Automated Software Analysis Techniques For High Reliability: A Concolic Testing Approach

Moonzoo Kim
Contents

• Automated Software Analysis Techniques
  – Background
  – Concolic testing process
  – Example of concolic testing

• Case Study: Busybox utility

• Future Direction and Conclusion
Main Target of Automated SW Analysis

- Requirements Specification
- Architectural Design
- Detailed Design
- Source Code
- Unit Test
- Integration Test
- System Test
- Acceptance Test

Manual Labor

Abstraction
Automated Software Analysis Techniques

• Aims to explore possible behaviors of target systems in an exhaustive manner
• Key methods:
  – Represents a target program/or executions as a “logical formula”
  – Then, analyze the logical formula (a target program) by using logic analysis techniques

Weakness of conventional testing

Symbolic execution (1970)
Model checking (1980)
SW model checking (2000)
Concolic testing (2005 ~)
Hierarchy of SW Coverages

(SW) Model checking

Concolic testing
Weaknesses of the Branch Coverage

/* TC1: x= 1, y = 1;  
   TC2: x=-1, y=-1; */
void foo(int x, int y) {
    if ( x > 0)
        x++;  
    else
        x--; 
    if(y >0)
        y++;  
    else
        y--;  
    assert (x * y >= 0); 
} 

Systematic testing techniques are necessary for quality software!
-> Integration testing is not enough
-> Unit testing with automated test case generation is desirable for both productivity and quality
## Dynamic v.s. Static Analysis

<table>
<thead>
<tr>
<th>Pros</th>
<th>Dynamic Analysis (i.e., testing)</th>
<th>Static Analysis (i.e. model checking)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Real result</td>
<td>• Complete analysis result</td>
<td></td>
</tr>
<tr>
<td>• No environmental limitation</td>
<td>• Fully automatic</td>
<td></td>
</tr>
<tr>
<td>• Binary library is ok</td>
<td>• Concrete counter example</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cons</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Incomplete analysis result</td>
<td>• Consumed huge memory space</td>
<td></td>
</tr>
<tr>
<td>• Test case selection</td>
<td>• Takes huge time for verification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• False alarms</td>
<td></td>
</tr>
</tbody>
</table>

-> **Concolic testing**
Concolic Approach

• Combine concrete and symbolic execution
  – Concrete + Symbolic = Concolic

• In a nutshell, concrete execution over a concrete input guides symbolic execution
  – No false positives

• Automated testing of real-world C Programs
  – Execute target program on automatically generated test inputs
  – All possible execution paths are to be explored
  – Higher branch coverage than random testing
Overview of Concolic Testing Process

1. Select input variables to be handled symbolically
2. A target C program is statically instrumented with probes, which record symbolic path conditions
3. The instrumented C program is executed with given input values
   • Initial input values are assigned randomly
4. Obtain a symbolic path formula $\varphi_i$ from a concrete execution over a concrete input
5. One branch condition of $\varphi_i$ is negated to generate the next symbolic path formula $\psi_i$
6. A constraint solver solves $\psi_i$ to get next concrete input values
   • Ex. $\varphi_i: (x < 2) \land (x + y \geq 2)$ and $\psi_i: (x < 2) \land (x + y < 2)$.
     One solution is $x=1$ and $y=0$
7. Repeat step 3 until all feasible execution paths are explored
Concolic Testing Example

- **Random testing**
  - Probability of reaching `Error()` is extremely low

- **Concolic testing generates the following 4 test cases**
  - `(0,0,0): initial random input`
    - Obtained symbolic path formula (SPF) φ: `a!=1`
    - Next SPF ψ generated from φ: `!(a!=1)`
  - `(1,0,0): a solution of ψ (i.e. !(a!=1))`
    - SPF φ: `a==1 && b!=2`
    - Next SPF ψ: `a==1 && !(b!=2)`
  - `(1,2,0)`
    - SPF φ: `a==1 && (b==2) && (c!=3*a + b)`
    - Next SPF ψ: `a==1 && (b==2) && !(c!=3*a + b)`
  - `(1,2,5)`
    - Covered all paths and `Error()` reached

```c
// Test input a, b, c
void f(int a, int b, int c) {
  if (a == 1) {
    if (b == 2) {
      if (c == 3*a + b) {
        Error();
      }
    }
  }
}
```
Example

