Analysis on Unit-level Concolic Testing for Real-world C Programs

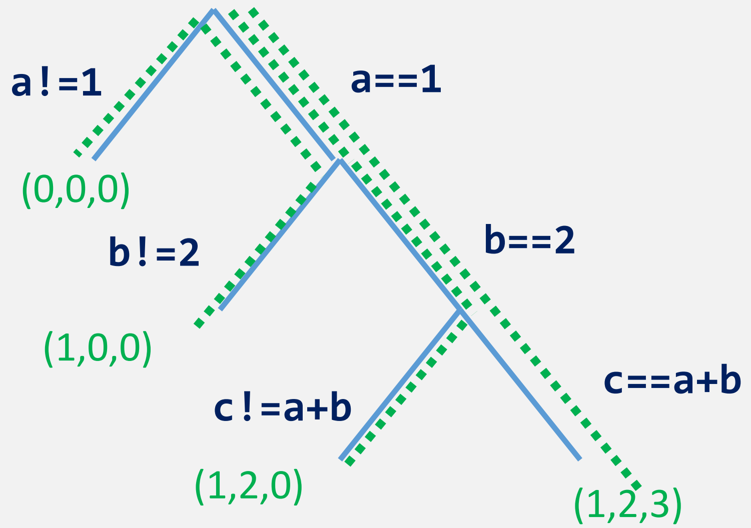
Zidong Yang
School of Computing – KAIST
Advisor: Prof. Moonzoo Kim

December 15th 2021

Background
Concolic Unit Testing

- Automatically generating test cases to cover all execution paths of target function.

```
// Symbolic input: a, b, c
void f(int a, int b, int c) {
  if (a == 1)
    if (b == 2)
      if (c == a + b)
        Error();
}
```



- Concolic unit testing will generate 4 test cases for the function **f**
  - First TC: (0,0,0)
    - SPF (symbolic path formula): `a!=1`
    - Next SPF: `!(a!=1)`
  - Next TC: (1,0,0)
    - SPF: `a==1 && b!=2`
    - Next SPF: `a==1 && !(b!=2)`
  - Next TC: (1,2,0)
    - SPF: `a==1 && b==2 && c!=a+b`
    - Next SPF: `a==1 && b==2 && c==a+b`
  - Next TC: (1,2,3)
    - All the paths are covered

Related Work

- The following table lists the unit-level test case generation techniques.

Tool	Test Case Generation Approach	Driver?	Stub?	Support Local Static Variable	Comb. SS	Random Value Gen If 1st TC Crash	Support Func Pointer	Stub for File-handling Func	Utilize different testing engine
CUTE	Concolic Testing	✓	✗	✗	✗	✗	✗	✗	✗
UC-KLEE	Symbolic Execution	✓	✗	✗	✓	✗	✗	✗	✗
CONBOL	Concolic Testing	✓	✓	✗	✗	✗	✗	✓	✗
CONBRIO	Concolic Testing	✓	✓	✗	✗	✗	✗	✓	✗
MAESTRO	Concolic Testing , Fuzzing	✓	✓	✓	✓	✗	✓	✗	✓
CR2^{imp}	**Concolic Testing**	✓	✓	✓	✓	✓	✓	✓	✓

Motivation

- In principle, concolic unit testing can achieve 100% branch coverage.
- The experimental results (branch coverage) on real-world subjects are not as good as expected.
  - The following table shows the branch coverage achieved by concolic unit testing.

Subject	# Branch	Branch Coverage
flex2.4.3	2,021	22.6%
grep2.0	3,416	22.7%

The detailed experimental settings and results will be shown later.

- Concolic unit testing may have some limitations that prevent it from achieving high branch coverage.
  - To discover these limitations, we need to analyze the not-covered branches of the real-world subjects.

Thesis Statement

Concolic testing can be improved to achieve high branch coverage by addressing its limitations identified through **systematic coverage analysis** on real-world subjects.

Systematic coverage analysis:

1. Obtain the coverage achieved by concolic testing.
2. For each target real-world subject, find all the not-covered branches.
3. Analyze each not-covered branch to discover the limitations of concolic unit testing.

- This dissertation performs coverage analysis on the result of CROWN2.0 ver 2020, a popular commercial concolic unit testing tool.

