Techniques for Improving Software Productivity

Instructor: Mooly Sagiv
TA: Kalev Alpernas

http://cs.tau.ac.il/~msagiv/courses/software-productivity.html

Slides from Eran Yahav, Zach Tatlock and the Noun Project, Wikipedia
Course Prerequisites

• Soft: Logic in Computer Science
• Hard: Software Project
Course Requirements

• The students must solve all homework assignments but one (40%)
  – Apply a tool
  – ~10 hours per project
  – First assignment available on Thursday

• 60% final project
Software is Everywhere
Software is Everywhere

Exploitable
Exploitable Software is Everywhere
void foo (char *x) {
    char buf[2];
    strcpy(buf, x);
}

int main (int argc, char *argv[]) {
    foo(argv[1]);
}

> ./a.out

Segmentation fault

memory

return address

ca
ra
ab

terminal

source code

abracadabra
Buffer Overrun Exploits

```c
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0) auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0) auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(check_authentication(argv[1])) {
        printf("\n----\nAccess Granted.\n----\n");
        printf("\n----\nAccess Granted.\n----\n");
        } else
        printf("\nAccess Denied.\n");
}
```

(source: “hacking – the art of exploitation, 2nd Ed”)
Attack

evil input

Application

AAAAAAAAAAAAAA

Access Granted. 65
• A sailor on the U.S.S. Yorktown entered a 0 into a data field in a kitchen-inventory program
• The 0-input caused an overflow, which crashed all LAN consoles and miniature remote terminal units
• The Yorktown was dead in the water for about two hours and 45 minutes
The New York Times

The Stock Market Bell Rings, Computers Fail, Wall Street Cringes

By NATHANIEL POPPER  JULY 8, 2015

Problems with technology have at times roiled global financial markets, but the 223-year-old New York Stock Exchange has held itself up as an oasis of humans ready to step in when the computers go haywire.

On Wednesday, however, those working on the trading floor were left helpless when the computer systems at the exchange went down for nearly four hours in the middle of the day, bringing an icon of capitalism’s ceaseless energy to a costly halt.

The exchange ultimately returned to action shortly before the closing bell,
One Day Last Summer...

The New York Times
The Stock Market Bell Rings, Computers Fail, Wall Street Cringes

By NATHANIEL POPPER JULY 8, 2015

Problems with technology have at times roiled global financial markets, but the 223-year-old New York Stock Exchange has held itself up as an oasis of humans ready to step in when the computers go haywire.

On Wednesday, however, those working on the trading floor were left helpless when the computer systems at the exchange went down for nearly four hours in the middle of the day, bringing an icon of capitalism's ceaseless energy to a costly halt.

The exchange ultimately returned to action shortly before the closing bell,
One Day Last Summer...

The New York Times
The Stock Market Bell Rings, Computers Fail, Wall Street Cringes

By NATHANIEL POPPER  JULY 8, 2015

Problems with technology have at times roiled global financial markets, but the 223-year-old New York Stock Exchange has held itself up as an oasis of humans ready to step in when the computers go haywire.

On Wednesday, however, those working on the trading floor were left helpless when the computer systems at the exchange went down for nearly four hours in the middle of the day, bringing an icon of capitalism's ceaseless energy to a costly halt.

The exchange ultimately returned to action shortly before the closing bell,
Software is Complex
### MySQL Workbench 5.2 Code Statistics

#### Files vs Lines of Code

<table>
<thead>
<tr>
<th>Category</th>
<th>Files</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>65</td>
<td>43782</td>
</tr>
<tr>
<td>Windows</td>
<td>430</td>
<td>93065</td>
</tr>
<tr>
<td>MacOSX</td>
<td>97</td>
<td>19198</td>
</tr>
<tr>
<td>Common</td>
<td>1497</td>
<td>325001</td>
</tr>
<tr>
<td>MForms</td>
<td>36</td>
<td>9499</td>
</tr>
<tr>
<td>3rd Party</td>
<td>457</td>
<td>201401</td>
</tr>
<tr>
<td>Total</td>
<td>2582</td>
<td>691946</td>
</tr>
</tbody>
</table>

#### Code by Category

- Linux: 58%
- Common: 17%
- MForms: 3%
- MacOSX: 18%
- 3rd Party: 1%

#### Code by Language

- C/C++: 56%
- Python: 30%
- C#: 10%
- Objective-C: 4%
- Lua: 4%
Cost of software bugs