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}

• Random Test Driver:
  • random memory graph reachable from p
  • random value for x

• Probability of reaching Error() is extremely low

Example from the slides “CUTE: A Concolic Unit Testing Engine for C” by K.Sen 2005
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

p = p₀, x = x₀

p

, x = 236

NULL

12/42
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
            return 0;
}
```
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}

Concrete Execution

Symbolic Execution

solve: $x_0 > 0$ and $p_0 \neq \text{NULL}$

$p_0 = \text{NULL}$, $x_0 = 236$

$x_0 > 0$

$p = p_0$, $x = x_0$

$x_0 = 236$, $p_0$
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
Concolic Testing

typedef struct cell {
int v;
struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>p=634, x=236.0</td>
<td>p=p0, x=x0, p-&gt;v=v0, p-&gt;next=n0</td>
<td>x0&gt;0, p0!=NULL, 2x0+1!=v0</td>
</tr>
</tbody>
</table>
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}

Concrete Execution
Symbolic Execution

solve: $x_0 > 0$ and $p_0 \neq \text{NULL}$
and $2x_0 + 1 = v_0$

$x_0 = 1$, $p_0 = \text{NULL}$

$p = p_0$, $x = x_0$, $p->v = v_0$, $p->next = n_0$
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0) {
        if (p != NULL) {
            if (f(x) == p->v) {
                if (p->next == p)
                    Error();
            }
        }
    }
    return 0;
}

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

p=p_0, x=x_0, p->v=v_0, p->next=n_0

p=3, x=1

3
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}

Concrete Execution

Symbolic Execution

custom state

symbolic state

concrete state

symbols

\[ p = p_0, \ x = x_0, \ p -> v = v_0, \ p -> next = n_0 \]

x_0 > 0
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}

Concrete Execution  Symbolic Execution

Concrete state  Symbolic state  constraints

\begin{align*}
\text{Concrete state} & : & p=p_0, & x=x_0, \\
\text{Symbolic state} & : & p=p_0, & x=x_0, \\
\text{Constraints} & : & x_0 > 0, & p_0 \neq \text{NULL} \\
\end{align*}
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)  
                if (p->next == p)
                    Error();
            return 0;
    }

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

x_0 > 0
p_0 \neq NULL
2x_0 + 1 = v_0
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
            return 0;
}

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0 > 0$
$p_0 \neq NULL$
$2x_0 + 1 = v_0$
$n_0 \neq p_0$
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2 * v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
            return 0;
    return 0;
}
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
            return 0;
}

Concolic Testing

Concrete Execution

Symbolic Execution

concrete state

symbolic state

solve: $x_0 > 0$ and $p_0 \neq \text{NULL}$ and $2x_0 + 1 = v_0$ and $n_0 = p_0$

constraints

$x_0 > 0$
$p_0 \neq \text{NULL}$
$2x_0 + 1 = v_0$
$p_0 \neq p_0$

$p = p_0$, $x = x_0$, $p->v = v_0$, $p->next = n_0$
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
            return 0;
}

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

x_0 > 0
p_0 \neq NULL

p = p_0, x = x_0,
p->v = v_0,
p->next = n_0

35/42
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}

Concolic Testing

Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

$x_0 > 0$
$p_0 \neq NULL$
$2x_0 + 1 = v_0$
Concolic Testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}

Concrete Execution

Symbolic Execution

Error() reached
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    Error();
    return 0;
}
Summary: Concolic Testing

• Pros
  – Automated test case generation
  – High coverage
  – High applicability (no restriction on target programs)

• Cons
