Automated Detection of Software Bugs

Harikrishnan Balagopal

March 14, 2018

Contents

What is Program Analysis?

Program analysis is the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.

What is Program Analysis?

Program analysis is the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.

- Focuses on two major areas: program optimization and program correctness.

What is Program Analysis?

Program analysis is the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.

- Focuses on two major areas: program optimization and program correctness.
- Can be performed without executing the program (Static program analysis)

What is Program Analysis?

Program analysis is the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.

- Focuses on two major areas: program optimization and program correctness.
- Can be performed without executing the program (Static program analysis)
- Or during runtime (Dynamic program analysis)

What is Program Analysis?

Program analysis is the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.

- Focuses on two major areas: program optimization and program correctness.
- Can be performed without executing the program (Static program analysis)
- Or during runtime (Dynamic program analysis)
- Or in a combination of both.

Problems with Manual Testing

```
int f(int x)
{
        int y = x * 2;
        if(y == 12)fail(); // Crash
        else return y; // OK
}
```

Problems with Manual Testing

```
int f(int x)
{
        int y = x * 2;
        if(y == 12)fail(); // Crash
        else return y; // OK
}
```

- Checking 1 input tells us nothing about behaviour for other inputs. Thus we need to check all possible inputs.

Problems with Manual Testing

```
int f(int x)
{
        int y = x * 2;
        if(y == 12)fail(); // Crash
        else return y; // OK
}
```

- Checking 1 input tells us nothing about behaviour for other inputs. Thus we need to check all possible inputs.
- The number of possible inputs is astronomically large. Infeasible to check every single one.

Problems with Manual Testing

```
int f(int x)
{
        int y = x * 2;
        if(y == 12)fail(); // Crash
        else return y; // OK
}
```

- Checking 1 input tells us nothing about behaviour for other inputs. Thus we need to check all possible inputs.
- The number of possible inputs is astronomically large. Infeasible to check every single one.
- Testing every input is a waste of time and computational power since large sets of inputs will follow the same path through the program.

Problems with Manual Testing

```
int f(int x)
{
        int y = x * 2;
        if(y == 12)fail(); // Crash
        else return y; // OK
}
```

- Checking 1 input tells us nothing about behaviour for other inputs. Thus we need to check all possible inputs.
- The number of possible inputs is astronomically large. Infeasible to check every single one.
- Testing every input is a waste of time and computational power since large sets of inputs will follow the same path through the program.
- Test code could have bugs in it since its written by a human. Additional time and energy spent debugging test code.

Real world example: MINIX's tr tool

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                         10*
4 :     if (*arg == '\\') {                                  11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:          i++;
11:        } while (i<4 && *arg>='0' && *arg<='7');
12:        *buffer++ = ac;
13:      } else if (*arg != '\0')
14:        *buffer++ = *arg++;
15:    } else if (*arg == '[') {                             12*
16:      arg++;                                              13
17:      i = *arg++;                                         14
18:      if (*arg++ != '-') {                                15!
19:        *buffer++ = '[';
20:        arg -= 2;
21:        continue;
22:      }
23:      ac = *arg++;
24:      while (i <= ac) *buffer++ = i++;
25:      arg++;      /* Skip ']' */
26:    } else
27:      *buffer++ = *arg++;
28:  }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

tr(short for translate) is a UNIX tool that converts certain characters in a stream into other characters.

KLEE is a program analysis tool that uses symbolic execution.

When KLEE runs on tr, it finds a buffer overflow error at line 18 in just a few seconds and then produces a concrete input ([""" """) that hits the bug.

Why do Program Analysis?

Why do Program Analysis?

- Manual testing of programs is a very time consuming and labor intensive process. It is also prone to human error.

Motivation

Why do Program Analysis?

- Manual testing of programs is a very time consuming and labor intensive process. It is also prone to human error.
- Automated detection of bugs is preferable as it is **easier, faster, covers more of your code, and avoids bugs in the code doing the testing**.

Motivation

Why do Program Analysis?

- Manual testing of programs is a very time consuming and labor intensive process. It is also prone to human error.
- Automated detection of bugs is preferable as it is **easier, faster, covers more of your code, and avoids bugs in the code doing the testing**.
- Finds bugs that can compromise security or cause other undesirable behavior.

Motivation

Why do Program Analysis?

- Manual testing of programs is a very time consuming and labor intensive process. It is also prone to human error.
- Automated detection of bugs is preferable as it is **easier, faster, covers more of your code, and avoids bugs in the code doing the testing**.
- Finds bugs that can compromise security or cause other undesirable behavior.
- To prove your program does not contain certain types of bugs (invalid memory access, divide by zero, null pointer dereference, array index out of bounds, buffer overflow).

Why do Program Analysis?

- Manual testing of programs is a very time consuming and labor intensive process. It is also prone to human error.
- Automated detection of bugs is preferable as it is **easier, faster, covers more of your code, and avoids bugs in the code doing the testing**.
- Finds bugs that can compromise security or cause other undesirable behavior.
- To prove your program does not contain certain types of bugs (invalid memory access, divide by zero, null pointer dereference, array index out of bounds, buffer overflow).
- Compiler optimizations such as dead code elimination, constant propagation, common subexpression elimination etc rely on analysis such as control flow analysis, data flow analysis, etc.

Types of Program Analysis

- Static Analysis
  - Control Flow Analysis
  - Data Flow Analysis
  - Model Checking
  - Abstract Interpretation(Symbolic Execution)
  - Type Systems
- Dynamic Analysis
  - Testing
  - Monitoring
  - Program Slicing

Static Program Analysis

Static program analysis is the analysis of computer software that is performed without actually executing programs.

Static Program Analysis

Static program analysis is the analysis of computer software that is performed without actually executing programs.

- Extract information about the behavior of programs by systematic inspection of program text.

Static Program Analysis

Static program analysis is the analysis of computer software that is performed without actually executing programs.

- Extract information about the behavior of programs by systematic inspection of program text.
- Can be performed on some version of the source code or some form of the object code.

Static Program Analysis

Static program analysis is the analysis of computer software that is performed without actually executing programs.

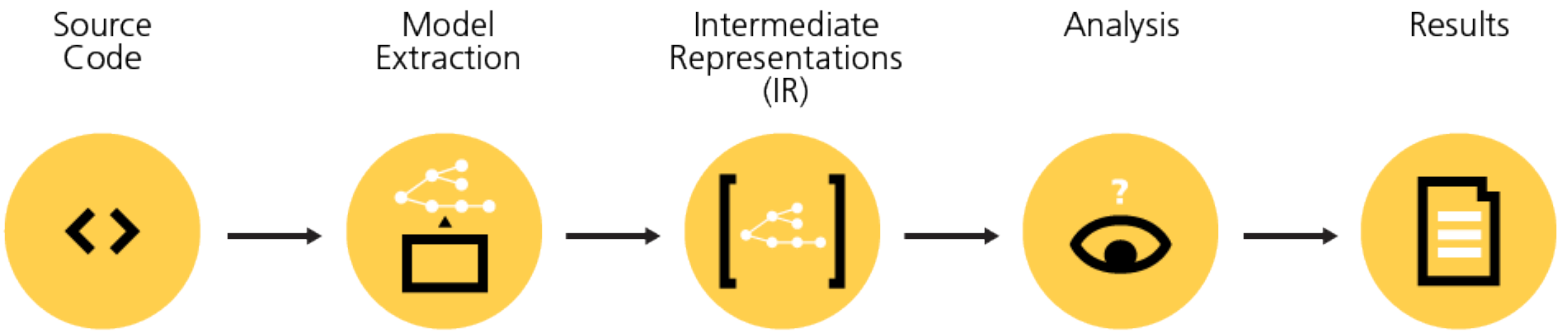
- Extract information about the behavior of programs by systematic inspection of program text.
- Can be performed on some version of the source code or some form of the object code.
- Usually performed by an automated tool

Static Program Analysis

Static program analysis is the analysis of computer software that is performed without actually executing programs.

- Extract information about the behavior of programs by systematic inspection of program text.
- Can be performed on some version of the source code or some form of the object code.
- Usually performed by an automated tool
- When done by a human it is called program understanding, program comprehension, or code review.

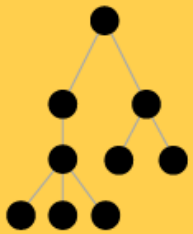
Static Analysis Diagram



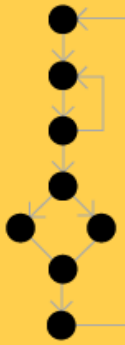
Names Databases/Symbol Table

Name	Kind	Location
copy_item	function	item. c:25
item_cache	variable	itemc:10
color	parameter	pallette.c:23
header.h	file	chapes.c

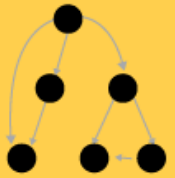
Abstract Syntax Tree (AST)



Control Flow graph (CFG)



Call Graph



Static Analysis Advantages and Disadvantages

Advantages:

- No need to build and execute the program.
- Can detect bugs early in development cycle.
- Cost efficient to correct bugs early.
- Can detect the exact location of bugs in the source code.
- Since it works on source code, it can detect bugs such as unreachable code, boundary value violations, out of bounds memory access etc.

Static Analysis Advantages and Disadvantages

Advantages:

- No need to build and execute the program.
- Can detect bugs early in development cycle.
- Cost efficient to correct bugs early.
- Can detect the exact location of bugs in the source code.
- Since it works on source code, it can detect bugs such as unreachable code, boundary value violations, out of bounds memory access etc.

Disadvantages:

- Time consuming if done manually.
- Automated tools produce false positives and false negatives.
- Does not find vulnerabilities introduced in the runtime environment.

Dynamic Analysis

Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor.

Dynamic Analysis

Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor.

- Analysis is done on the compiled binary.

Dynamic Analysis

Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor.

- Analysis is done on the compiled binary.
- The program is executed within a controlled environment and test inputs are used.

Dynamic Analysis

Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor.

- Analysis is done on the compiled binary.
- The program is executed within a controlled environment and test inputs are used.
- Instrumentation of the program binary is done to detect memory leaks, unsafe use of user input, monitor performance, etc.

Dynamic Analysis

Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor.

- Analysis is done on the compiled binary.
- The program is executed within a controlled environment and test inputs are used.
- Instrumentation of the program binary is done to detect memory leaks, unsafe use of user input, monitor performance, etc.
- Capable of exposing a subtle flaw or vulnerability too complicated for static analysis alone to reveal(race conditions, environment specific bugs).

Dynamic Analysis Advantages and Disadvantages

Advantages:

- Can find bugs that are specific to the runtime environment.
- Allows for analysis of applications without access to the source code.
- Can detect subtle bugs such as race conditions, memory leaks, etc that are hard to detect using static analysis.

Dynamic Analysis Advantages and Disadvantages

Advantages:

- Can find bugs that are specific to the runtime environment.
- Allows for analysis of applications without access to the source code.
- Can detect subtle bugs such as race conditions, memory leaks, etc that are hard to detect using static analysis.

Disadvantages:

- Need to build and run the program in order to do analysis.
- Cannot guarantee the full test coverage of the source code.
- It is more difficult to trace the vulnerability back to the exact location in the code, taking longer to fix the problem.