• 59.5 billion dollars in the US due to software bugs
• Software security
  – Cars, Planes, Radiotherapy, Internet, ….
• Software agility
Improving Software Productivity

- High level programming languages
  - Abstractions
- Software Engineering
  - Software designs
- Software tools
  - Software testing
  - Software debugging
  - Formal Verification
Software Testing

• Goal: “to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances” [E. F. Miller, Introduction to Software Testing Technology]
The Testing Spectrum

- Unit Testing: basic unit of software
- Integration Testing: combination
- System’s testing: end-to-end
- Acceptance testing: client check
Testing Techniques

- Random testing: Runs the program on random inputs
- Symbolic techniques
- Concolic techniques
- Adequacy of test suit
  - Coverage
  - Mutation testing: Modify the program in a small way
    - Check the adequacy of the test suit
Symbolic vs. Concrete Testing

Mooly Sagiv
Program Path

• Program Path
  – A path in the control flow of the program
    • Can start and end at any point
    • Appropriate for imperative programs

• Feasible program path
  – There exists an input that leads to the execution of this path

• Infeasible program path
  • No input that leads to the execution
void grade(int score) {
    A: if (score < 45) {
        B: printf("fail");
    }
    else
    C: printf("pass");
}
D: if (score > 85) {
    E: printf("with honors");
}
F:
Concrete vs. Symbolic Executions

• Real programs have many infeasible paths
  – Ineffective concrete testing
• Symbolic execution aims to find rare errors
Symbolic Testing Tools

- EFFIGY [King, IBM 76]
- PEX [MSR]
- SAGE [MSR]
- SATURN [Stanford]
- KLEE [Stanford]
- Java pathfinder [NASA]
- Bitscope [Berkeley]
- Cute [UIUC, Berkeley]
- Calysto [UBC]
Finding Infeasible Paths Via Constraint Solving

```c
void grade(int score) {
    A: if (score < 45) {
        B: printf("fail");
    } else
    C: printf("pass");
    }
    D: if (score > 85) {
        E: printf("with honors");
    }
    F:
}
```

\[\text{score} < 45 \land \text{score} > 85\] UNSAT
Plan

• Random Testing
• Symbolic Testing
• Concolic Testing
Fuzzing [Miller 1990]

- Test programs on random unexpected data
- Can be realized using black/white testing
- Can be quite effective
  - Operating Systems
  - Networks
- ...
- Usually implemented via instrumentation
- Tricky to scale for programs with many paths

```c
If (x == 10001) {
    ....
    if (f(*y) == *z) {
        ....
    }
}

int f(int *p) {
    if (p !=NULL) {
        return q ;
    }
}
```
Success Stories Fuzzing

- Crashes to Unix [90s]
- Crashes to all systems
  - American Fuzzy Lop
    http://lcamtuf.coredump.cx/afl/
Symbolic Exploration

• Execute a program on symbolic inputs
• Track set of values symbolically
• Update symbolic states when instructions are executed
• Whenever a branch is encountered check if the path is feasible using a theorem prover call
Symbolic Execution Tree

• The constructed symbolic execution paths
• Nodes
  – Symbolic Program States
• Edges
  – Potential Transitions
• Constructed during symbolic evaluation
• Each edge requires a theorem prover call
1) int x, y;
2) if (x > y) {
   3) x = x + y;
   4) y = x - y;
   5) x = x - y;
   6) if (x > y)
      7) assert false;
   8})
int f(int x) { return 2 * x ;}
int h(int x, int y) {
  1) if (x!= y) {
    2) if (f(x) == x +10) {
      3) abort() // * error */
    }
  }
  4) return 0;
}
Non-Deterministic Behavior

```c
int x; y;
1) if (nondet()) {
   2) x = 7;
}
else {
   3) x = 19;
}
4)
```
Loops

1) int i;
2) while i < n {
   i = i + 1;
}
3) if (n == 10^6) {
   abort();
}
4) }
Scaling Issues for Symbolic Exploration
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

Concrete Execution
symbolic state

Concrete state
x = 22, y = 7

Symbolic state
x = x₀, y = y₀

path condition
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```

### Concrete Execution
- x = 22, y = 7, z = 14
- x = x₀, y = y₀, z = 2*y₀

### Symbolic Execution
- concrete state
- symbolic state
- path condition
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution
- **Concrete State**: `x = 22, y = 7, z = 14`