  – If a target program has external binary function calls, coverage might not be complete
    • Ex. if( sin(x) + cos(x) == 0.3) { error(); }
  – Current limitation on pointer and array
  – Slow analysis speed due to a large # of TCs
SAGE: Whitebox Fuzzing for Security Testing @ Microsoft

- X86 binary concolic testing tool to generate millions of test files automatically targeting large applications
  - used daily in Windows, Office, etc.
- Mainly targets crash bugs in various windows file parsers (>hundreds)
- Impact: since 2007 to 2013
  - 500+ machine years
  - 3.4 Billion+ constraints
  - 100s of apps, 100s of bugs
    - 1/3 of all security bugs detected by Win7 WEX team were found by SAGE
  - Millions of dollars saved

This slide quotes PLDI 2013 MSR Open House Event poster “SAGE: WhiteboxFuzzing for Security Testing” by E.Bounimova, P.Godefroid, and D.Molnar
Microsoft Project Springfield

- Azure-based cloud service to find security bugs in x86 windows binary
- Based on concolic testing techniques of SAGE
2016 Aug DARPA Cyber Grand Challenge - the world’s 1st all-machine hacking tournament

- Each team’s Cyber Reasoning System automatically identifies security flaws and applies patches to its own system in a hack-and-defend style contest targeting a new Linux-based OS DECREE
- Mayhem won the best score, which is CMU’s concolic testing based tool
Case Study: Busybox

• We test a busybox by using CREST.
  – BusyBox is a one-in-all command-line utilities providing a fairly complete programming/debugging environment
  – It combines tiny versions of ~300 UNIX utilities into a single small executable program suite.
  – Among those 300 utilities, we focused to test the following 10 utilities
    • `grep`, `vi`, `cut`, `expr`, `od`, `printf`, `tr`, `cp`, `ls`, `mv`.
    • We selected these 10 utilities, because their behavior is easy to understand so that it is clear what variables should be declared as symbolic
    • Each utility generated 40,000 test cases for 4 different search strategies
## Busybox Testing Result

<table>
<thead>
<tr>
<th>Utility</th>
<th>LOC</th>
<th># of branches</th>
<th># of covered branches</th>
<th>DFS # of covered branch/time</th>
<th>CFG # of covered branch/time</th>
<th>Random # of covered branch/time</th>
<th>Random input # of covered branch/time</th>
<th>Merge of all 4 strategies # of covered branch/time</th>
</tr>
</thead>
<tbody>
<tr>
<td>grep</td>
<td>914</td>
<td>178</td>
<td>105(59.0%) 2785s</td>
<td>85(47.8%) 56s</td>
<td>136(76.4%) 85s</td>
<td>50(28.1%) 45s</td>
<td>136(76.4%)</td>
<td></td>
</tr>
<tr>
<td>vi</td>
<td>4000</td>
<td>1498</td>
<td>855(57.1%) 1495s</td>
<td>965(64.4%) 1036s</td>
<td>1142(76.2%) 723s</td>
<td>1019(68.0%) 463s</td>
<td>1238(82.6%)</td>
<td></td>
</tr>
<tr>
<td>cut</td>
<td>209</td>
<td>112</td>
<td>67(59.8%) 42s</td>
<td>60(53.6%) 45s</td>
<td>84(75.0%) 53s</td>
<td>48(42.9%) 45s</td>
<td>90(80.4%)</td>
<td></td>
</tr>
<tr>
<td>expr</td>
<td>501</td>
<td>154</td>
<td>104(67.5%) 58s</td>
<td>101(65.6%) 44s</td>
<td>105(68.1%) 50s</td>
<td>48(31.2%) 31s</td>
<td>108(70.1%)</td>
<td></td>
</tr>
<tr>
<td>od</td>
<td>222</td>
<td>74</td>
<td>59(79.7%) 35s</td>
<td>72(97.3%) 41s</td>
<td>66(89.2%) 42s</td>
<td>44(59.5%) 30s</td>
<td>72(97.3%)</td>
<td></td>
</tr>
<tr>
<td>printf</td>
<td>406</td>
<td>144</td>
<td>93(64.6%) 84s</td>
<td>109(75.7%) 41s</td>
<td>102(70.8%) 40s</td>
<td>77(53.5%) 30s</td>
<td>115(79.9%)</td>
<td></td>
</tr>
<tr>
<td>tr</td>
<td>328</td>
<td>140</td>
<td>67(47.9%) 58s</td>
<td>72(51.4%) 50s</td>
<td>72(51.4%) 50s</td>
<td>63(45%) 42s</td>
<td>73(52.1%)</td>
<td></td>
</tr>
<tr>
<td>cp</td>
<td>191</td>
<td>32</td>
<td>20(62.5%) 38s</td>
<td>20(62.5%) 38s</td>
<td>20(62.5%) 38s</td>
<td>17(53.1%) 30s</td>
<td>20(62.5%)</td>
<td></td>
</tr>
<tr>
<td>ls</td>
<td>1123</td>
<td>270</td>