Symbolic Execution

Definition: "Symbolic execution (also symbolic evaluation) is a means of analyzing a program to determine what inputs cause each part of a program to execute.

An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch."

What is Symbolic Execution?

The workhorse of modern Program Analysis. It is a part of a Static Analysis technique called Abstract Interpretation.

What is Symbolic Execution?

The workhorse of modern Program Analysis. It is a part of a Static Analysis technique called Abstract Interpretation.

During normal execution of a program, variables have concrete values. (eg:- `int x, y; // becomes x = 4, y = 3`)

What is Symbolic Execution?

The workhorse of modern Program Analysis. It is a part of a Static Analysis technique called Abstract Interpretation.

During normal execution of a program, variables have concrete values. (eg:- `int x, y; // becomes x = 4, y = 3`)
During symbolic execution of a program, variables have symbolic values. (eg:- `int x, y; // assume x = X, y = Y`)

What is Symbolic Execution?

The workhorse of modern Program Analysis. It is a part of a Static Analysis technique called Abstract Interpretation.

During normal execution of a program, variables have concrete values. (eg:- `int x, y; // becomes x = 4, y = 3`)
During symbolic execution of a program, variables have symbolic values. (eg:- `int x, y; // assume x = X, y = Y`)
We proceed line by line through the code and add new constraints to these variables.

What is Symbolic Execution?

The workhorse of modern Program Analysis. It is a part of a Static Analysis technique called Abstract Interpretation.

During normal execution of a program, variables have concrete values. (eg:- `int x, y; // becomes x = 4, y = 3`)
During symbolic execution of a program, variables have symbolic values. (eg:- `int x, y; // assume x = X, y = Y`)
We proceed line by line through the code and add new constraints to these variables.
(eg:- `y = 2 * x; // x = X, y = 2X`)

What is Symbolic Execution?

The workhorse of modern Program Analysis. It is a part of a Static Analysis technique called Abstract Interpretation.

During normal execution of a program, variables have concrete values. (eg:- `int x, y; // becomes x = 4, y = 3`)
During symbolic execution of a program, variables have symbolic values. (eg:- `int x, y; // assume x = X, y = Y`)
We proceed line by line through the code and add new constraints to these variables.
(eg:- `y = 2 * x; // x = X, y = 2X`)
(eg:- `if(y > 4)x = y; else x = 4 * y;`

$$x = \begin{cases} 2X, & 2X > 4 \\ 8X, & 2X \leq 4 \end{cases}, y = 2X$$

)

What is Symbolic Execution?

The workhorse of modern Program Analysis. It is a part of a Static Analysis technique called Abstract Interpretation.

During normal execution of a program, variables have concrete values. (eg:- `int x, y; // becomes x = 4, y = 3`)
During symbolic execution of a program, variables have symbolic values. (eg:- `int x, y; // assume x = X, y = Y`)
We proceed line by line through the code and add new constraints to these variables.
(eg:- `y = 2 * x; // x = X, y = 2X`)
(eg:- `if(y > 4)x = y; else x = 4 * y;`

$$x = \begin{cases} 2X, & 2X > 4 \\ 8X, & 2X \leq 4 \end{cases}, y = 2X$$

)
These constraints allow us to consider ALL possible values those variables can take at the same time.

Standard execution

```
int max(int x, int y)
{
        int t = 0;
        if(x > y)
        {
                t = x;
        }
        else
        {
                t = y;
        }
        return t;
}
```

Standard execution

```
int max(int x, int y)
{
        int t = 0;
        if(x > y)
        {
                t = x;
        }
        else
        {
                t = y;
        }
        return t;
}
```

```
int max(int x = 2, int y = 4)
{
        int t = 0;
        if(2 > 4) // false
        {
                t = x;
        }
        else
        {
                t = 4;
        }
        return 4;
}
```

Standard execution

```
int max(int x, int y)
{
        int t = 0;
        if(x > y)
        {
                t = x;
        }
        else
        {
                t = y;
        }
        return t;
}
```

```
int max(int x = 4, int y = 2)
{
        int t = 0;
        if(4 > 2) // true
        {
                t = 4;
        }
        else
        {
                t = 2;
        }
        return 4;
}
```

Symbolic execution

```
int max(int x, int y)
{
        int t = 0;
        if(x > y)
        {
                t = x;
        }
        else
        {
                t = y;
        }
        return t;
}
```

```
int max(int x = X, int y = Y)
{
        int t = 0;
        if(X > Y)
        {
                t = X;
        }
        else
        {
                t = Y;
        }
        // now t = T
        return T;
}
```

$$T = \begin{cases} X, & X > Y \\ Y, & X \leq Y \end{cases}$$

Constraints

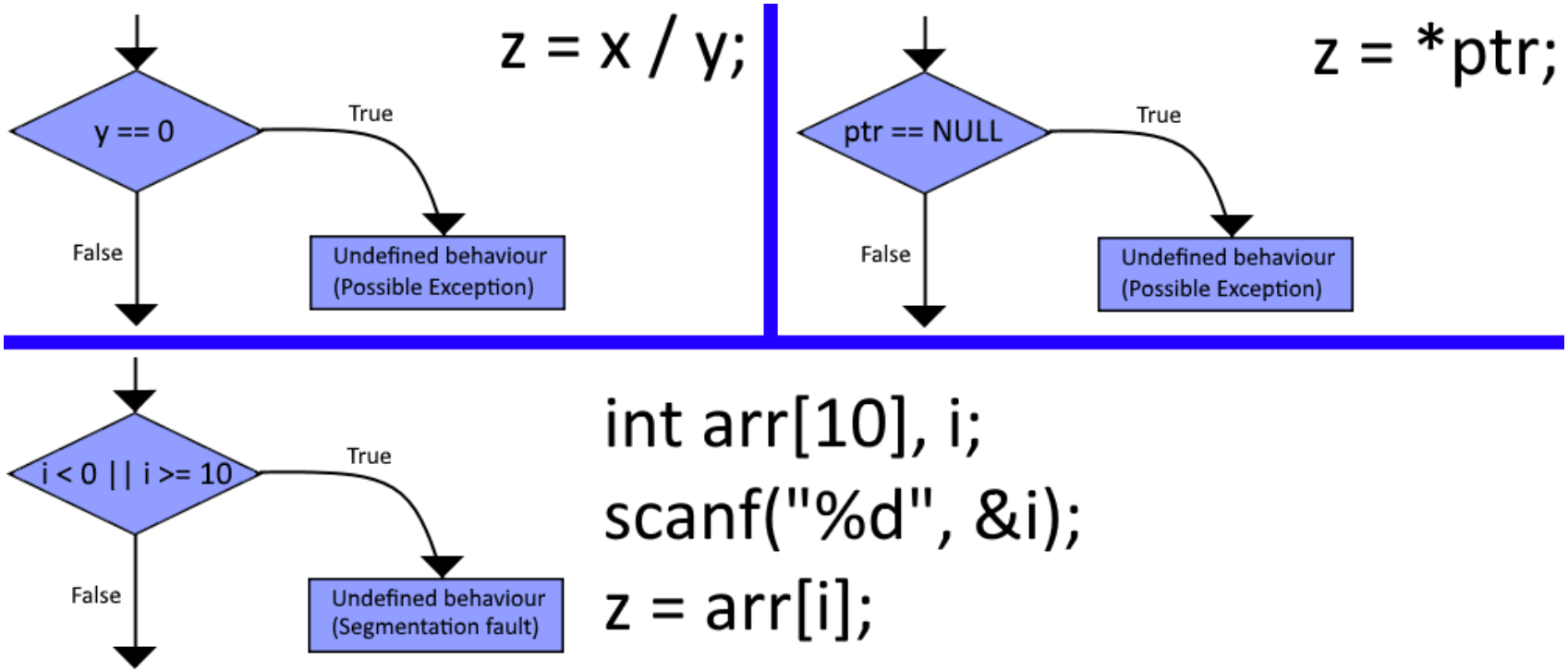
- Constraints generated from symbolic execution
  - $X > Y \implies T = X$
  - $X \leq Y \implies T = Y$
- After symbolic execution finishes, add an additional constraint "$T < X \vee \ T < Y \vee (T \neq X \wedge T \neq Y)$" to check correctness of the function max. This constraint represents a failure state.

$$(X > Y \implies T = X) \wedge$$
$$(X \leq Y \implies T = Y) \wedge$$
$$(T < X \vee \ T < Y \vee (T \neq X \wedge T \neq Y))$$

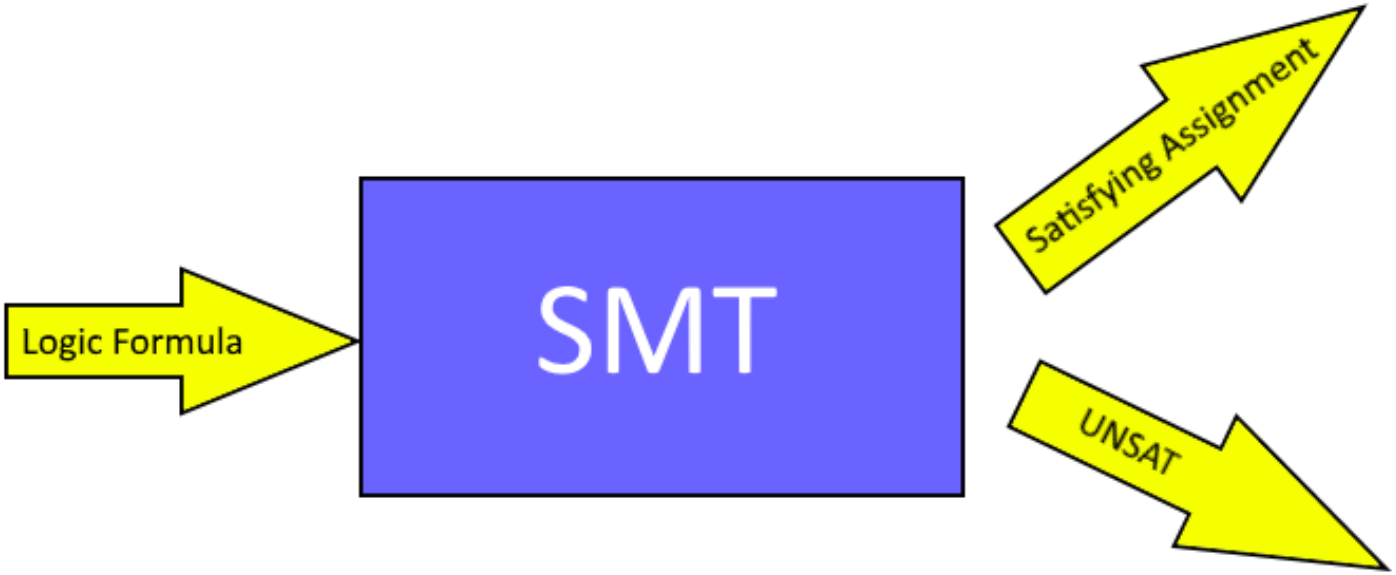
- If all these constraints can be satisfied at the same time then there is a bug. The solution to the constraints is a concrete input that reproduces the bug.
- In this example there is no solution to this set of constraints. Therefore the function is correct.