Contributions

- I highlight **13 common problems** of CROWN2.0 ver 2020 by extensively analyzing 324 groups of not-covered branches on 6 different widely-used target subjects.
  - flex, grep, gzip, sed, libquantum, and sjeng
- I propose and implement **6 ideas that address 8/13 common problems**
  - My solutions improved branch coverage of CROWN 2.0 ver 2020 by up to 188% on a target subject (i.e., sed) and by 77% on average on the 6 subjects.
- I provide a detailed report to explain the reasons of 324 not-covered branch groups.
  - The detailed report can contribute to the concolic testing research.

CROWN2.0: Commercial Concolic Unit Testing Tool (1/2)

- CR2 (CROWN2.0) automatically generates driver/stubs for the unit testing of a target function.
  - The driver symbolically sets all global variables and parameters of the target function.

Type	Description	Code Example

Primitive	set a corresponding symbolic value	`int a;`
		`SYM_int(a);`

Array	set all elements of the array	`int a[3];`
		`for(int i=0; i<3; i++) {SYM_int(a[i]);}`

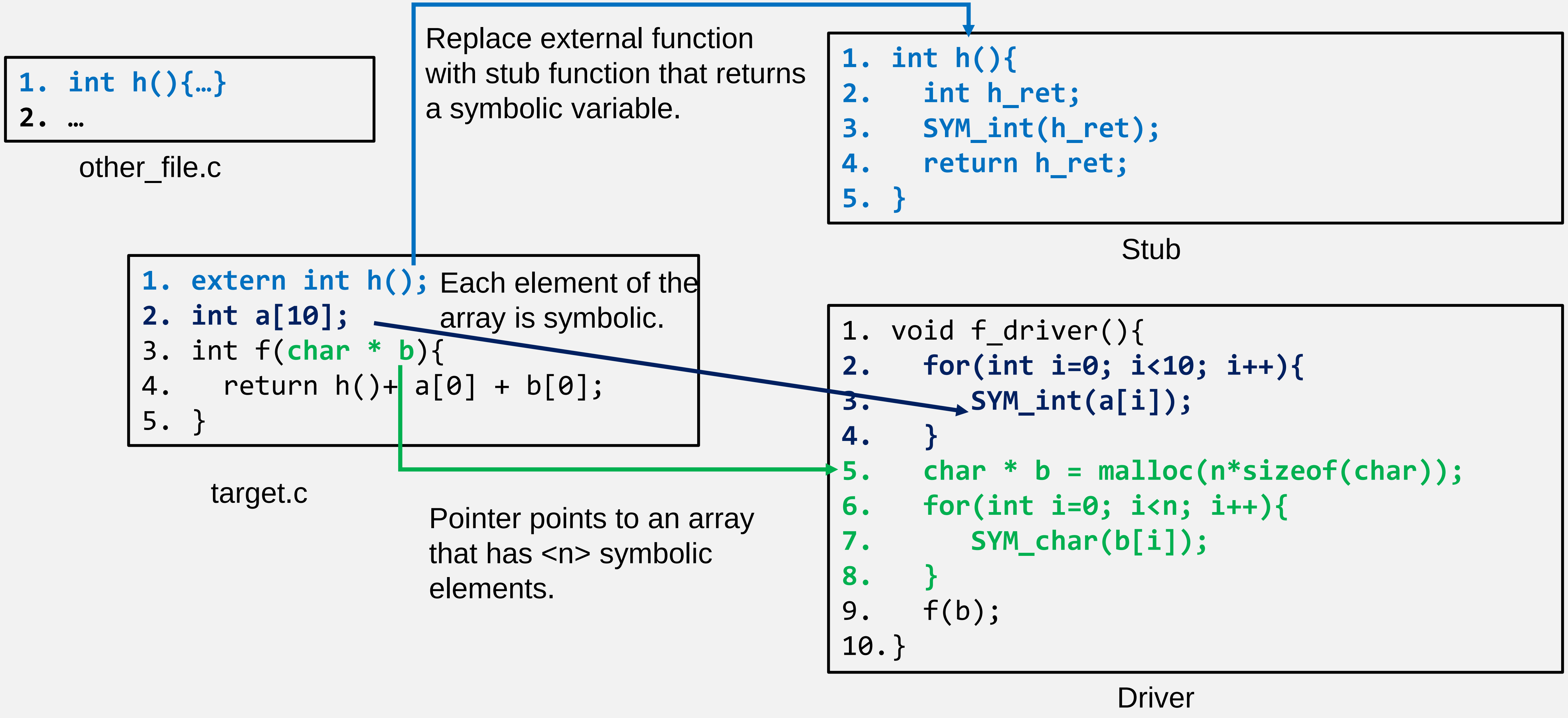
Pointer	allocate an array with **n** symbolic pointee elements and make the pointer point to the array (**n** is decided by users)	`int *a;`
		`a = malloc(n* sizeof(int));`
		`for(int i=0; i<n; i++) {SYM_int(a[i]);}`