Symbolic Execution
- **Symbolic State**: `x = x₀, y = y₀, z = 2*y₀`
- **Path Condition**: `2*y₀ != x₀`
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

```
concrete state
```

Symbolic Execution

```
symbolic state
```

Solve: \(2y_0 == x_0\)
Solution: \(x_0 = 2, y_0 = 1\)

x = 22, y = 7, z = 14
x = x_0, y = y_0, z = 2*y_0

2*y_0 != x_0
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>concrete state</td>
<td>symbolic state</td>
</tr>
<tr>
<td>x = 2, y = 1</td>
<td>x = x₀, y = y₀</td>
</tr>
<tr>
<td>path condition</td>
<td>path condition</td>
</tr>
</tbody>
</table>
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void teste (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```

Concrete Execution

- **concrete state**: `x = 2, y = 1, z = 2`
- **symbolic state**: `x = x₀, y = y₀, z = 2*y₀`
- **path condition**: `2*y₀ == x₀`
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Concolic Testing Approach

int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

Concrete Execution

Symbolic Execution

concrete state  symbolic state  path condition

Solve: (2*y₀ == x₀) \land (x₀ > y₀ + 10)
Solution: x₀ = 30, y₀ = 15

x = 2, y = 1,  z = 2
x = x₀, y = y₀, z = 2*y₀

2*y₀ == x₀
x₀ \cdot y₀ + 10
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

- `x = 30, y = 15`

Symbolic Execution

- `x = x₀, y = y₀`

Concrete state

- `x = x₀, y = y₀`

Path condition
The Concolic Testing Algorithm

1. Classify input variables into symbolic / concrete
2. Instrument to record symbolic vars and path conditions
3. Choose an arbitrary input
4. Execute the program
5. Symbolically re-execute the program
6. Negate the unexplored last path condition
7. Is there an input satisfying constraint
SAGE: Whitebox Fuzzing for Security Testing

• Check correctness of Win’7, Win’8
• 200+ machine years
• 1 Billion+ SMT constraints
• 100s of apps, 100s of bugs
• 1/3 of all Win7 WEX security bugs found
• Millions of dollars saved
Automatic Program Verification

Program $P$

Desired Properties $\varphi$

Solver

Is there a behavior of $P$ that violates $\varphi$?

Counterexample

Proof
Example

```c
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0) auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0) auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(check_authentication(argv[1])) {
        printf("\n-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-\n");
        printf( " Access Granted.\n");
        printf("\nAccess Granted.\n");
    }
    printf("\nAccess Denied.\n");
}
```
Undecidability

• The Halting Problem
  – Does the program $P$ terminate on input $I$

• Rice’s Theorem
  – Any non-trivial property of partial functions, there is no general and effective method to decide if program computes a partial function with that property
Coping with Undecidability

- Permits occasional divergence
- Limited programs (not Turing Complete)
- Unsound Verification
  - Explore limited program executions
- Incomplete Verification
  - Explore superset of program executions
- Programmer Assistance
  - Inductive loop invariants
Limited Programs

• Finite state programs
  – Finite state model checking
    • Explicit state SPIN, CHESS
    • Symbolic model checking SMV

• Loop free programs
  – Configuration files
Unsound Verification

- Dynamic checking
  - Valgrind, Parasoft Insure, Purify, Eraser
- Bounded Model Checking
- Concolic Executions
The SAT Problem

- Given a propositional formula (Boolean function)
  - $\varphi = (a \lor b) \land (\neg a \lor \neg b \lor c)$
- Determine if $\varphi$ is satisfiable
  - Find a satisfying assignment or report that such does not exit
- For $n$ variables, there are $2^n$ possible truth assignments to be checked
SAT made some progress...
Bounded Model Checking

Program P → FrontEnd → Propositional Formula $[P(k)] \land \neg \varphi$ → SAT Solver → UNSAT

Input Bound k

Desired Properties $\varphi$

Assignment
A Simple Example

Program

```c
int x;
int y=8, z=0, w=0;
if (x)
    z = y - 1;
else
    w = y + 1;
assert (z == 5 ||
        w == 9)
```