Branching Statements

"if" conditions are not the only points where execution can branch. Execution also branches when executing any code that could fail such as:



Solving Constraints



What is SMT?

Satisfiability Modulo Theories

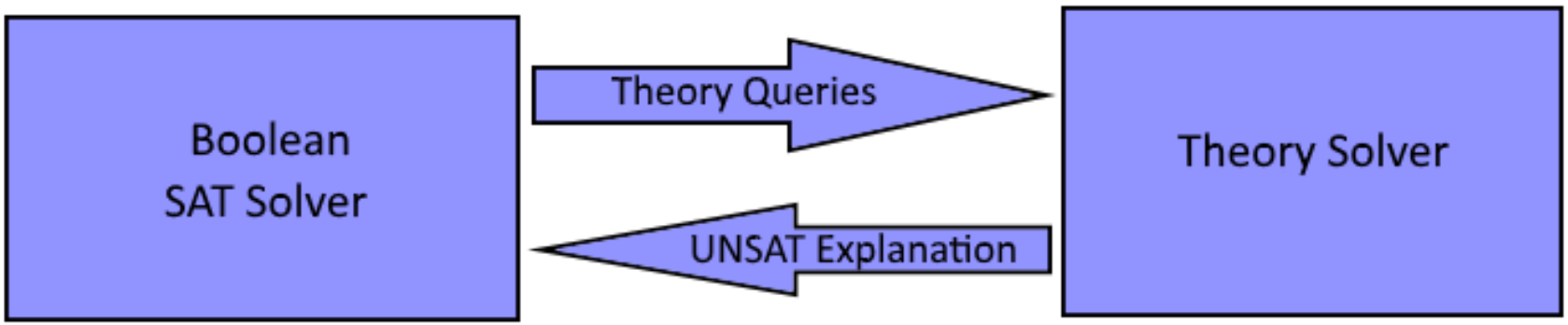
- Boolean SAT + Theories

Common Theories

- Bit Vectors (Fixed Length)
- Theory of Arrays
- Integer Arithmetic (Linear)
- Uninterpreted functions

SMT Working

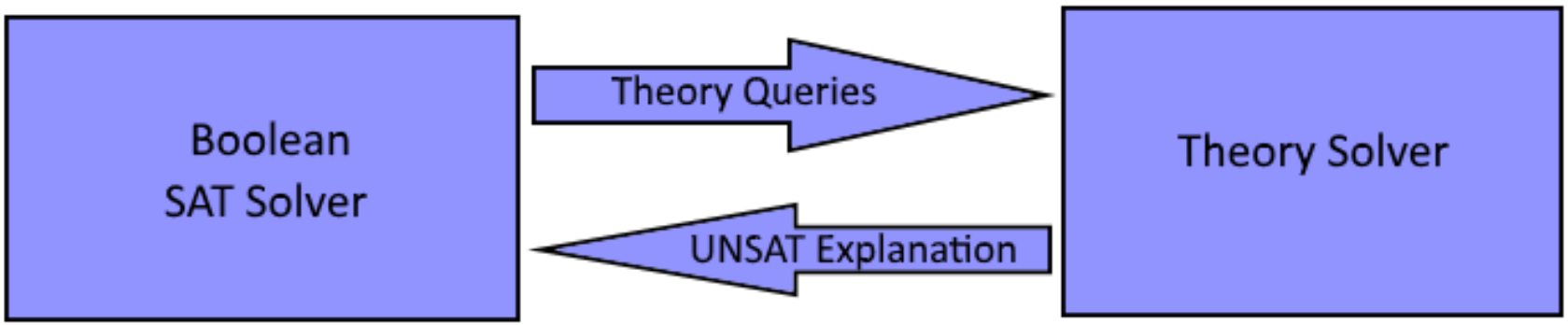
Constraints: $(X > 5) \wedge (Y < 5) \wedge ((Y > X) \vee (Y > 2))$



SMT Working

Constraints: $(X > 5) \wedge (Y < 5) \wedge ((Y > X) \vee (Y > 2))$
Simplify: $F1 \quad \wedge \quad F2 \quad \wedge \; (\; F3 \quad \vee \quad F4 \;)$

Give this pure boolean expression to the Boolean SAT solver.



SMT Working

Constraints: $(X > 5) \wedge (Y < 5) \wedge ((Y > X) \vee (Y > 2))$

Simplify: $F1 \quad \wedge \quad F2 \quad \wedge \; (\; F3 \quad \vee \quad F4 \;)$

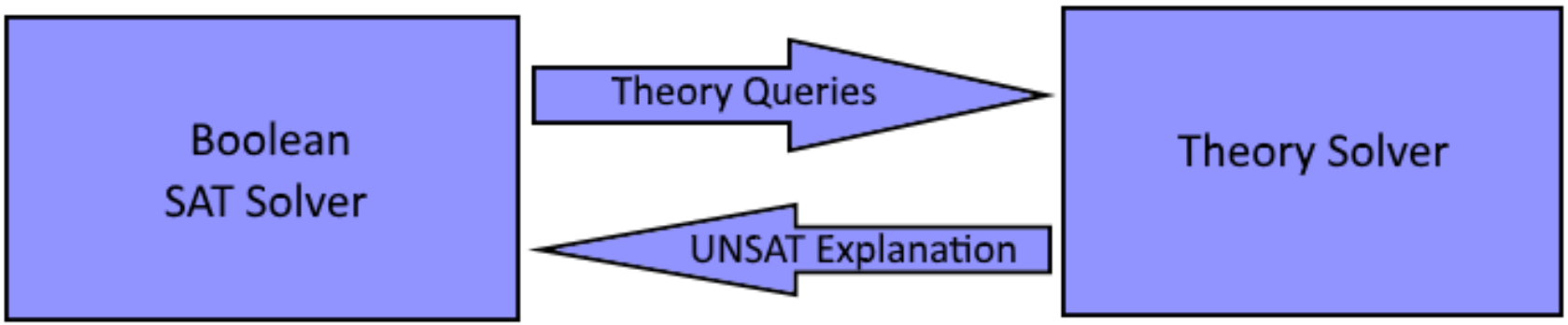
Give this pure boolean expression to the Boolean SAT solver.

Query 1:

$F1 \wedge F2 \wedge F3 : \; X > 5, Y < 5, Y > X$

Answer:

Unsatisfiable: $\neg(F1 \wedge F2 \wedge F3)$



SMT Working

Constraints: $(X > 5) \wedge (Y < 5) \wedge ((Y > X) \vee (Y > 2))$

Simplify: $F1 \wedge F2 \wedge (F3 \vee F4)$

$\wedge (\neg(F1 \wedge F2 \wedge F3))$

Give this pure boolean expression to the Boolean SAT solver.

Query 1:

$F1 \wedge F2 \wedge F3 : X > 5, Y < 5, Y > X$

Answer:

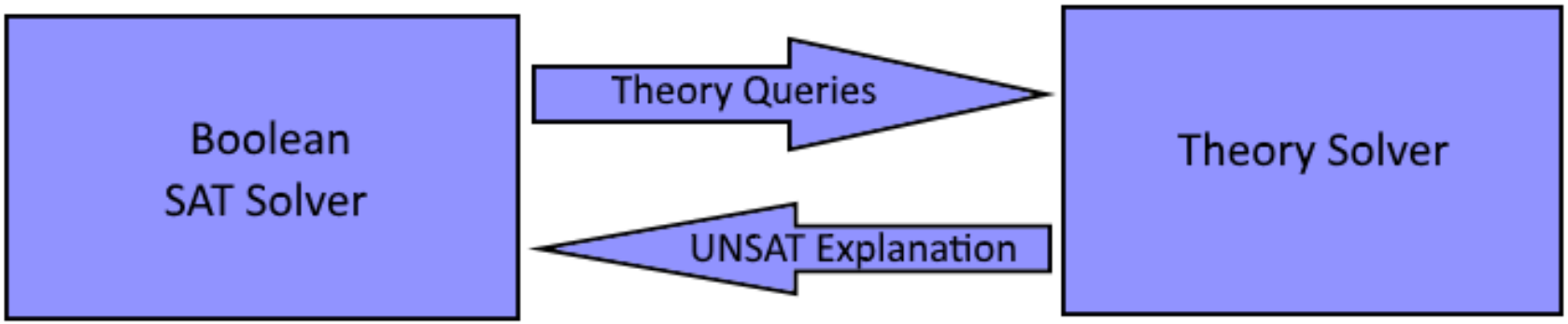
Unsatisfiable: $\neg(F1 \wedge F2 \wedge F3)$

Query 2:

F1 and F2 and F4: $X > 5, Y < 5, Y > 2$

Answer:

Satisfiable: $X = 6, Y = 3$



Concolic Execution

Also called Dynamic Symbolic Execution, combines concrete values with symbolic values to reduce the complexity of generated constraints in exchange for reduced code coverage.

Concolic Execution

Also called Dynamic Symbolic Execution, combines concrete values with symbolic values to reduce the complexity of generated constraints in exchange for reduced code coverage.

- Start with concrete input (could be random values).

Concolic Execution

Also called Dynamic Symbolic Execution, combines concrete values with symbolic values to reduce the complexity of generated constraints in exchange for reduced code coverage.

- Start with concrete input (could be random values).
- Execute the program normally but instrument the binary to keep track of the symbolic state on the side (shadow state).

Concolic Execution

Also called Dynamic Symbolic Execution, combines concrete values with symbolic values to reduce the complexity of generated constraints in exchange for reduced code coverage.

- Start with concrete input (could be random values).
- Execute the program normally but instrument the binary to keep track of the symbolic state on the side (shadow state).
- This shadow state keeps track of the constraints that must be satisfied to execute this path. These set of constraints are called the path condition.

Concolic Execution

Also called Dynamic Symbolic Execution, combines concrete values with symbolic values to reduce the complexity of generated constraints in exchange for reduced code coverage.

- Start with concrete input (could be random values).
- Execute the program normally but instrument the binary to keep track of the symbolic state on the side (shadow state).
- This shadow state keeps track of the constraints that must be satisfied to execute this path. These set of constraints are called the path condition.
- After execution is finished explore new paths by negating randomly chosen constraints in the path condition and solving for this new set of constraints to get a concrete input.

Concolic Execution

Also called Dynamic Symbolic Execution, combines concrete values with symbolic values to reduce the complexity of generated constraints in exchange for reduced code coverage.

- Start with concrete input (could be random values).
- Execute the program normally but instrument the binary to keep track of the symbolic state on the side (shadow state).
- This shadow state keeps track of the constraints that must be satisfied to execute this path. These set of constraints are called the path condition.
- After execution is finished explore new paths by negating randomly chosen constraints in the path condition and solving for this new set of constraints to get a concrete input.