Structure	set symbolic for each field of the structure	`struct _st{int e1,char *e2} a;`
		`SYM_int(a.e1);`
		`a.e2 = malloc(n* sizeof(char));`
		`for(int i=0; i<n; i++) {SYM_char(a.e2[i]);}`

- CR2 replaces the external function (i.e., the function defined in other C file) with stub function.
- Example is provided in the next page.

CROWN2.0: Commercial Concolic Unit Testing Tool (2/2)

- CR2 replaces the external functions invoked by the target function with stubs.



```c
int h(){…}
…
```

other_file.c

Replace external function with stub function that returns a symbolic variable.

```c
int h(){
  int h_ret;
  SYM_int(h_ret);
  return h_ret;
}
```

Stub

```c
extern int h();
int a[10];
int f(char * b){
  return h()+ a[0] + b[0];
}
```

Each element of the array is symbolic.

target.c

Pointer points to an array that has <n> symbolic elements.

```c
void f_driver(){
  for(int i=0; i<10; i++){
     SYM_int(a[i]);
  }
  char * b = malloc(n*sizeof(char));
  for(int i=0; i<n; i++){
     SYM_char(b[i]);
  }
  f(b);
}
```

Driver

Target Subjects

- This part introduces the experimental settings and experimental results of the baseline CR2 (version 2020).
    - The coverage analysis is **based on the baseline results**.
- This paper targets six subjects (four from SIR and two from SPEC2006).
    - Four SIR subjects are frequently used by Linux users.
    - I choose two SPEC2006 subjects to avoid making the analysis results overfitted to one benchmark.

Benchmark	Subject	Loc	# Branch	# Function
SIR	flex2.4.3	11,864	2,021	144
	grep 2.0	10,930	3,416	119
	gzip1.0.7	6,358	1,446	80
	sed1.17	8,060	2,395	63
SPEC2006	libquantum0.2.4	5,059	724	111
	sjeng11.2	18,057	6,364	164

Experimental Setup
Baseline Technique (CR2 ver 2020)

- The experimental settings of baseline CR2 are as follows.
  - Array size: 3
    - The # elements of an allocated array that a pointer points to.
  - Strategy: dfs (depth-first-search)
    - The search heuristic of concolic testing (e.g., dfs, rev-dfs, random)
  - Timeout1: 60s
    - The maximum execution time of a test case.
  - Timeout2: 300s
    - The test case generation time for each function.
  - Target Subjects: Six subjects
    - 4 from SIR benchmark and 2 from SPEC2006 benchmark.
  - Machine: Ubuntu 18.04 with AMD 8-core Ryzen 7 3800 XT (3.9GHz) and 16 GB RAM

Baseline Results

- The following table shows the branch coverage achieved by the baseline technique (i.e., CR2 ver 2020)

Benchmark	Subject	Loc	# Branch	# Function	Branch Coverage	# Function that Reach Timeout2	Avg TCgen Time (s) for Each Func
SIR	flex2.4.3	11,864	2,021	144	**22.6%**	**20**	**15.3**
	grep 2.0	10,930	3,416	119	**22.7%**	**9**	**27.3**
	gzip1.0.7	6,358	1,446	80	**41.4%**	**9**	**52.7**
	sed1.17	8,060	2,395	63	**17.7%**	**1**	**15.8**
SPEC2006	libquantum0.2.4	5,059	724	111	**41.6%**	**5**	**5.5**
	sjeng11.2	18,057	6,364	164	**27.9%**	**20**	**26.7**

Avg: **28.9%**

The branch coverage achieved by baseline CR2 on average for 6 subjects is poor

Analysis on Baseline Results

- To discover the limitations that make CR2 only achieve 28.9% branch coverage on average, I performed an analysis for each subject.

