				√							
DroidDeluxe		√											
DroidDream	√	√				√							
DroidDreamLight						√							√
DroidKungFu1	√	√			√	√						√	
DroidKungFu2	√	√			√	√						√	
DroidKungFu3	√	√			√	√						√	
DroidKungFu4						√							
DroidKungFu5	√	√			√	√						√	
DroidKungFuUpdate													
Endofday						√			√			√	
FakeNetflix													√
FakePlayer									√ ‡				
GamblerSMS											√		
Geinimi						√		√	√ †	√	√	√	
GGTracker									√ ‡	√	√	√	
GingerMaster			√			√						√	
GoldDream						√		√	√ †		√	√	
Gone60											√		
GPSSMSSpy									√				
HippoSMS									√ ‡	√			
Jifake									√ ‡				
jSMSHider						√			√ †	√		√	
KMin						√			√ †	√			
Lovetrap									√ †	√			
NickyBot							√		√		√		
Nickyspy						√			√		√		
Pjapps						√			√ †	√		√	
Plankton						√							
RogueLemon						√			√ †	√	√		
RogueSPPush									√ ‡	√			
SMSReplicator									√		√		
SndApps													√
Spitmo						√			√ †	√	√	√	
TapSnake													
Walkinwat									√				
YZHC						√			√ †	√		√	
zHash	√												
Zitmo											√		
Zsone									√ ‡	√			
number of families	*6*	*8*	*1*	*1*	*4*	*27*	*1*	*4*	*28*	*17*	*13*	*15*	*3*
number of samples	*389*	*440*	*4*	*8*	*363*	*1171*	*1*	*246*	*571*	*315*	*138*	*563*	*43*

Case Studies in Android: Permission Abuse

- Permission mapping?
  - 146 (documented) permissions as of Android API 19
- Android Essentials Toolbox
  - Maps permissions to permission protected methods (INTERNET, CAMERA, …)
  - Maps permission to permission level (normal, dangerous, root, …)
  - Maps permission to permission group (battery usage, location, …)



Case Studies in Android: Permission Abuse

- How do we know if an app is abusing permissions?
  - Context Matters
  - Thought Experiment: Imagine we have two identical applications…

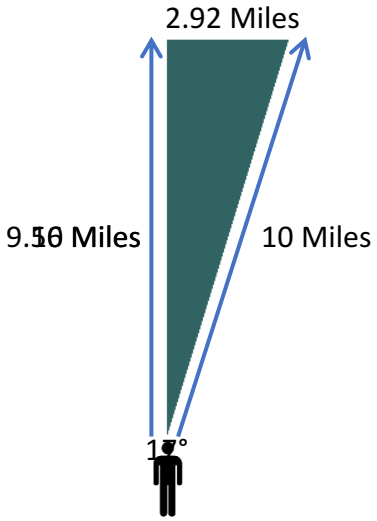
App 1: Alaskan Backpacker App

App 2: US Army Navigator App

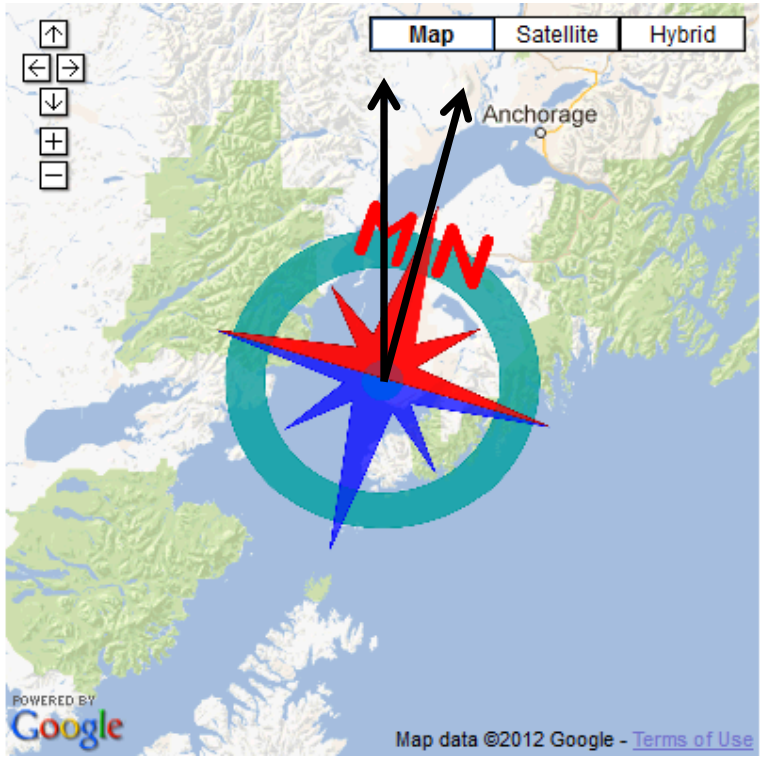
Behavior	App Purpose	Classification
Send location to Internet	Phone locator	Benign
Send location to Internet	Podcast player	Malicious
Selectively block SMS messages	Ad blocker	Benign
Selectively block SMS messages	Navigation	Malicious