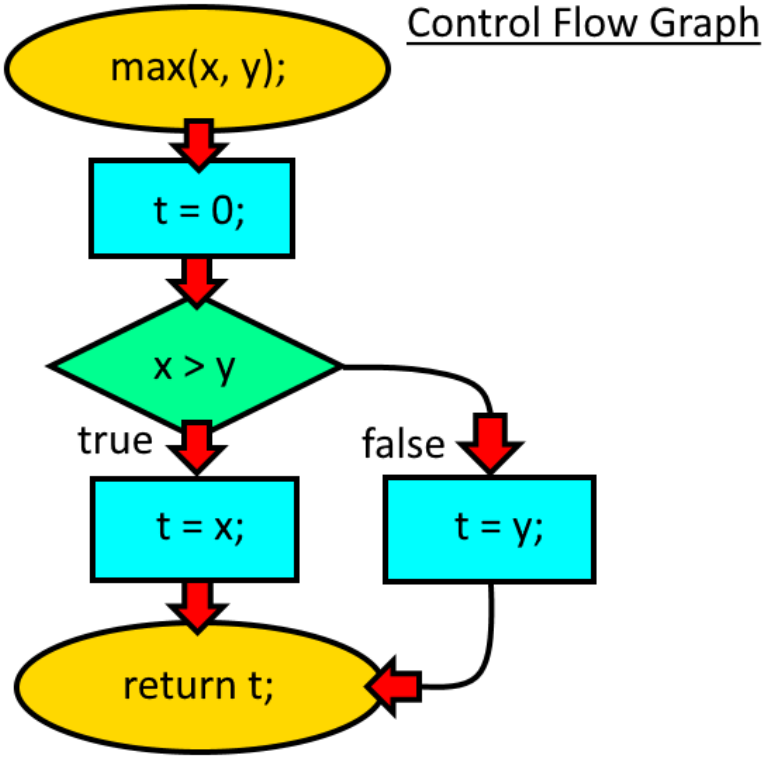
This technique reduces the complexity of constraints that are generated since we are not exploring every path at once.

Concolic Execution Example

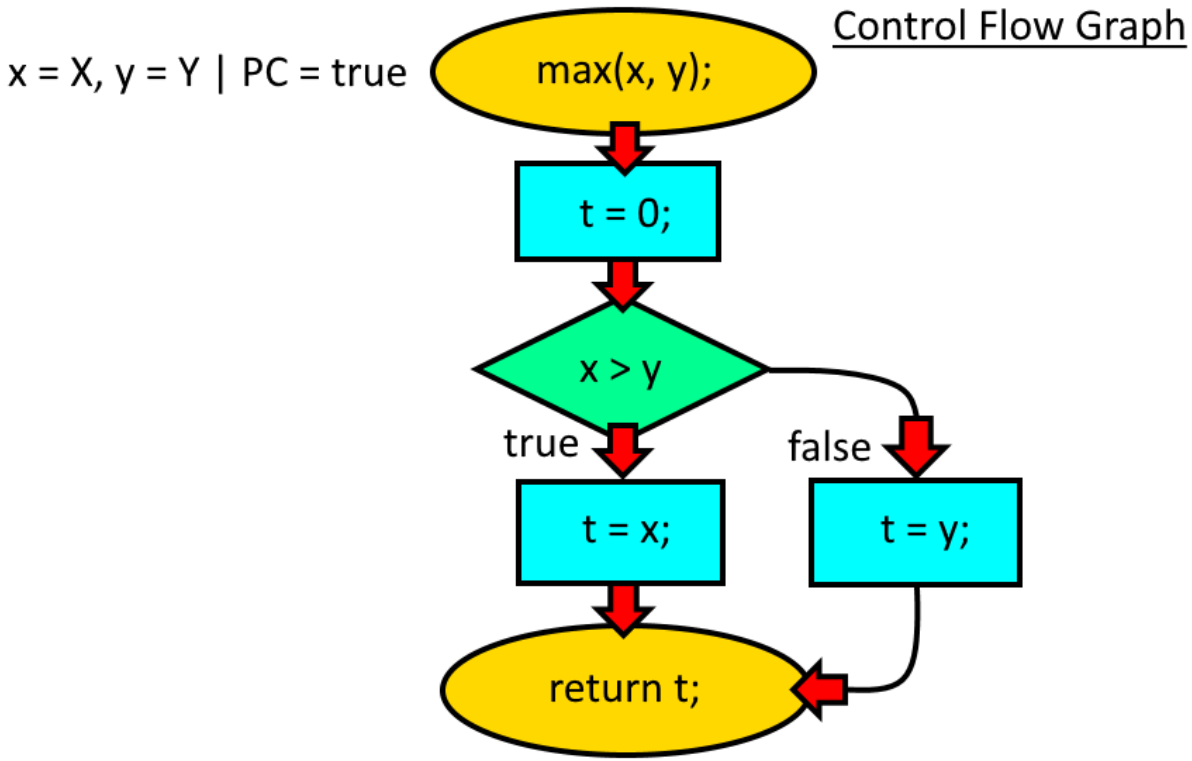
Program

```
int max(int x, int y)
{
    int t = 0;
    if(x > y)
    {
        t = x;
    }
    else
    {
        t = y;
    }
    return t;
}
```