STEP 01

Download Coverage Report

Download the coverage report of each target subject

STEP 02

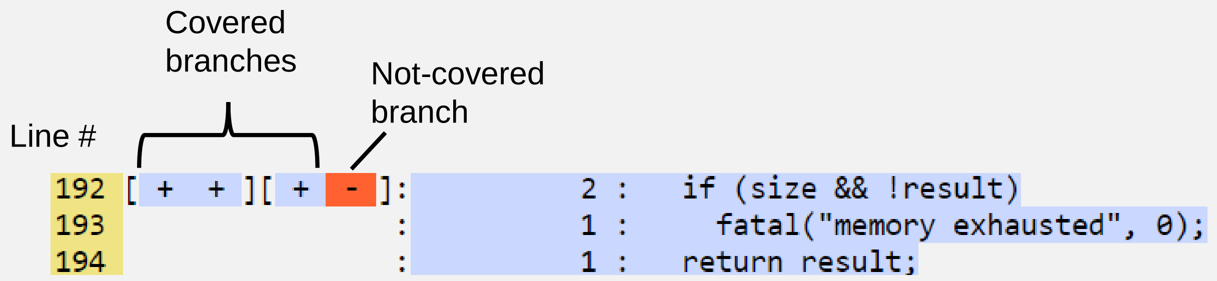
Check Coverage Report

Check the not-covered branches and discover the reasons

STEP 03

Summarize Limitations

Summarize and list all the limitations



```
192 [ +  + ][ + - ]:          2 :   if (size && !result)
193                 :          1 :     fatal("memory exhausted", 0);
194                 :          1 :   return result;
```

Snapshot of coverage report

Limitations Overview

- After analyzing the not-covered branches of six subjects, 13 limitations of CR2 are discovered.
- The table on the right shows the information of each limitation.

Abbreviation	Description
FTCC	First TC crashes
NSEF	No support for external C library functions (except for C file handling functions)
NSFF	No support for external file-handling functions
NSFP	No support for function pointers
NSGP	No support for global void pointer
NSLS	No support for local static variable
NSSF	No support for symbolic file
NSSP	No support for symbolic pointers
RT1	Timeout1 is reached
TIWC	Target program's Illegal write to the structure of CROWN
UB	Existence of unreachable branches in unit-testing
USV	Uncovered branch caused by the unrealistic symbolic input values
WDFS	Weakness of dfs

Limitations
NSLS: No support for Local Static Variables

- CR2 cannot set local static variables as symbolic, and the unit driver calls the target function only once.
  - This feature may produce some not-covered branches.

```
f1_driver(){
  ...
  f1();
}
```

The driver code calls `f1` **once**.

Solution: Execute `f1` twice

```
1. void f1(){
2.    static int var = 0;
3.    if(var == 1){
4.       ... // not-covered branches
5.    }else{
6.       ... // f1 is not called
7.    }
8.    var += 1;
9. }
```

nsls.c

The local static variable `var` is initialized with **0**

CR2 cannot cover line 4 because the value of `var` is 0

The value of var is updated, so the 2nd execution of f2 can cover line 4.

Report Example
NSLS: No support for Local Static Variables

Subject: Sed

File Name: `sed.c`
Function Name: `init_syntax_once`
Reason: **NSLS**
Line No: 2302
Description:

1. Line 2299: done is a **local static variable**, and it is assigned with 0
2. Line 2301: the "then" branch cannot be reached as the value of done is 0
3. Line 2317: done is set to 1, which means the **second execution** of the target function (i.e., `init_syntax_once`) can cover line 2302
   - CR2 executes the target function only **once** for each TC, so line 2302 cannot be covered.

```
2295. static void
2296. init_syntax_once()
2297. {
        ...
2299.   static int done = 0;
2300.
2301.   if(done)
2302.     return;       // not-covered
        ...            // done is not altered
2317.   done = 1;
2318. }
```