Case Studies in Android: Permission Abuse

- App 1: Alaskan Backpacker App
  - Adjusts for magnetic declination so hikers don't get lost
  - In Alaska magnetic declination is 17° E



~3 Miles off course…

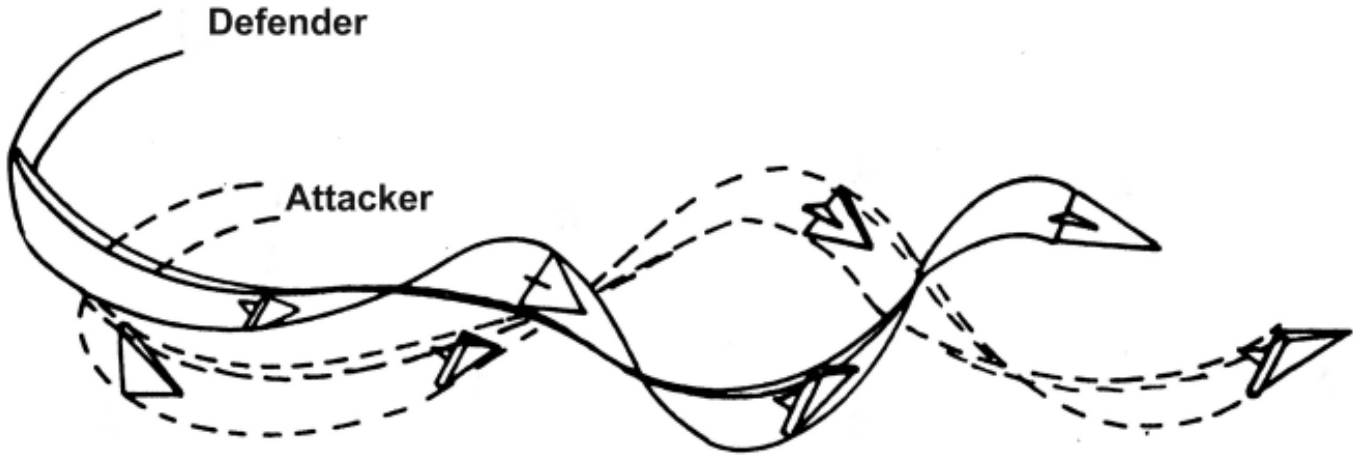


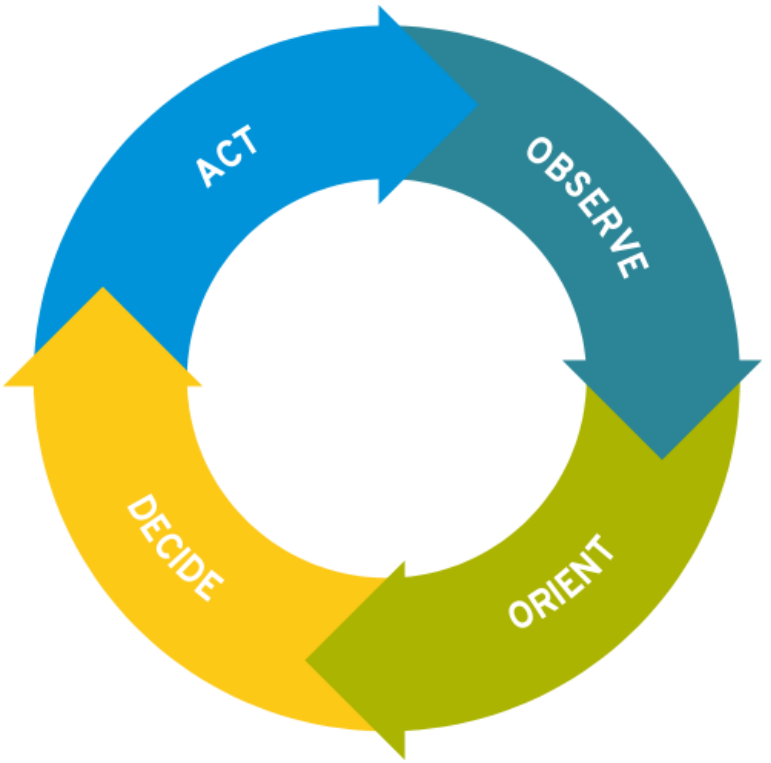
Case Studies in Android: Permission Abuse