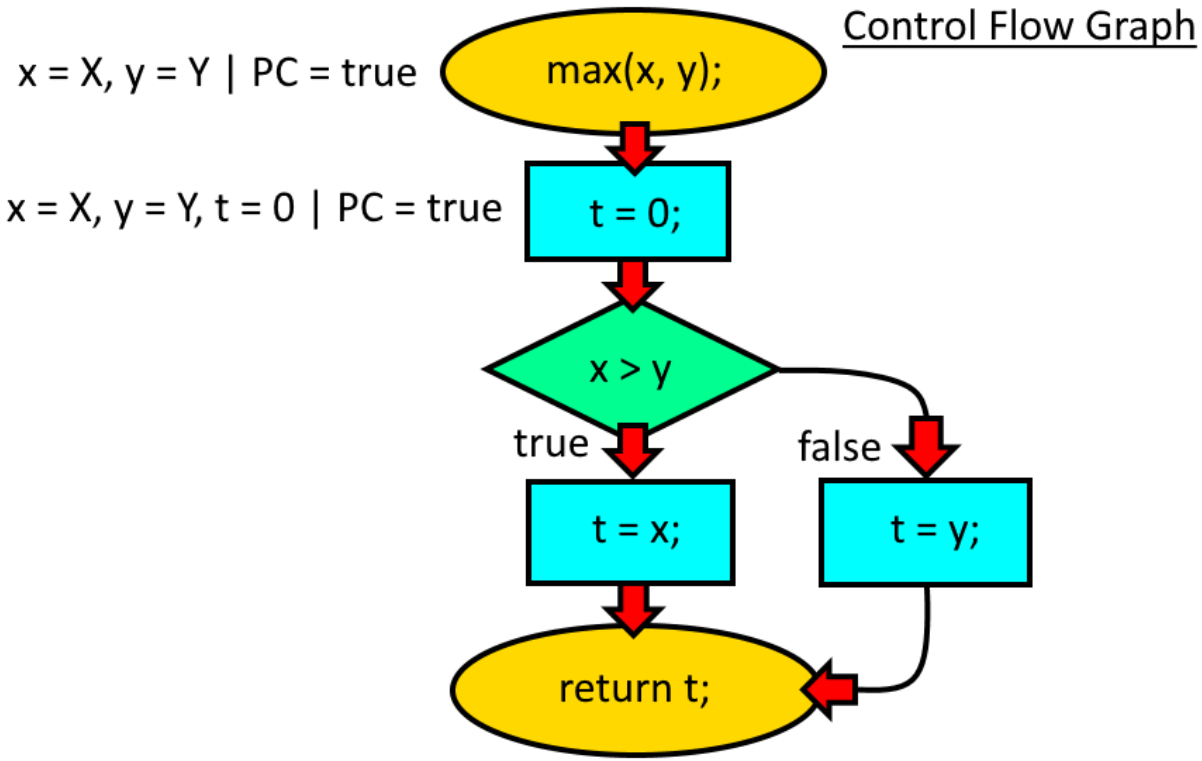
Control Flow Graph



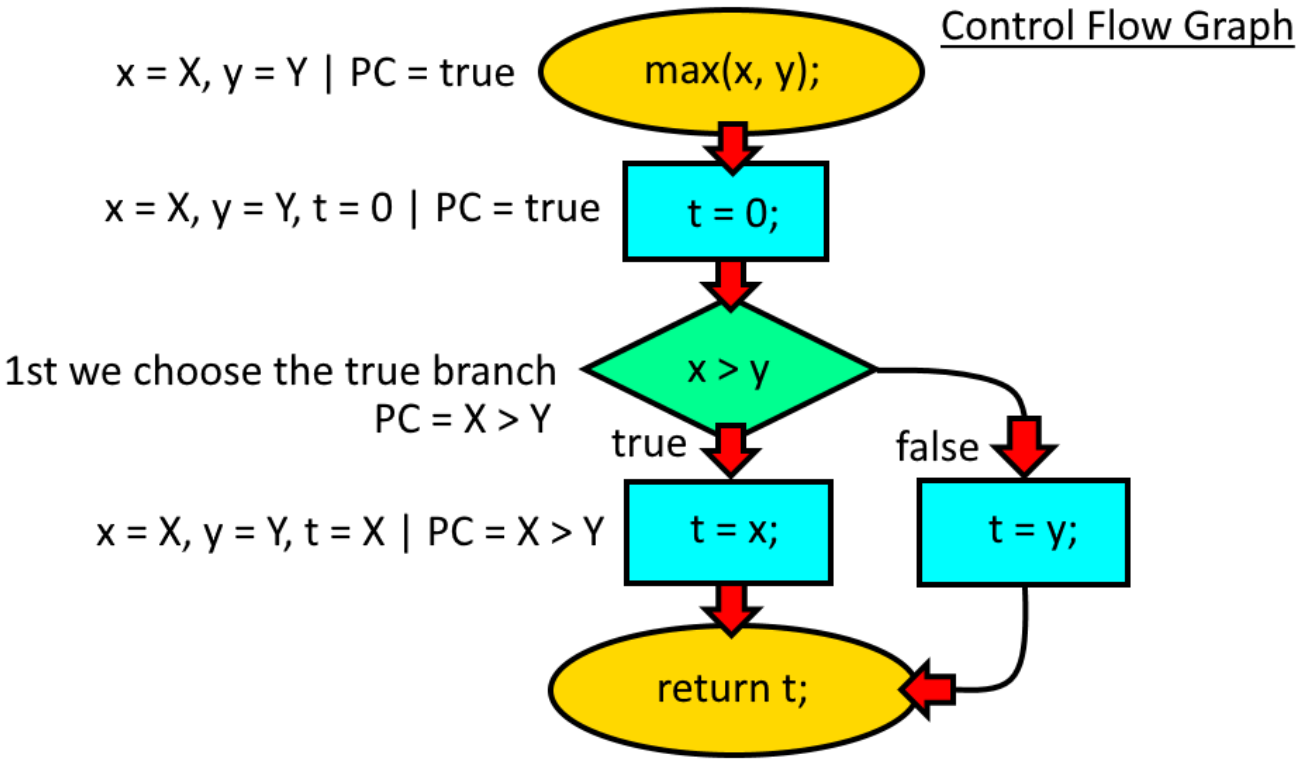
Concolic Execution Example



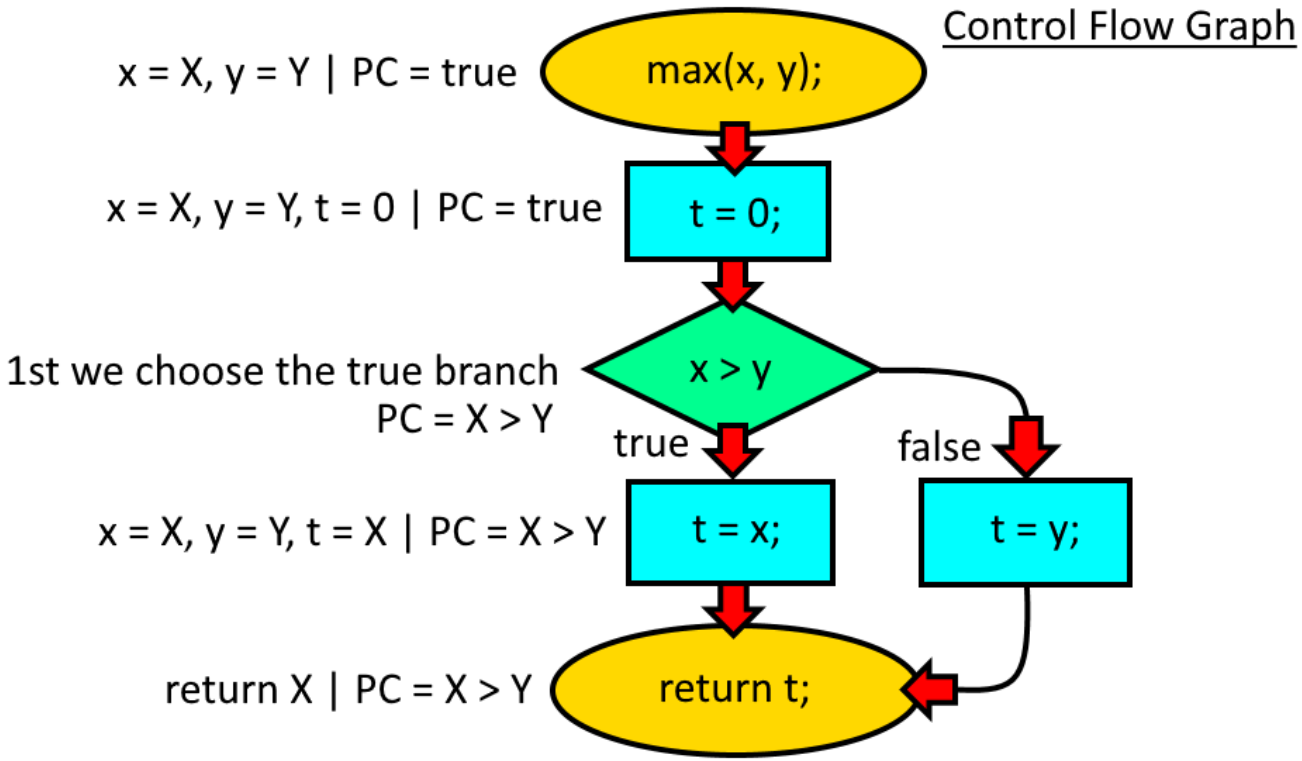
Concolic Execution Example



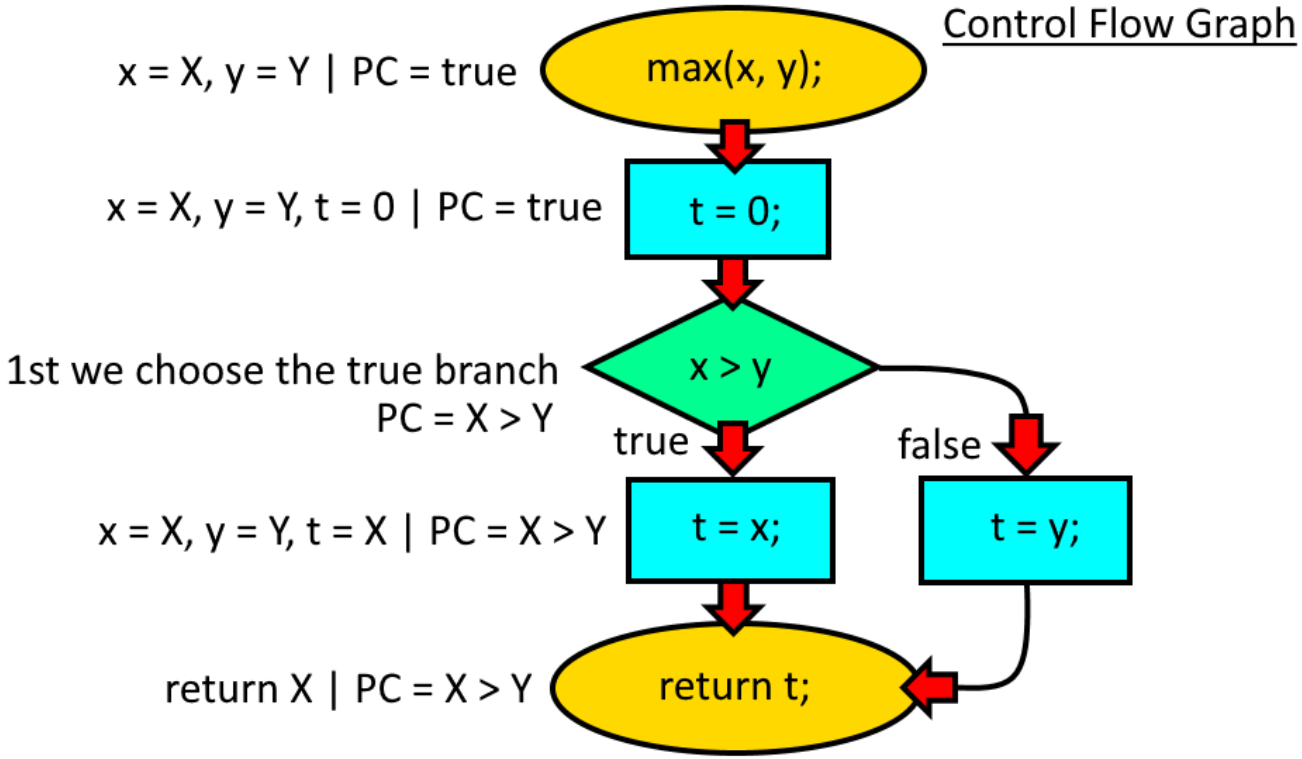
Concolic Execution Example



Concolic Execution Example

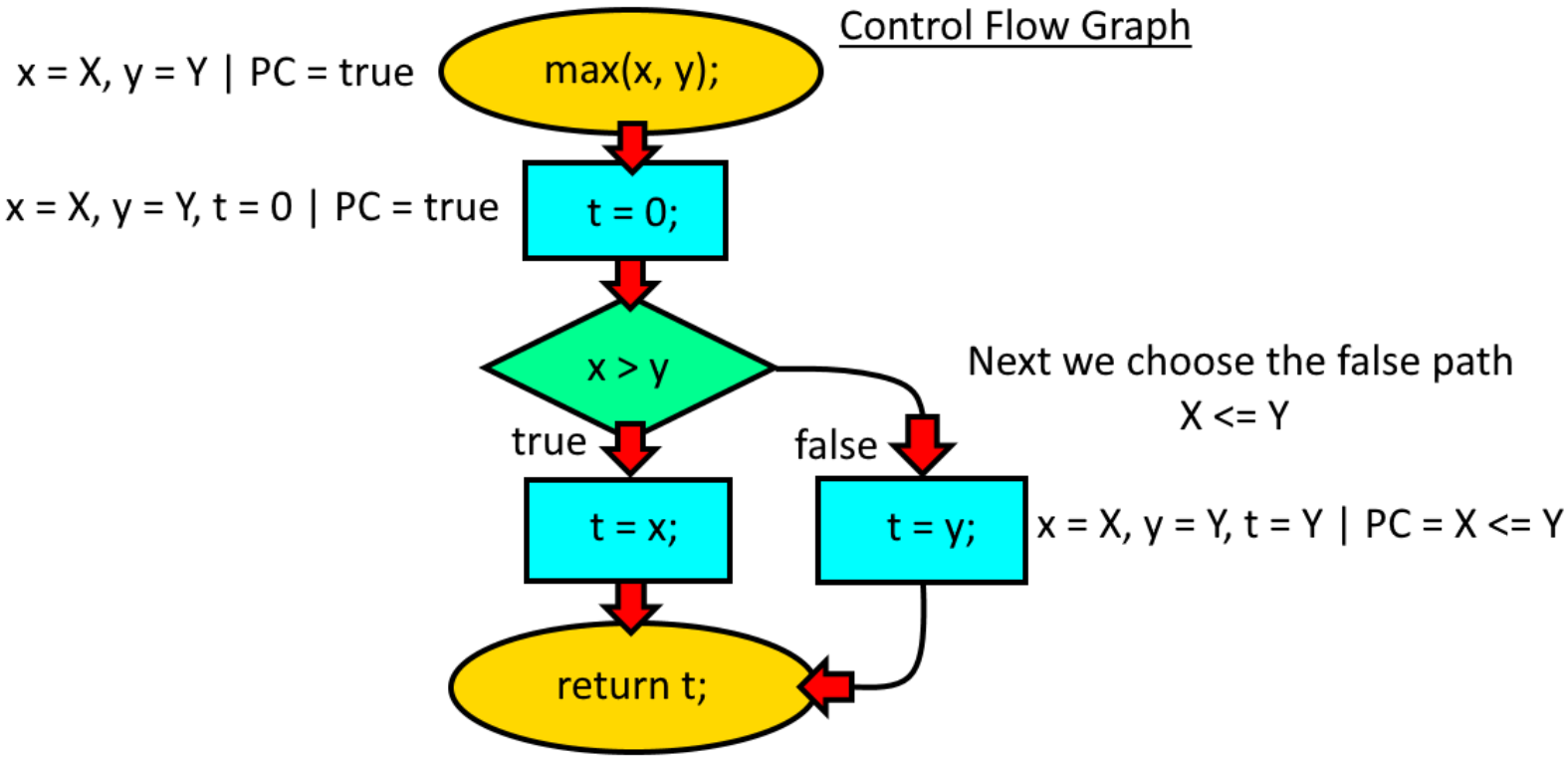


Concolic Execution Example

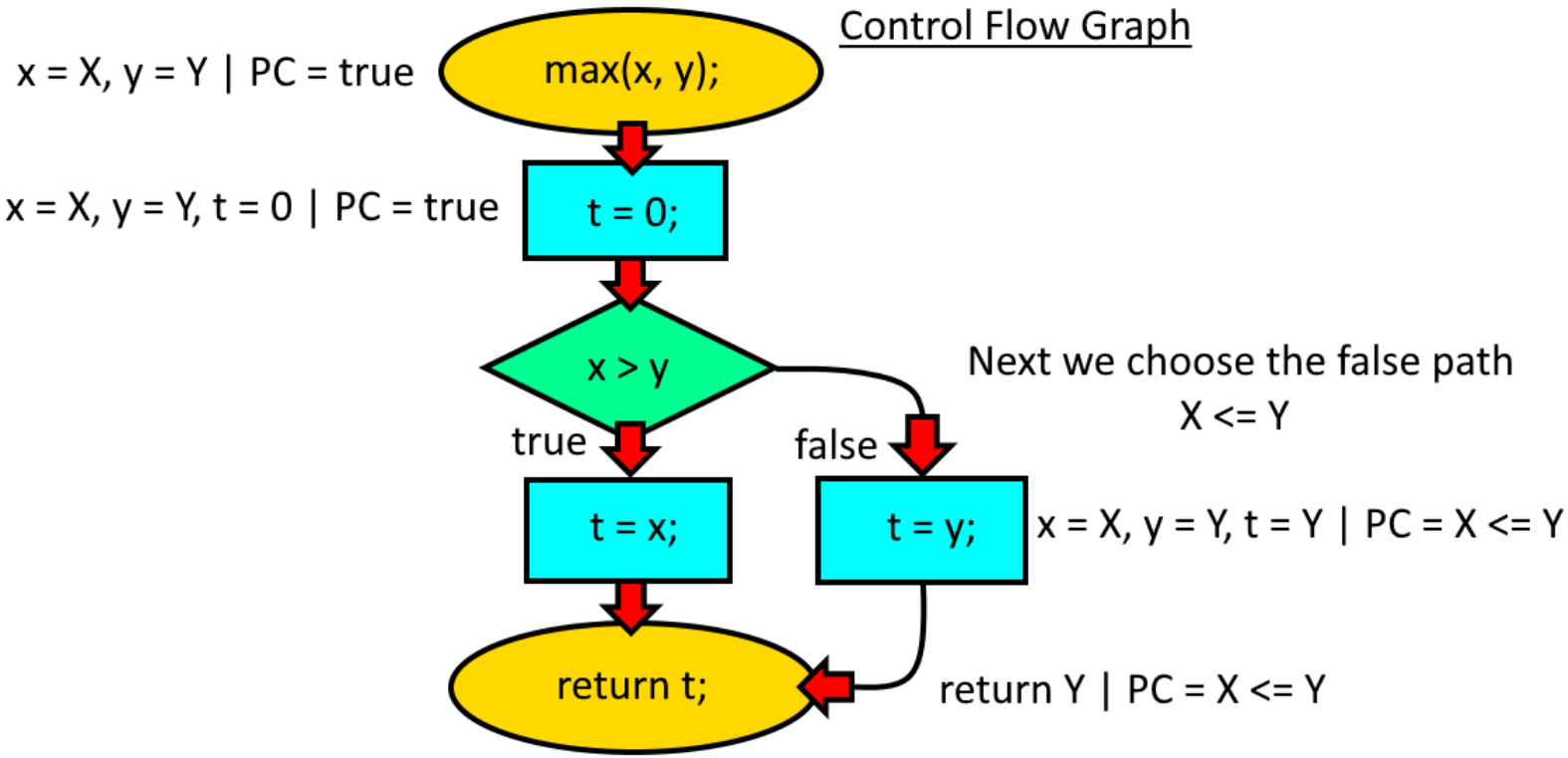


Now we can add the failure condition to the PC to see whether the function can fail:
Final Constraints: (X > Y) and ((t < X) or (t < Y) or (t =/= X and t =/= Y))
= (X > Y) and ((X < X) or (X < Y) or (X =/= X and X =/= Y))
= (X > Y) and ((false) or (X < Y) or (false and X =/= Y))
= (X > Y) and (X < Y)
= false (Unsatisfiable)
So we find that there is no bug along this execution path.

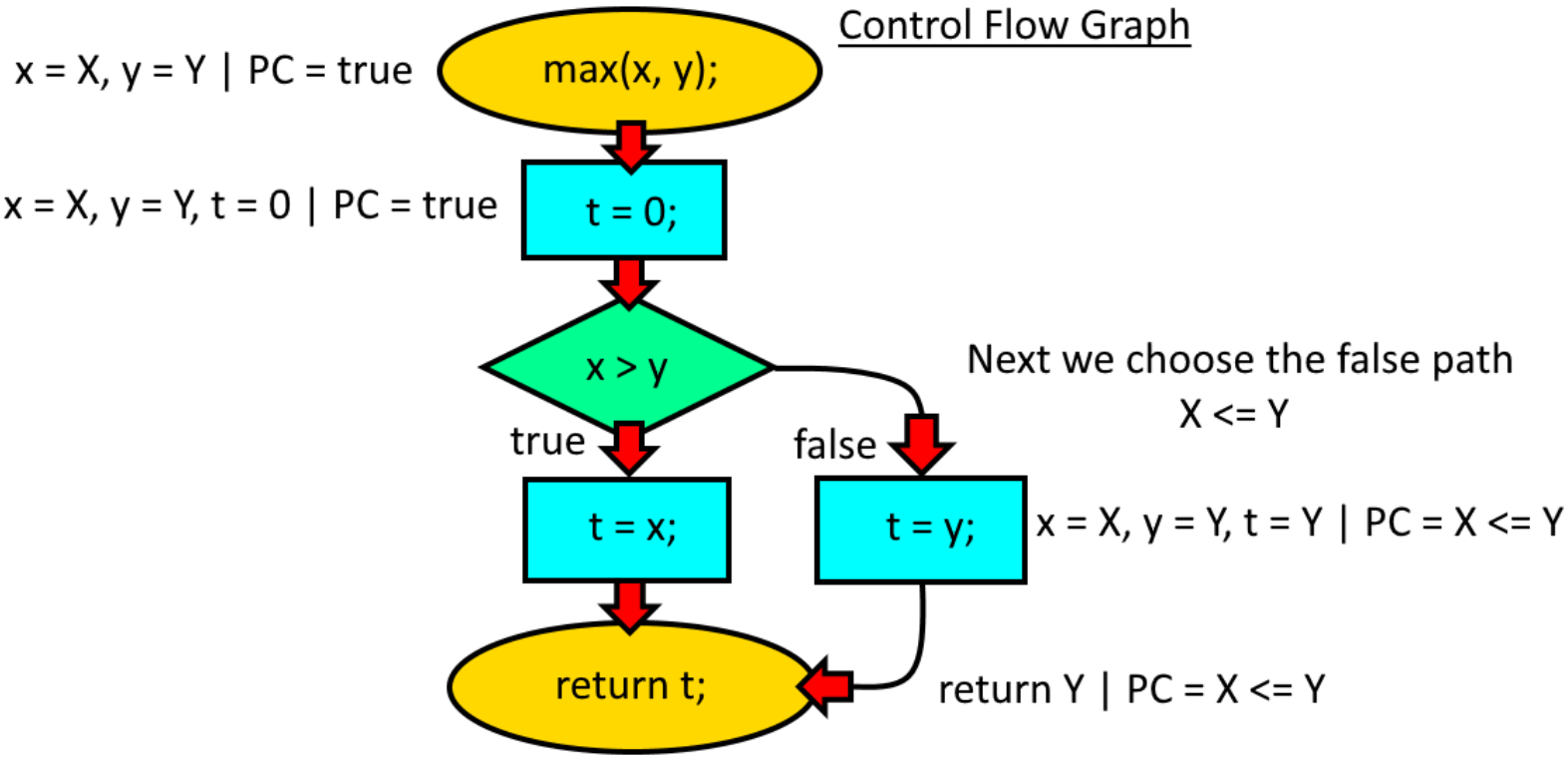
Concolic Execution Example



Concolic Execution Example



Concolic Execution Example



Again we add the failure condition to the PC to see if execution along this path can fail:
Final Constraints: (X <= Y) and ((t < X) or (t < Y) or (t =/= X and t =/= Y))
= (X <= Y) and ((Y < X) or (Y < Y) or (Y =/= X and Y =/= Y))
= (X <= Y) and ((Y < X) or (false) or (Y =/= X and false))
= (X <= Y) and (Y < X)
= false (Unsatisfiable)
So we find that there is no bug along this execution path either.
We have checked every path through the function so the function is correct.

Challenges to Symbolic and Concolic Execution

- *State space explosion* - Number of paths to be executed(and corresponding states) grows exponentially with the number of branches (if statements). Loops also explode the state space.

- *Memory Model* - Any arbitrarily complex object can be regarded as an array of bytes and each byte associated with a distinct symbol. However, when possible, exploiting structural properties of the data may be more convenient.

- *Environment* - Real-world applications constantly interact with the environment (e.g., the file system or the network) through libraries and system calls. These interactions may cause side-effects (such as the creation of a file) that could later affect the execution and must be therefore taken into account.

KLEE LLVM Execution Engine

KLEE is an open source symbolic virtual machine built on top of the LLVM compiler infrastructure.

- Generates tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs.
- KLEE was used to thoroughly check all 89 stand-alone programs in the GNU COREUTILS utility suite.
- Generated tests achieved high line coverage — on average over 90% per tool (median: over 94%) — and beat the coverage of the developers' own hand-written test suites by 16.8%!
- It found ten fatal errors in COREUTILS (including three that had escaped detection for 15 years), which account for more crashing bugs than were reported in 2006, 2007 and 2008 combined.

KLEE Architecture

At a high level, KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter.

KLEE Architecture

At a high level, KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter.

- Each symbolic process has a register file, stack, heap, program counter, and path condition.

KLEE Architecture