Source code of function `init_syntax_once`

Limitations
WDFS: Weakness of Dfs

Solution: use rev-dfs

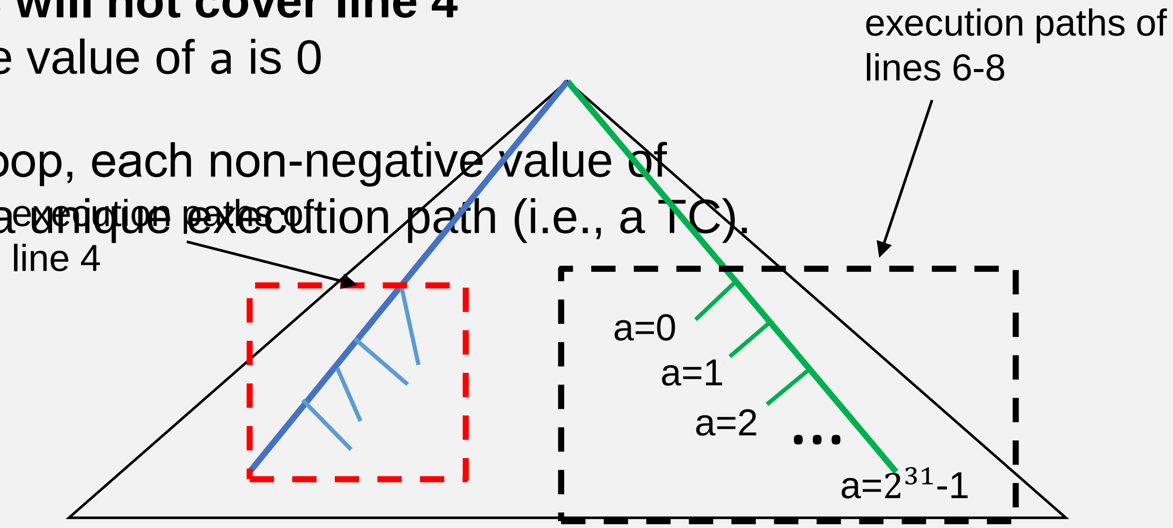
```
 1. long f2(int a){
 2.    long b = 0;
 3.    if(a < 0){
 4.      ... // not-covered branches
 5.    }
 6.    for(int i=0; i<a; i++){
 7.      b += i;
 8.    }
 9.    return b;
10. }
```

wdfs.c

The symbolic variable a is assigned with **0** for the first test case

The first TC **will not cover line 4** because the value of a is 0

In the "for" loop, each non-negative value of a produces a unique execution path (i.e., a TC).



execution paths of f2

- **Path explosion** problem happens because the dfs strategy has to generate 2^{31} test cases before covering line 4.
  - Test case generation timeout is reached.

Report Example
WDFS: Weakness of DFS

Subject: Sjeng

File Name: `attacks.c`
Function Name: `nk_attacked`
Reason: **WDFS**
Line No: 461-498
Description:

1. Line 460: `color` is a symbolic integer, and it is assigned with 0, the “else” branch is explored first.
2. Lines 503-515 have **huge possible execution paths** (i.e., at least $2_{<L508>} * (2_{<L510>} * 4_{<L511>} * 2_{<L512>})^{13}$ unique paths)
3. Dfs strategy has to explore all the paths mentioned above before exploring lines 461-498. `Timeout2` will be reached.

```
38. int board[144]; // symbolic array
…
448. bool nk_attacked(int square, int color){
       ...
453.   static const int bishop_o[4] = {11,-11,13,-13};
       ...
460.   if (color&1){
         ...   // not-covered
498.   }
499.
500.
501.   else{
         ...
508.     if(basq=bpawn && ...) return true;
509.     // a_sq = 0, ndir = 11 initially
510.     while(basq != frame){
511.       if(basq == bbishop || basq == bqueen) {...}
512.       if(basq != npiece) {...}
513.       a_sq += ndir;
514.       basq = board[a_sq]; // board[0,11,22,…,143]
515.     }
516.   }
       ...
542. }
```

Source code of function nk_attacked

Solutions Overview

- I provide 6 ideas to overcome the limitations mentioned previously.