- App 2: US Army Navigation App
  - Maliciously adds 17° to compass under certain conditions

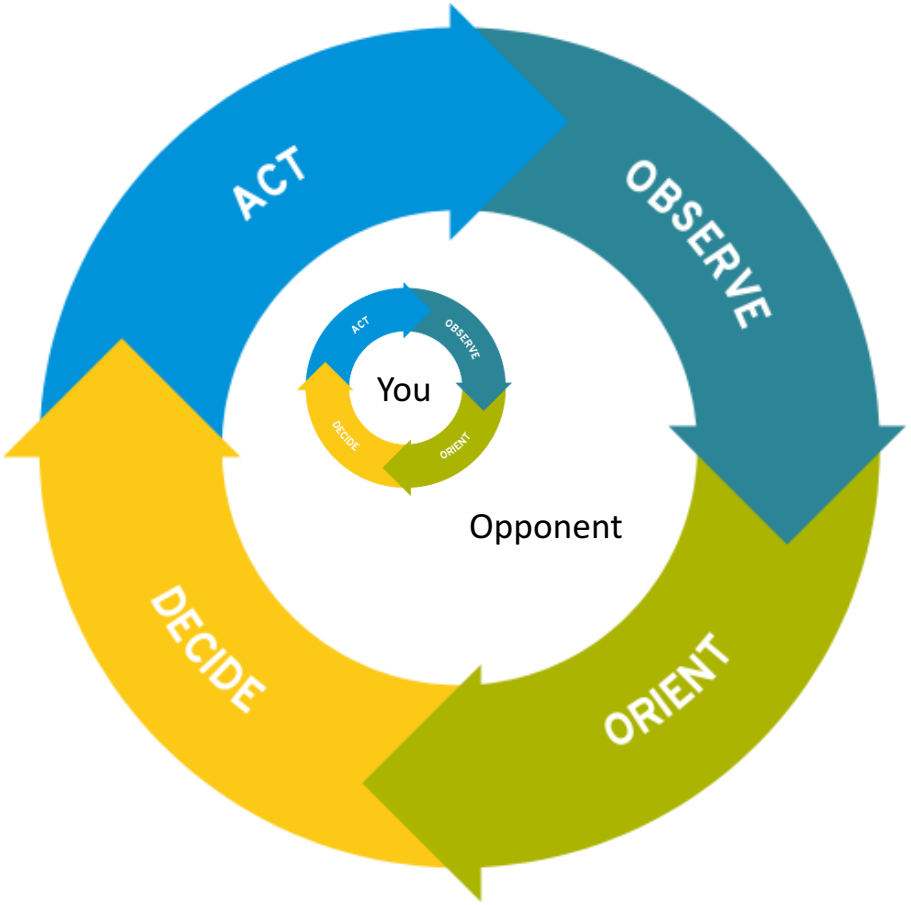
John Boyd's OODA Loop

"Security is a process, not a product" – Bruce Schneier





John Boyd's OODA Loop



Our opponent

- Time
- Evolution of malware

"...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." – Fred Brooks

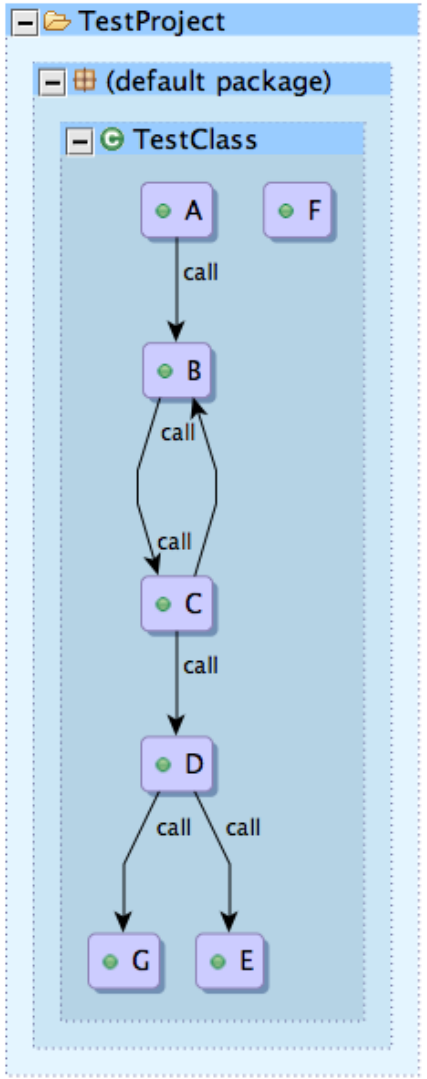
Speeding Through OODA (Man + Machine)

```
1  public class TestClass {
2
3      public void A() {
4          B();
5      }
6
7      public void B() {
8          C();
9      }
10
11     public void C() {
12         B();
13         D();
14     }
15
16     public void D() {
17         G();
18         E();
19     }
20
21     public void E() {
22
23     }
24
25     public void F() {
26
27     }
28
29     public void G(){
30
31     }
32
33 }
```