At a high level, KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter.

- Each symbolic process has a register file, stack, heap, program counter, and path condition.
- KLEE's representation of a symbolic process is called a state.

KLEE Architecture

At a high level, KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter.

- Each symbolic process has a register file, stack, heap, program counter, and path condition.
- KLEE's representation of a symbolic process is called a state.
- At any one time, KLEE may be executing a large number of states. The core of KLEE is an interpreter loop which selects a state to run and then symbolically executes a single instruction in the context of that state.

KLEE Architecture

At a high level, KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter.

- Each symbolic process has a register file, stack, heap, program counter, and path condition.
- KLEE's representation of a symbolic process is called a state.
- At any one time, KLEE may be executing a large number of states. The core of KLEE is an interpreter loop which selects a state to run and then symbolically executes a single instruction in the context of that state.
- This loop continues until there are no states remaining, or a user-defined timeout is reached.

KLEE Architecture

At a high level, KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter.

- Each symbolic process has a register file, stack, heap, program counter, and path condition.
- KLEE's representation of a symbolic process is called a state.
- At any one time, KLEE may be executing a large number of states. The core of KLEE is an interpreter loop which selects a state to run and then symbolically executes a single instruction in the context of that state.
- This loop continues until there are no states remaining, or a user-defined timeout is reached.
- Unlike a normal process, storage locations for a state — registers, stack and heap objects — refer to expressions (trees) instead of raw data values.

KLEE's approach to challenges

- State Explosion - By implementing the heap as an immutable map, portions of the heap can be shared amongst multiple states. Since KLEE tracks all memory objects, it can implement copy-on-write at the object level (rather than page granularity), dramatically reducing per-state memory requirements.

KLEE's approach to challenges

- State Explosion - By implementing the heap as an immutable map, portions of the heap can be shared amongst multiple states. Since KLEE tracks all memory objects, it can implement copy-on-write at the object level (rather than page granularity), dramatically reducing per-state memory requirements.
- Environment Modeling - The environment is handled by redirecting library calls that access it to models that understand the semantics of the desired action well enough to generate the required constraints. Crucially, these models are written in normal C code which the user can readily customize, extend, or even replace without having to understand the internals of KLEE. KLEE has about 2,500 lines of code to define simple models for roughly 40 system calls (e.g., open, read, write, stat, lseek, ftruncate, ioctl, etc).