01 Execute target function **multiple times**

Target: NSLS

02 Implement **combined** strategy

Target: WDFS

03 **Random values** for symbolic inputs

Target: FTCC

04 **Static analysis** for function pointers

Target: NSFP

05 Stub for all **file-handling functions**

Target: NSFF

06 Utilize **other concolic testing tool**

Targets: NSEF, NSSP, USV

Solutions
NSLS => Execute Target Function Multiple Times

- This approach allows the users to decide **the number of calls (M)** to the target function.

```c
1. void f1(){
2.   static int var = 0;
3.   if(var == 1){
4.     ... //         Covered
5.   }else{
6.     ... // f1 is not called
7.   }
8.   var += 1;
9. }
```

nsls.c

Line 4 can be covered if **M** is greater than 2

```c
1. f1_driver(){
2.     ...
3.   for(int i = 0; i < M; i++){
4.     f1();
5.   }
6. }
```

driver.c

Solutions
WDFS => Implement Combined Strategy

- The combined strategy is to **overcome the limitation of dfs strategy**.
  - It integrates **four search strategies** (i.e., dfs, rev-dfs, cfg, random) of CROWN and works as follows.

```
Input: $T$: timeout2 which is giver by the user
Output: a sequence of TCs
1: $t \leftarrow \frac{T}{4}$
2: $TC1 \leftarrow Run(dfs, t)$
3: if CROWN finishes in $t$ then
4:     $TC \leftarrow TC1$
5: else
6:     $TC2 \leftarrow Run(revdfs, t)$
7:     $TC3 \leftarrow Run(random, t)$
8:     $TC4 \leftarrow Run(cfg, t)$
9:     $TC \leftarrow TC1 + TC2 + TC3 + TC4$
10: end if
11: return $TC$
```

The users give the test case generation timeout T

Generate test case using dfs strategy in $T/4$

If dfs strategy finishes in $T/4$, we assume CROWN has explored all the paths of the target function, and stop the TC generation process.

Otherwise, we assume dfs stuck in the complex statements (e.g., loop statement).
We run the remaining 3 search strategies in order, each strategy uses $T/4$

Accumulate all the test cases generated by the combined strategy.

Evaluation
Experimental Results

- I developed CR2^{imp} to intergrade the six ideas mentioned above.
- The experimental settings of CR2^{imp} are identical to the ones of baseline CR2.

Benchmark	Subject	Loc	# Branch	# Function	Branch Coverage (CR2)	Avg TCgen Time (s) for Each Func (CR2)	Branch Coverage (CR2^{imp})	Avg TCgen Time (s) for Each Func (CR2^{imp})	# Crash TCs
SIR	flex2.4.3	11,864	2,021	144	22.6%	15.3	**53.2%**	**122.7**	**4,341**
	grep 2.0	10,930	3,416	119	22.7%	27.3	**52.7%**	**56.7**	**9,172**
	gzip1.0.7	6,358	1,446	80	41.4%	52.7	**50.1%**	**98.9**	**10,359**
	sed1.17	8,060	2,395	63	17.7%	15.8	**51.1%**	**54.9**	**10,315**
SPEC2006	libquantum0.2.4	5,059	724	111	41.6%	5.5	**50.4%**	**9.8**	**3,392**
	sjeng11.2	18,057	6,364	164	27.9%	26.7	**50.1%**	**93.7**	**19,477**
				Avg:	28.9%	23.9	**51.3%**	**72.8**	

CR2^{imp} improved the branch coverage relatively by 77% on average for six subjects.

CR2^{imp} reports more than 40k crash TCs.

Crash Deduplication

- CR2^{imp} reports more than 40k crashes.
  - Manually analyzing all the crashes is impractical and time-consuming.
- Many crashes may have **the same execution path** (i.e., they are duplicate crashes).
- I developed a crash deduplication approach, which **compares the first K stack frames** of the crashes to remove the duplicate crashes.
  - Crash deduplication results are shown in the following table.