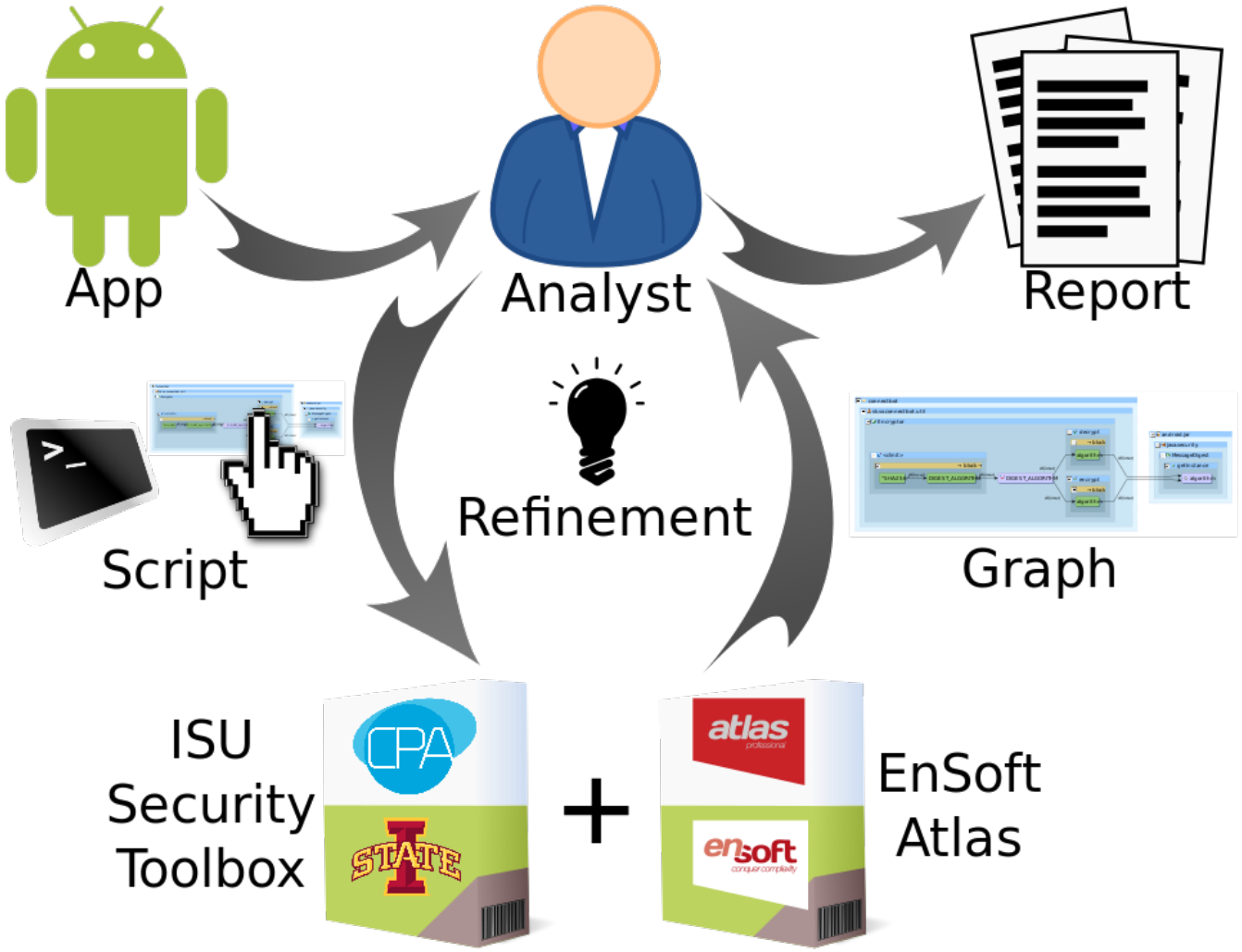
Program Declarations, Control Flow, and Data Flow