Constraints

```
y = 8,
z = x ? y - 1 : 0,
w = x ? 0 : y + 1,
z != 5,
w != 9
```

SAT

counterexample found!

```
y = 8, x = 1, w = 0, z = 7
```
A Simple Example

Program

```c
int x;
int y=8,z=0,w=0;
if (x)
    z = y - 1;
else
    w = y + 1;
assert (z == 7 ||
    w == 9)
```

Constraints

```c
y = 8,
z = x ? y - 1 : 0,
w = x ? 0 : y + 1,
z != 7,
w != 9
```

UNSAT
Assertion always holds!
Summary Bounded Model Checking

• Excellent tools exist (CBMC, Alloy)
• Many bugs occur on small inputs
• Useful for designs too
• Scalability is an issue
• Challenging features
  – Bounded arithmetic
  – Pointers and Heap
  – Procedures
  – Concurrency
Success Stories BMC

• Car industry
• Amazon
• Regression
System $S$ is **safe** if no bad state is reachable.

- $R_0 = Init$ – Initial states, reachable in 0 transitions
- $R_{i+1} = R_i \cup \{\sigma' \mid \sigma \rightarrow \sigma' \text{ and } \sigma \in R_i\}$
- $R = R_0 \cup R_1 \cup R_2 \cup \ldots$
- Safety: $R \cap Bad = \emptyset$
- K-Safety: $R_K \cap Bad = \emptyset$
Inductive Invariants

System S is safe if no bad state is reachable
System S is safe iff there exists an inductive invariant $\text{Inv}$ s.t.:

- $\text{Inv} \cap \text{Bad} = \emptyset$ (Safety)
- $\text{Init} \subseteq \text{Inv}$ (Initiation)
- If $\sigma \in \text{Inv}$ and $\sigma \rightarrow \sigma'$ then $\sigma' \in \text{Inv}$ (Consecution)
Counterexample To Induction (CTI)

States $\sigma, \sigma'$ are a CTI of $\text{Inv}$ if:
• $\sigma \in \text{Inv}$
• $\sigma' \notin \text{Inv}$
• $\sigma \rightarrow \sigma'$

• A CTI may indicate:
  • A bug in the system
  • A bug in the safety property
  • A bug in the invariant
    • Too weak
    • Too strong
Strengthening & Weakening from CTI

Strengthening:

Weakening:

\[ \sigma \in \text{Inv}_{\sigma} \]

\[ \sigma' \notin \text{Inv}_{\sigma} \]
Deductive (Semi-Automatic) Verification

- **Program** $P$
- **Candidate Inductive Invariant** $I$
- **Safety Property** $\varphi$

**Solver**

*Is there a behavior of $P$ that violates the inductiveness of $I$?*

- **Counterexample to induction (CTI)**
- **Unknown**
- **Proof**
Deductive Verification

Is there a behavior of $P$ that violates the inductiveness of $I$?

1: $x := 1$;
2: $y := 2$;
while * do {
    3: assert $x \geq 1$;
    4: $x := x + y$;
    5: $y := y + 1$
}
6:

$I = \text{at}(3) \Rightarrow x \geq 1$

at(3) $\Rightarrow x \geq 1$

Solver

Is there a behavior of $P$ that violates the inductiveness of $I$?

$x := x + y; y := y + 1$

1: $x = 1, y = -2$
2: $x = -1, y = -1$

$I$
Deductive Verification

1: x := 1;
2: y := 2;
while * do {
  3: assert x ≥ 1;
  4: x := x + y;
  5: y := y + 1
}
6:

Is there a behavior of P that violates the inductiveness of I?

at(3) ⇒ x ≥ 1 ∧ y ≠ -2

Solver

Is there a behavior of P that violates the inductiveness of I?

I

3: x = 1, y = -7

x := x + y; y := y + 1

x = -6, y = -6

¬I
Deductive Verification

1: x := 1;
2: y := 2;
while * do {
   3: assert x ≥ 1;
   4: x := x + y;
   5: y := y + 1
}
6:

Is there a behavior of P that violates the inductiveness of I?