Target Subject	# Crash TC	# Unique Crash TC (K=3)	# Unique Crash TC (K=5)
flex2.4.3	4,341	123	124
grep 2.0	9,172	241	248
gzip1.0.7	10,359	56	56
sed1.17	10,315	166	167
libquantum0.2.4	3,392	289	289
sjeng11.2	19,477	581	579

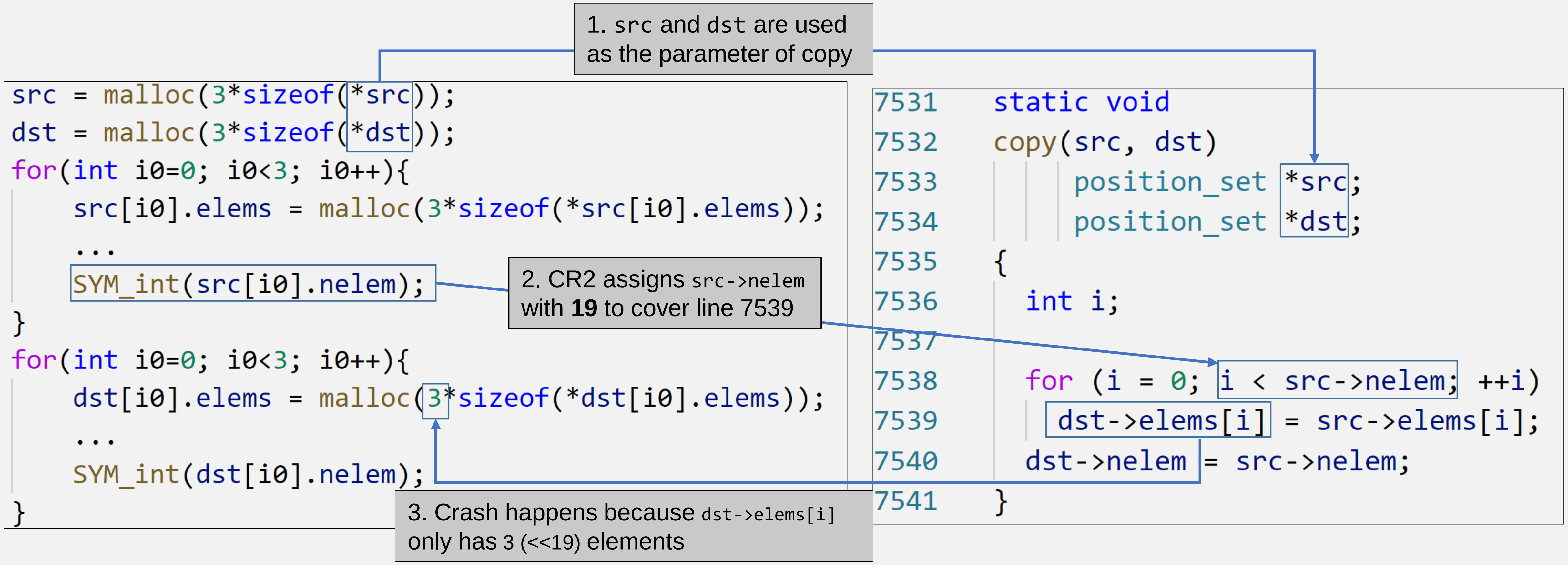
I analyzed these crashes to find the root cause of them.

Crash Analysis

- After analyzing the 595 crashes, three causes of false alarms are identified.
    - **USIV**. Unrealistic Symbolic Input Value
    - **USVI**. Unrealistic Symbolic Variable Initialization
    - **ICIO**. Invalid Comparison Caused by Integer Overflow

Crash Analysis
USIV: Unrealistic Symbolic Input Values

- The following is a false alarm example, which is caused by USIV, of subject grep.



1. `src` and `dst` are used as the parameter of copy

```
src = malloc(3*sizeof(*src));
dst = malloc(3*sizeof(*dst));
for(int i0=0; i0<3; i0++){
    src[i0].elems = malloc(3*sizeof(*src[i0].elems));
    ...
    SYM_int(src[i0].nelem);
}
for(int i0=0; i0<3; i0++){
    dst[i0].elems = malloc(3*sizeof(*dst[i0].elems));
    ...
    SYM_int(dst[i0].nelem);
}
```