Queryable Graph Database

2-way Source Correspondence

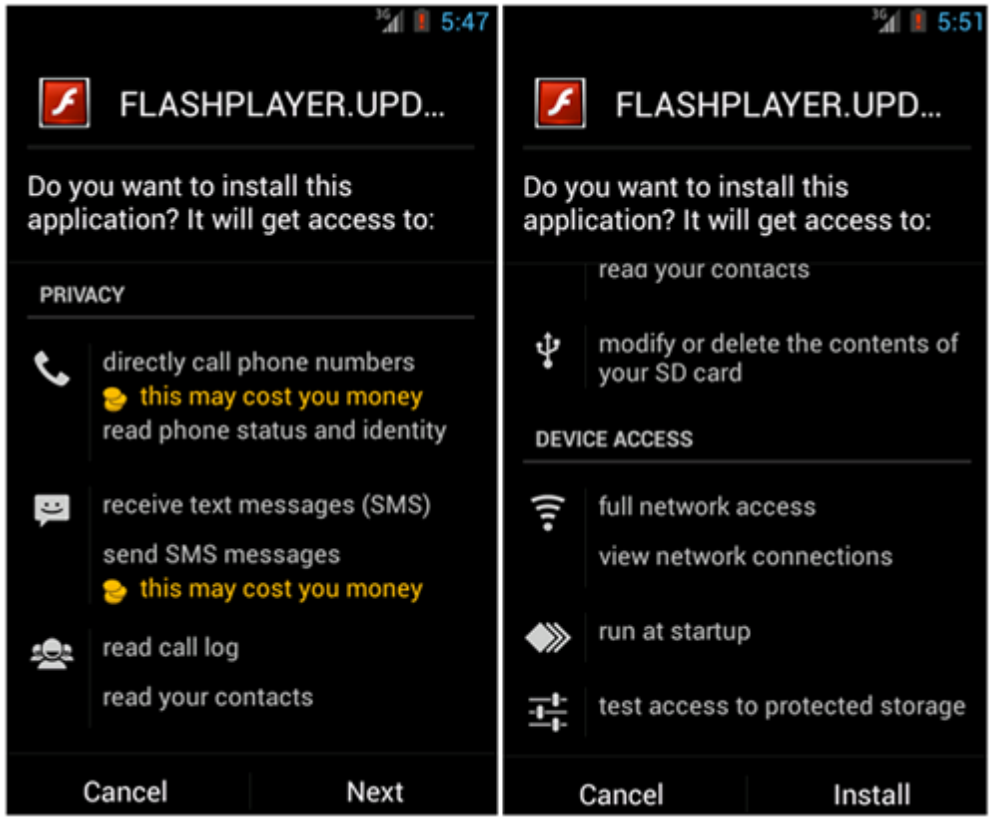


Speeding Through OODA (Man + Machine)



Case Studies in Android: Auditing Stels with Smells

- App Description: An email phishing campaign to install a fake “Adobe flash player update”
- Malware
  - Download and execute files
  - Steal contacts lists
  - Report system information
  - Make phone calls
  - Send SMS messages (to premium numbers)
  - Monitor and record and hide SMS messages
  - Show notifications
  - Uninstall apps



http://www.secureworks.com/cyber-threat-intelligence/threats/stels-android-trojan-malware-analysis/

Case Studies in Android: Auditing Battery Pro with Smart Views

- App Description: A battery charge state logging application. Application requests no permissions.
- Malware: Application leaks camera images over the internet.

Case Studies in Android: Auditing ConnectBotBad

- App Description: A telnet and SSH application for Android.
- Malware: Application captures remote login credentials and exfiltrates credentials to remote attacker machine.

Exercise: Let's Find it Together

Case Studies in Android: Timelapse of an Audit

- App Description: A network port scanner with the ability to write custom network scans and generate reports.
- Malware: The application contains a custom programming language and compiler for implementing network scan programs. A malicious binary is assembled in the custom language's VM memory from several binary blobs in the local database and is sent across the network to Windows hosts.

- Timelapse Audit: https://www.youtube.com/watch?v=p2mhfOMmgKI

Additional Resources

- https://www.ece.iastate.edu/kcsl/
- http://www.ensoftcorp.com/atlas/
- https://ensoftcorp.github.io/toolbox-repository/
- https://github.com/benjholla/MILCOM2017