KLEE's approach to challenges

- Query Optimization - Almost always, the cost of constraint solving dominates everything else — unsurprising, given that KLEE generates complicated queries for an NP-complete logic. A lot of effort is spend on tricks to simplify expressions:
  - Expression Rewriting
  - Constraint Set Simplification
  - Implied Value Concretization
  - These simplify and ideally eliminate queries before they reach the SMT. Simplified queries make solving faster, reduce memory consumption, and increase the query cache's hit rate.

KLEE's approach to challenges

- State scheduling - KLEE selects the state to run at each instruction by interleaving the following two search heuristics:
  - Random Path Selection maintains a binary tree recording the program path followed for all active states. States are selected by traversing this tree from the root and randomly selecting the path to follow at branch points. Therefore, when a branch point is reached, the set of states in each subtree has equal probability of being selected, regardless of the size of their subtrees.
  - Coverage-Optimized Search tries to select states likely to cover new code in the immediate future. It uses heuristics to compute a weight for each state and then randomly selects a state according to these weights. Currently these heuristics take into account the minimum distance to an uncovered instruction, the call stack of the state, and whether the state recently covered new code.

Tools

Other open source tools:

- angr (Python framework supporting x86, x86-64, ARM, AARCH64, MIPS, MIPS64, PPC, and PPC64)
- FuzzBALL for VineIL / Native
- JPF, jCUTE, janala2, JBSE and KeY for Java
- Otter for C
- SymDroid for Dalvik bytecode
- Kite for LLVM bytecode
- SymJS and Jalangi2 for Javascript
- Rubyx for Ruby
- Pex for .NET Framework

DARPA Cyber Grand Challenge at DEFCON 24

DARPA Cyber Grand Challenge at DEFCON 24

In 2016, DARPA challenged the global innovation community with a $2M prize to build a computer that can hack & patch unknown software with no one at the keyboard. It was the world's first all machine hacking tournament.

DARPA Cyber Grand Challenge at DEFCON 24

In 2016, DARPA challenged the global innovation community with a $2M prize to build a computer that can hack & patch unknown software with no one at the keyboard. It was the world's first all machine hacking tournament.

- 7 teams were selected to participate in the final event. Each team received $750,000 to build a high performance rig.

DARPA Cyber Grand Challenge at DEFCON 24

In 2016, DARPA challenged the global innovation community with a $2M prize to build a computer that can hack & patch unknown software with no one at the keyboard. It was the world's first all machine hacking tournament.

- 7 teams were selected to participate in the final event. Each team received $750,000 to build a high performance rig.
- These rigs would run custom programs written by the contest organizers designed to emulate real software such as email servers, web servers, database applications, etc.

DARPA Cyber Grand Challenge at DEFCON 24

In 2016, DARPA challenged the global innovation community with a $2M prize to build a computer that can hack & patch unknown software with no one at the keyboard. It was the world's first all machine hacking tournament.

- 7 teams were selected to participate in the final event. Each team received $750,000 to build a high performance rig.
- These rigs would run custom programs written by the contest organizers designed to emulate real software such as email servers, web servers, database applications, etc.
- Some of these programs contained bugs that could be exploited to leak memory, execute shell code, etc. The idea is to use these exploits to capture the opponents' flag.

DARPA Cyber Grand Challenge at DEFCON 24

In 2016, DARPA challenged the global innovation community with a $2M prize to build a computer that can hack & patch unknown software with no one at the keyboard. It was the world's first all machine hacking tournament.

- 7 teams were selected to participate in the final event. Each team received $750,000 to build a high performance rig.
- These rigs would run custom programs written by the contest organizers designed to emulate real software such as email servers, web servers, database applications, etc.
- Some of these programs contained bugs that could be exploited to leak memory, execute shell code, etc. The idea is to use these exploits to capture the opponents' flag.
- Fuzzing and Symbolic Execution were the fundamental techniques used to analyze, hack and patch the software.

DARPA Cyber Grand Challenge at DEFCON 24

In 2016, DARPA challenged the global innovation community with a $2M prize to build a computer that can hack & patch unknown software with no one at the keyboard. It was the world's first all machine hacking tournament.

- 7 teams were selected to participate in the final event. Each team received $750,000 to build a high performance rig.
- These rigs would run custom programs written by the contest organizers designed to emulate real software such as email servers, web servers, database applications, etc.
- Some of these programs contained bugs that could be exploited to leak memory, execute shell code, etc. The idea is to use these exploits to capture the opponents' flag.
- Fuzzing and Symbolic Execution were the fundamental techniques used to analyze, hack and patch the software.
- Exploits and patches were generated and deployed completely automatically.

DARPA Cyber Grand Challenge at DEFCON 24

- The CGC Final Event (CFE) was held on August 4, 2016 and lasted for 11 hours.
- The winning systems of the Cyber Grand Challenge (CGC) Final Event were:
  - "Mayhem" - developed by David Brumley, ForAllSecure, Carnegie Mellon University of Pittsburgh, Pa. - $2 million
  - "Xandra" - developed by TECHx, GrammaTech Inc., Ithaca, N.Y., and Charlottesville, Va. - $1 million
  - "Mechanical Phish" - developed by Shellphish, UC Santa Barbara, Ca. - $750,000
- The winner was also awarded the opportunity to play against humans in the 24th DEF CON capture the flag competition.
- Full event live stream: `https://youtu.be/n0kn4mDXY6I`

Applications

Symbolic Execution is one of those techniques that has broken out of the research bubble and made it into a lot of high impact applications.

Applications

Symbolic Execution is one of those techniques that has broken out of the research bubble and made it into a lot of high impact applications.

- Microsoft Sage
  - It is a whitebox fuzzer optimized for long symbolic executions at the x86 binary level.
  - Running continuously since 2008 at the largest fuzzing lab in the world on MS apps such as Word, Office, Powerpoint, etc.
  - Has found 100s of bugs in 100s of apps which were missed by other program analyses and blackbox testing techniques. For instance, revealed nearly one third of the bugs discovered during the development of Windows 7.
  - Bug fixes shipped quietly (no MSRCs) to 1 Billion+ PCs.

Applications

Symbolic Execution is one of those techniques that has broken out of the research bubble and made it into a lot of high impact applications.

- Microsoft Sage
  - It is a whitebox fuzzer optimized for long symbolic executions at the x86 binary level.
  - Running continuously since 2008 at the largest fuzzing lab in the world on MS apps such as Word, Office, Powerpoint, etc.
  - Has found 100s of bugs in 100s of apps which were missed by other program analyses and blackbox testing techniques. For instance, revealed nearly one third of the bugs discovered during the development of Windows 7.
  - Bug fixes shipped quietly (no MSRCs) to 1 Billion+ PCs.
- KLEE (Has found severe bugs in open source software).

Applications

Symbolic Execution is one of those techniques that has broken out of the research bubble and made it into a lot of high impact applications.

- Microsoft Sage
  - It is a whitebox fuzzer optimized for long symbolic executions at the x86 binary level.
  - Running continuously since 2008 at the largest fuzzing lab in the world on MS apps such as Word, Office, Powerpoint, etc.
  - Has found 100s of bugs in 100s of apps which were missed by other program analyses and blackbox testing techniques. For instance, revealed nearly one third of the bugs discovered during the development of Windows 7.
  - Bug fixes shipped quietly (no MSRCs) to 1 Billion+ PCs.
- KLEE (Has found severe bugs in open source software).
- S^2E - Virtual machine augmented with symbolic execution and modular path analyzers. It runs unmodified x86, x86-64, or ARM software stacks, including programs, libraries, the kernel, and drivers.

What I plan to do

- Replace the heuristics used for path selection in concolic execution with a machine learning algorithm.
- Augment the constraint solvers used with a machine learning based solver specifically aimed at solving non linear constraints.

Literature Survey

Randomized Coordinate Shrinking(RaCoS) algorithm from "Derivative-Free Optimization via Classification" research paper. Has been used to solve non linear constraints in concolic execution in another research paper("Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving").

- It is a machine learning based optimization algorithm.
- Can minimize non linear functions that are not differentiable, hence "derivative free optimization".
- Efficient for certain classes of functions(Polynomial time convergence). Has a strong theoretical foundation.
- Simple algorithm which is easy to implement and analyze.

Fuzzing

Fuzzing or fuzz testing 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, or failing built-in code assertions or for finding potential memory leaks.
- Typically, fuzzers are used to test programs that take structured inputs.
- This structure is specified, e.g., in a file format or protocol and distinguishes valid from invalid input.
- An effective fuzzer generates semi-valid inputs that are "valid enough" in that they are not directly rejected by the parser, but do create unexpected behaviors deeper in the program and are "invalid enough" to expose corner cases that have not been properly dealt with.

Fuzzing Example

```
int main(void)
{
        char name[100];

        printf("Enter your name:");
        scanf("%s", name);
        printf("Hello %s!!", name);

        return 0;
}
```

- This program reads a string as user input and displays it back to the user.
- If we enter a string less than 100 characters(Ex: "World") we will get the desired output("Hello World!!").
- However since the user input is read into a fixed length 100 byte buffer using scanf(which does no bounds checking), it is easy to overflow the buffer and rewrite the stack memory.

Fuzzing Example

```
int main(void)
{
        char name[100];

        printf("Enter your name:");
        scanf("%s", name);
        printf("Hello %s!!", name);

        return 0;
}
```

- A fuzzer when given a valid input (like "World") will generate 1000s of new inputs that are variations of the input.
  Ex: "$Worl#d"", "W01234RLD",
  "WWWWWWWWWWWWWWWWWWWWWWWWORLD",
  etc
- Eventually it will generate an input that is longer than 100 bytes and overflow the buffer and crash the program. It will then report this as a bug and provide the crashing input.

Driller: Combining Symbolic Execution and Fuzzing

Driller is a hybrid vulnerability excavation tool which leverages fuzzing and selective concolic execution in a complementary manner, to find deeper bugs in programs.

- There are two different categories of user input:
  - General input, which has a wide range of valid values (e.g., the name of a user) and
  - Specific input, which has a limited set of valid values (e.g., the hash of the aforementioned name).
- Checks for particular values of specific input effectively split an application into compartments, separated by such checks.

Driller: Combining Symbolic Execution and Fuzzing

Driller is a hybrid vulnerability excavation tool which leverages fuzzing and selective concolic execution in a complementary manner, to find deeper bugs in programs.

- Fuzzing is efficient when exploring possible values of general input, inside a compartment, but struggles to identify the precise values needed to satisfy checks on specific input and drive execution flow between compartments.
- On the other hand, Concolic Execution is efficient at determining the values that such specific checks require, but the path explosion problem makes it inefficient for pushing execution inside a compartment.