Proof

at(3) ⇒ x ≥ 1 ∧ y ≥ 0

Solver

at(3) ⇒ x ≥ 1
Algorithmic Deductive Verification

• SAT/SMT has made huge progress in the last decade

• Great impact on verification:
  Dafny[ITP’13], IronClad/IronFleet[SOSP’15], and more

• **State**: finite first-order structure over vocabulary V

• **Initial** states and **safety** property (first-order formulas):
  • \text{Init}(V) – initial states
  • \text{Bad}(V) – bad states

• **Transition relation**:
  first-order formula \text{TR}(V, V')
  \text{V'} is a copy of V describing the next state

Algorithmically Checking Inductiveness

\( \text{Inv} \) is an **inductive invariant** if:

- **Initiation:** \( \text{Init} \Rightarrow \text{Inv} \) \( \text{Init} \land \neg \text{Inv} \) unsat
- **Safety:** \( \text{Inv} \Rightarrow \neg \text{Bad} \) \( \text{Inv} \land \text{Bad} \) unsat
- **Consecution:** \( \text{Inv} \land \text{TR} \Rightarrow \text{Inv}' \) \( \text{Inv} \land \text{TR} \land \neg \text{Inv}' \) unsat

**System State Space**

[Diagram showing the system state space with regions for \( \text{Inv} \), \( \text{Reach} \), and \( \text{Bad} \)]
Algorithmic Deductive Verification

Program $P$

Candidate Inductive Invariant $I$

Safety Property $\varphi$

Solver

Is there a behavior of $P$ that violates the inductiveness of $I$?

Countereexample to induction (CTI)

Unknown

Proof
Challenges

1. Formal specification:
   • Modeling the system (TR, Init)
   • Formalizing the safety property (Bad)

2. Inductive Invariants (Inv)
   • Hard to specify manually
   • Hard to maintain
   • Hard to infer automatically

3. Deduction – Checking inductiveness
   • Undecidability of implication checking
     • Unbounded state, arithmetic, quantifier alternation
Existing Approaches for Verification

- Automated invariant inference
  - Abstract Interpretation
    - Ultimately limited due to undecidability
- Use SMT for deduction with manual program annotations (e.g. Dafny)
  - Requires programmer effort to provide inductive invariants
  - SMT solver may diverge (matching loops, arithmetic)
- Interactive theorem provers (e.g. Coq, Isabelle/HOL)
  - Programmer gives inductive invariant and proves it
  - Huge effort (10-50 lines of proof per line of code)
Abstract Interpretation

- Automatically prove that the program is correct by also considering infeasible executions
- Abstract interpretation of program statements/conditions
- Conceptually explore a superset of reachable states
- Sound but incomplete reasoning
- Automatically infer sound inductive invariants
Automatic Program Verification

Program $P$

Desired Properties $\varphi$

Solver

*Is there a behavior of $P$ that violates $\varphi$?*

Counterexample

Unknown

Proof
1: x = 2;
2: while true {x > 0} do
   3: x = 2* x – 1
4: pc: int(x)

1: [0, 0]
2: [2, 3]
3: [2, 2]
4: [3, 3]
1: \( x = 2; \)
2: while true \( \{x > 0\} \) do
   3: \( x = 2^*x - 1 \)

pc: \( \text{int}(x) \)

Interval Based Abstract Interpretation

- 1: \([0, 0]\)
- 2: \([2, 2]\)
- 3: \([2, \infty]\)
- 4: \([3, \infty]\)
Interval Based Abstract Interpretation

1: x = 2, y = 2
2: while true \( \{ x = y \} \) do
   3: x = 2 * x - 1,
      y = 2 * y - 1
4: pc: int(x), int(y)

1: [0, 0], [0, 0]
2: [2, 2], [2, 2]
3: [2, 2], [2, 2]
4: [3, 3], [3, 3]
2: [2, 3], [2, 3]
node search(node h, int v) {
    1: node x = h;
    2: while (h != NULL) {
        3: if (x->d == v) return x;
        4: assert x != null; x = x->n;
    }
    5: return (node) NULL
node search(node h, int v) {
    1: node x = h;
    2: while (x != NULL) {
        3: if (x->d == v) return x;
        4: assert x != null; x = x->n;
    }
    5: return (node) NULL
}
Odd/Even Abstract Interpretation

1: while (x != 1) do {
  2: if (x % 2) == 0
      { 3: x := x / 2; }
  else
      { 4: x := x * 3 + 1; }
  5: assert (x % 2 == 0); }
6: }

1: ?
2: ?
3: E
4: O
5: E
6: O
Abstract Interpretation

Concrete

Sets of stores

Abstract

Descriptors of sets of stores

$\alpha$, $\gamma$, $\alpha$, $\alpha$
Odd/Even Abstract Interpretation

All concrete states

\{x: x \in \text{Even}\} \{ -2, 1, 5 \}

\{0, 2\}

\{0\} \{2\}

\emptyset

\alpha \gamma \Omega \varepsilon \Omega \varepsilon
Odd/Even Abstract Interpretation