<td>179(71.6%) 87s</td>
<td>162(64.8%) 111s</td>
<td>191(76.4%) 86s</td>
<td>131(52.4%) 105s</td>
<td>191(76.4%)</td>
<td></td>
</tr>
<tr>
<td>mv</td>
<td>135</td>
<td>56</td>
<td>24(42.9%) 0s</td>
<td>24(42.9%) 0s</td>
<td>24(42.9%) 0s</td>
<td>17(30.3%) 0s</td>
<td>24(47.9%)</td>
<td></td>
</tr>
<tr>
<td>AVG</td>
<td>803</td>
<td>264</td>
<td>157.3(59.6%) 809s</td>
<td>167(63.3%) 146s</td>
<td>194.2(73.5%) 117s</td>
<td>151.4(57.4%) 83s</td>
<td>206.7(78.4%)</td>
<td></td>
</tr>
</tbody>
</table>
Result of grep

Experiment 1:
Iterations: 10,000
branches in grep.c: 178
Execution Command:

run_crest './busybox grep "define" test_grep.dat' 10000 -dfs
run_crest './busybox grep "define" test_grep.dat' 10000 -cfg
run_crest './busybox grep "define" test_grep.dat' 10000 -random
run_crest './busybox grep "define" test_grep.dat' 10000 -random_input

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Time cost (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dfs</td>
<td>2758</td>
</tr>
<tr>
<td>Cfg</td>
<td>56</td>
</tr>
<tr>
<td>Random</td>
<td>85</td>
</tr>
<tr>
<td>Pure_random</td>
<td>45</td>
</tr>
</tbody>
</table>
Test Oracles

- In the busybox testing, we do not use any explicit test oracles
  - Test oracle is an orthogonal issue to test case generation
  - However, still violation of runtime conformance (i.e., no segmentation fault, no divide-by-zero, etc) can be checked

- Segmentation fault due to integer overflow detected at grep 2.0
  - This bug was detected by test cases generated using DFS
  - The bug causes segmentation fault when
    - -B 1073741824 (i.e. 2^32/4)
    - PATTERN should match line(s) after the 1st line
    - Text file should contain at least two lines
  - Bug scenario
    - Grep tries to dynamically allocate memory for buffering matched lines (-B option).
    - But due to integer overflow (# of line to buffer * sizeof(pointer)), memory is allocated in much less amount
    - Finally grep finally accesses illegal memory area
**Bug 2653** - busybox grep with option -B can cause segmentation fault

**Status:** RESOLVED FIXED  
**Reported:** 2010-10-02 06:35 UTC by Yunho Kim  
**Modified:** 2010-10-03 21:50 UTC  
**Component:** Other  
**Version:** 1.17.x  
**Platform:** PC Linux  
**Assigned To:** unassigned  
**Importance:** P5 major  
**Target Milestone:** ---  
**Bug List:** 1 user (show)  

**Host:**  
**Target:**  
**Build:**

---

**Attachments**

Add an attachment (proposed patch, testcase, etc.)

---

Note: You need to log in before you can comment on or make changes to this bug.

Yunho Kim 2010-10-02 06:35:09 UTC

I report an integer overflow bug in a busybox grep applet, which causes an memory corruption.

**** findutils/grep.c ****

634 if (option_mask32 & OPT_C) {  
635 /* -C unsets prev -A and -B, but following -A or -B  
636 may override it */  
637 if (!((option_mask32 & OPT_A)) /* not overridden */  
638     lines_after = Copt;  
639 if (!((option_mask32 & OPT_B)) /* not overridden */  
640     lines_before = Copt;

---

- Bug patch was immediately made in 1 day, since this bug is critical one
  - Importance: P5 major
  - major loss of function
  - Busybox 1.18.x will have fix for this bug
Future Direction

• Tool support will be strengthened for automated SW analysis
  – Ex. CBMC, BLAST, CREST, KLEE, and Microsoft PEX
  – Automated SW analysis will be performed routinely like GCC
  – Labor-intensive SW analysis => automated SW analysis by few experts

• Supports for concurrency analysis
  – Deadlock/livelock detection
  – Data-race detection

• Less user input, more analysis result and less false alarm
  – Fully automatic C++ syntax & type check (1980s)
  – (semi) automatic null-pointer dereference check (2000s)
  – (semi) automatic user-given assertion check (2020s)
  – (semi) automatic debugging (2030s)