Example of combining the 2 approaches

```
int main(int argc, char** argv)
{
   int cmd, input1, input3;
   float input2;

   if(argc < 2)return -1;
   else if(strcmp(argv[1], "echo") == 0)
   {
      cmd = 1;
      input1 = atoi(argv[2]);
   }
   else if(strcmp(argv[1], "add") == 0)
   {
      cmd = 2;
      input1 = atoi(argv[2]);
      input2 = atof(argv[3]);
   }
   else if(strcmp(argv[1], "exec") == 0)
   {
      cmd = 3;
      input3 = argv[2];
   }
   else return -1;

   /*
   Complicated parameter parsing and command execution code
   */

   return 0;
}
```

Example of combining the 2 approaches

```
int main(int argc, char** argv)
{
   int cmd, input1, input3;
   float input2;

   if(argc < 2)return -1;
   else if(strcmp(argv[1], "echo") == 0)
   {
      cmd = 1;
      input1 = atoi(argv[2]);
   }
   else if(strcmp(argv[1], "add") == 0)
   {
      cmd = 2;
      input1 = atoi(argv[2]);
      input2 = atof(argv[3]);
   }
   else if(strcmp(argv[1], "exec") == 0)
   {
      cmd = 3;
      input3 = argv[2];
   }
   else return -1;

   /*
   Complicated parameter parsing and command execution code
   */

   return 0;
}
```

This program takes some input as argv

Example of combining the 2 approaches

```
int main(int argc, char** argv)
{
   int cmd, input1, input3;
   float input2;

   if(argc < 2)return -1;
   else if(strcmp(argv[1], "echo") == 0)
   {
      cmd = 1;
      input1 = atoi(argv[2]);
   }
   else if(strcmp(argv[1], "add") == 0)
   {
      cmd = 2;
      input1 = atoi(argv[2]);
      input2 = atof(argv[3]);
   }
   else if(strcmp(argv[1], "exec") == 0)
   {
      cmd = 3;
      input3 = argv[2];
   }
   else return -1;

   /*
   Complicated parameter parsing and command execution code
   */

   return 0;
}
```

This program takes some input as argv

Compares the 1st argument to different commands and initializes some variables

Example of combining the 2 approaches

```
int main(int argc, char** argv)
{
   int cmd, input1, input3;
   float input2;

   if(argc < 2)return -1;
   else if(strcmp(argv[1], "echo") == 0)
   {
      cmd = 1;
      input1 = atoi(argv[2]);
   }
   else if(strcmp(argv[1], "add") == 0)
   {
      cmd = 2;
      input1 = atoi(argv[2]);
      input2 = atof(argv[3]);
   }
   else if(strcmp(argv[1], "exec") == 0)
   {
      cmd = 3;
      input3 = argv[2];
   }
   else return -1;

   /*
   Complicated parameter parsing and command execution code
   */

   return 0;
}
```

This program takes some input as argv

Compares the 1st argument to different commands and initializes some variables

If the 1st argument is not a valid command then abort

Example of combining the 2 approaches

```
int main(int argc, char** argv)
{
   int cmd, input1, input3;
   float input2;

   if(argc < 2)return -1;
   else if(strcmp(argv[1], "echo") == 0)
   {
      cmd = 1;
      input1 = atoi(argv[2]);
   }
   else if(strcmp(argv[1], "add") == 0)
   {
      cmd = 2;
      input1 = atoi(argv[2]);
      input2 = atof(argv[3]);
   }
   else if(strcmp(argv[1], "exec") == 0)
   {
      cmd = 3;
      input3 = argv[2];
   }
   else return -1;

   /*
   Complicated parameter parsing and command execution code
   */

   return 0;
}
```

This program takes some input as argv

Compares the 1st argument to different commands and initializes some variables

If the 1st argument is not a valid command then abort

Otherwise does some complicated parsing and execution of the command.

Example of combining the 2 approaches

```
int main(int argc, char** argv)
{
   int cmd, input1, input3;
   float input2;

   if(argc < 2)return -1;
   else if(strcmp(argv[1], "echo") == 0)
   {
      cmd = 1;
      input1 = atoi(argv[2]);
   }
   else if(strcmp(argv[1], "add") == 0)
   {
      cmd = 2;
      input1 = atoi(argv[2]);
      input2 = atof(argv[3]);
   }
   else if(strcmp(argv[1], "exec") == 0)
   {
      cmd = 3;
      input3 = argv[2];
   }
   else return -1;

   /*
   Complicated parameter parsing and command execution code
   */

   return 0;
}
```

Here 1st argument is a specific input. Only valid values are "echo", "add" and "exec".

Example of combining the 2 approaches

```
int main(int argc, char** argv)
{
   int cmd, input1, input3;
   float input2;

   if(argc < 2)return -1;
   else if(strcmp(argv[1], "echo") == 0)
   {
      cmd = 1;
      input1 = atoi(argv[2]);
   }
   else if(strcmp(argv[1], "add") == 0)
   {
      cmd = 2;
      input1 = atoi(argv[2]);
      input2 = atof(argv[3]);
   }
   else if(strcmp(argv[1], "exec") == 0)
   {
      cmd = 3;
      input3 = argv[2];
   }
   else return -1;

   /*
   Complicated parameter parsing and command execution code
   */

   return 0;
}
```

Here 1st argument is a specific input. Only valid values are "echo", "add" and "exec".

These specific checks separate the program into compartments.

Example of combining the 2 approaches

```
int main(int argc, char** argv)
{
  int cmd, input1, input3;
  float input2;

  if(argc < 2) return -1;
  else if(strcmp(argv[1], "echo") == 0)
  {
    cmd = 1;
    input1 = atoi(argv[2]);
  }
  else if(strcmp(argv[1], "add") == 0)
  {
    cmd = 2;
    input1 = atoi(argv[2]);
    input2 = atof(argv[3]);
  }
  else if(strcmp(argv[1], "exec") == 0)
  {
    cmd = 3;
    input3 = argv[2];
  }
  else return -1;

  /*
  Complicated parameter parsing and command execution code
  */

  return 0;
}
```

Here 1st argument is a specific input. Only valid values are "echo", "add" and "exec".

These specific checks separate the program into compartments.

Compartments (area between checks)
- **Easy to fuzz**
- **Hard to do concolic exec**

Specific checks (separates compartments)
- **Difficult to fuzz**
- **Easy to solve with concolic exec**

Driller: Results on Cyber Grande Challenge Binaries

Experiments were done on a cluster of modern AMD64 processors.
Each binary had four dedicated fuzzer nodes and 64 concolic execution nodes, shared among all binaries.
Each concolic execution job had max 4GB RAM. Analyzed each binary until either a crash was found or 24 hours had passed.

Driller: Results on Cyber Grande Challenge Binaries

Experiments were done on a cluster of modern AMD64 processors. Each binary had four dedicated fuzzer nodes and 64 concolic execution nodes, shared among all binaries.
Each concolic execution job had max 4GB RAM. Analyzed each binary until either a crash was found or 24 hours had passed.

- The symbolic execution baseline experiment faired poorly on this dataset. Out of the 126 applications, symbolic execution discovered vulnerabilities in only 16.

Driller: Results on Cyber Grande Challenge Binaries

Experiments were done on a cluster of modern AMD64 processors. Each binary had four dedicated fuzzer nodes and 64 concolic execution nodes, shared among all binaries.
Each concolic execution job had max 4GB RAM. Analyzed each binary until either a crash was found or 24 hours had passed.

- The symbolic execution baseline experiment faired poorly on this dataset. Out of the 126 applications, symbolic execution discovered vulnerabilities in only 16.
- Fuzzing proved to be sufficient to discover crashes in 68.

Driller: Results on Cyber Grande Challenge Binaries

Experiments were done on a cluster of modern AMD64 processors. Each binary had four dedicated fuzzer nodes and 64 concolic execution nodes, shared among all binaries.
Each concolic execution job had max 4GB RAM. Analyzed each binary until either a crash was found or 24 hours had passed.

- The symbolic execution baseline experiment faired poorly on this dataset. Out of the 126 applications, symbolic execution discovered vulnerabilities in only 16.
- Fuzzing proved to be sufficient to discover crashes in 68.
- In Driller's run, Driller's concolic execution was able to generate a total of 101 new inputs for 13 of these applications. Using these extra inputs, AFL was able to discover an additional 9 crashes, for a total of 77, meaning that Driller achieves a 12% improvement over baseline fuzzing in relation to discovered vulnerabilities.

Conclusion

Symbolic Execution is a extremely useful technique to find software bugs. It can be used at both the source code and binary level. It can check practically infinite number of inputs at once, as opposed to traditional testing techniques which checks 1 input at a time. It can also check all possible paths of execution through the program thus achieving 100% code coverage. This makes Symbolic Execution both Sound (No false positives, generated test cases lead to actual bugs) and Complete (Finds all buggy inputs).

While there are still several challenges involved in scaling symbolic execution to large programs, significant research is going on in this area and as machines get faster and more powerful, it is sure to become more and more feasible in the future.

References

- **KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs**
  Cadar, Cristian and Dunbar, Daniel and Engler, Dawson R and others,
  Stanford University,
  book OSDI, volume 8, pages 209–224, year 2008,
  `http://www.doc.ic.ac.uk/~cristic/papers/klee-osdi-08.pdf`
- **SAGE: whitebox fuzzing for security testing**
  Godefroid, Patrice and Levin, Michael Y and Molnar, David, 2012,
  journal Communications of the ACM, volume 55, number 3, pages 40–44, year 2012, publisher ACM,
  `https://courses.cs.washington.edu/courses/cse484/14au/reading/sage-cacm-2012.pdf`

References

- **Derivative-Free Optimization via Classification**
  Yang Yu and Hong Qian and Yi-Qi Hu,
  Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16),
  book AAAI, volume 16, pages 2286–2292, year 2016,
  `http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/12367/11874`
- **Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving**
  Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li,
  book Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering,
  pages 554–559, year 2016, ACM,
  `https://dl.acm.org/citation.cfm?id=2970364`
- **A survey of static program analysis techniques**
  Wögerer, Wolfgang, Technische Universität Wien, year 2005,

References

- **A survey of dynamic program analysis techniques and tools**
  Gosain, Anjana and Sharma, Ganga,
  book Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014, pages 113–122, year 2015, Springer,
  `https://link.springer.com/chapter/10.1007/978-3-319-11933-5_13`
- **Driller: Augmenting Fuzzing Through Selective Symbolic Execution**
  Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna,
  UC Santa Barbara,
  book NDSS, volume 16, pages 1–16, year 2016,
  `http://cs.ucsb.edu/~chris/research/doc/ndss16_driller.pdf`

References

- Wikipedia:
  - Program analysis,
    `https://en.wikipedia.org/wiki/Program_analysis`
  - Static program analysis,
    `https://en.wikipedia.org/wiki/Static_program_analysis`
  - Symbolic execution, Symbolic execution tools
    `https://en.wikipedia.org/wiki/Symbolic_execution`
  - Concolic testing,
    `https://en.wikipedia.org/wiki/Concolic_testing`
  - Fuzzing,
    `https://en.wikipedia.org/wiki/Fuzzing`
- Youtube MIT OpenCourseWare
  `https://youtu.be/yRVZPvHYHzw`
- DARPA's Cyber Grand Challenge: Final Event Program,
  `https://youtu.be/n0kn4mDXY6I`


THANK
YOU!