All concrete states

\{
\{-2, 1, 5\}
\{x: x \in \text{Even}\}
\{0, 2\}
\{0\}
\{2\}
\emptyset
\}

\{0, 2\}
\{2\}
\{0\}
\emptyset

E

? O

\gamma \alpha
\alpha
\alpha
\alpha
\alpha
\alpha
\alpha

\gamma \alpha

\{x: x \in \text{Even}\}
Odd/Even Abstract Interpretation

All concrete states

\{\{-2, 1, 5\}\}
\{x: x \in Even\}
\{0, 2\}
\{0\} \{2\}

\emptyset

\gamma \alpha
\alpha
(Best) Abstract Transformer

Concrete Representation → Concrete Transition → Concrete Representation

Abstract Representation ← Abstract Transition ← Abstract Representation

Concretization → Concrete Representation

Abstraction → Abstract Representation

St
Odd/Even Abstract Interpretation

1: while (x != 1) do {
  2: if (x % 2) == 0 {
    3: x := x / 2;
  } else {
    4: x := x * 3 + 1;
  }
  5: assert (x % 2 == 0);
}
6: }
Summary Abstract Interpretation

• Conceptual method for building static analyzers
• A lot of techniques:
  – join, meet, widening, narrowing, procedures
• Can be combined with theorem provers
“Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we’re building tools that can do actual proof about the software and how it works in order to guarantee the reliability.” Bill Gates
Success Story: Astrée

• Developed at ENS
• A tool for checking the absence of runtime errors in Airbus flight software

[WCRE’2001] A. Miné: The Octagon Abstract Domain
Success: Panaya
Making ERP easy

• Static analysis to detect the impact of a change for ERP professionals (slicing)
• Developed by N. Dor and Y. Cohen
• Acquired by Infosys

[ISSTA’08] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, D. Weiss: Customization change impact analysis for erp professionals via program slicing
<table>
<thead>
<tr>
<th>Week</th>
<th>Lecture</th>
<th>Recitation</th>
<th>Exercise</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview</td>
<td>No Recitation</td>
<td>No assignment</td>
</tr>
<tr>
<td>2</td>
<td>SAT and SMT Solvers</td>
<td>Z3</td>
<td>Graph algorithms with Z3</td>
</tr>
<tr>
<td>3</td>
<td>Bounded Model Checking</td>
<td>CBMC</td>
<td>CBMC</td>
</tr>
<tr>
<td>4</td>
<td>Concolic Testing</td>
<td>KLEE</td>
<td>KLEE</td>
</tr>
<tr>
<td>5</td>
<td>Deductive Verification 1</td>
<td>No Recitation</td>
<td>No assignment</td>
</tr>
<tr>
<td>6</td>
<td>Deductive Verification 2</td>
<td>Dafny</td>
<td>Dafny</td>
</tr>
<tr>
<td>7</td>
<td>Static Analysis</td>
<td>Apron, Absint</td>
<td>Apron and AbsInt</td>
</tr>
<tr>
<td>8</td>
<td>Random Testing</td>
<td>Quickcheck, Randoop, Simullant, Autotest, YETI, GramTest</td>
<td>Use the tools</td>
</tr>
<tr>
<td>9</td>
<td>Fuzz Testing</td>
<td>TBD</td>
<td>TBD</td>
</tr>
<tr>
<td>10</td>
<td>Mutation Testing</td>
<td>TBD</td>
<td>TBD</td>
</tr>
<tr>
<td>11</td>
<td>Unit Testing</td>
<td>TBD</td>
<td>TBD</td>
</tr>
<tr>
<td>12</td>
<td>Delta Debugging</td>
<td>TBD</td>
<td>TBD</td>
</tr>
<tr>
<td>13</td>
<td>Program Synthesis</td>
<td>TBD</td>
<td>TBD</td>
</tr>
<tr>
<td>14</td>
<td>System's Code</td>
<td>TBD</td>
<td>TBD</td>
</tr>
<tr>
<td>15</td>
<td>Network and Cloud</td>
<td>TBD</td>
<td>TBD</td>
</tr>
</tbody>
</table>
Course Benefits

• Learn about research which is becoming mature
• Understand the limits of formal methods