2. CR2 assigns `src->nelem` with **19** to cover line 7539

3. Crash happens because `dst->elems[i]` only has 3 (<<19) elements

```
7531    static void
7532    copy(src, dst)
7533            position_set *src;
7534            position_set *dst;
7535    {
7536      int i;
7537
7538      for (i = 0; i < src->nelem; ++i)
7539        dst->elems[i] = src->elems[i];
7540      dst->nelem = src->nelem;
7541    }
```

Driver code of function copy (grep)

Source code of function copy (grep)

Conclusion

- Discovered **13 limitations** of CR2 through **systematic coverage analysis** on six subjects.
- Proposed **six ideas** to address the discovered limitations
- Developed CR2^{imp} that integrates the six ideas and improved the branch coverage **relatively by 77%** on average

Future Work

- More analysis on the detected crashes.
- Performing experiment on more subjects.
- Integrating other testing tools (e.g., AFL).
- Obtaining seed TC from system level TC to guide concolic testing

Q&A

THANK YOU FOR WATCHING

Crash Analysis
USVI: Unrealistic Symbolic Variable Initialization

- The driver code will ignore some useful statements in the target source code.
  - For example, a global pointer variable b is initialized with an array a which has 100 elements (L2), but the driver code just ignores this important information and make b **point to a new array** which has 3 elements only (L6).

```
1. int a[100];
2. int *b = a;
3. void target(){…}
```

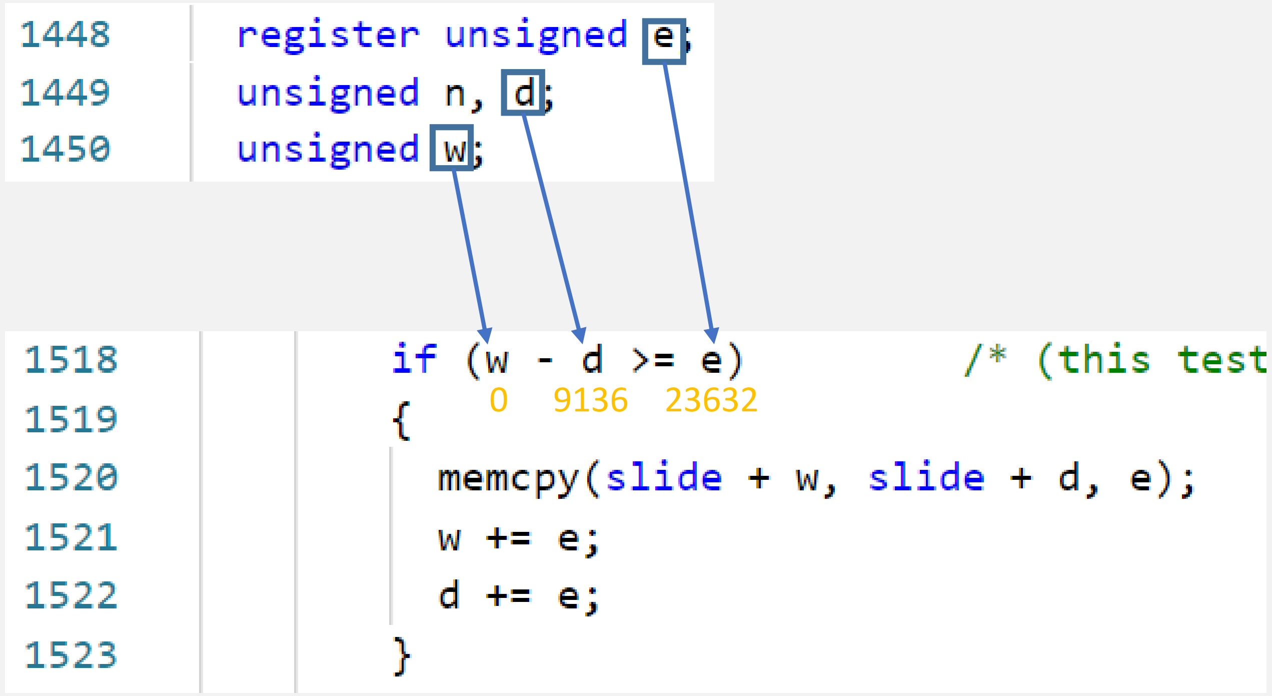
Source code

```
5. void target(){
6.   b = malloc(3*sizeof(int));
7.   ...
8. }
```

Driver code

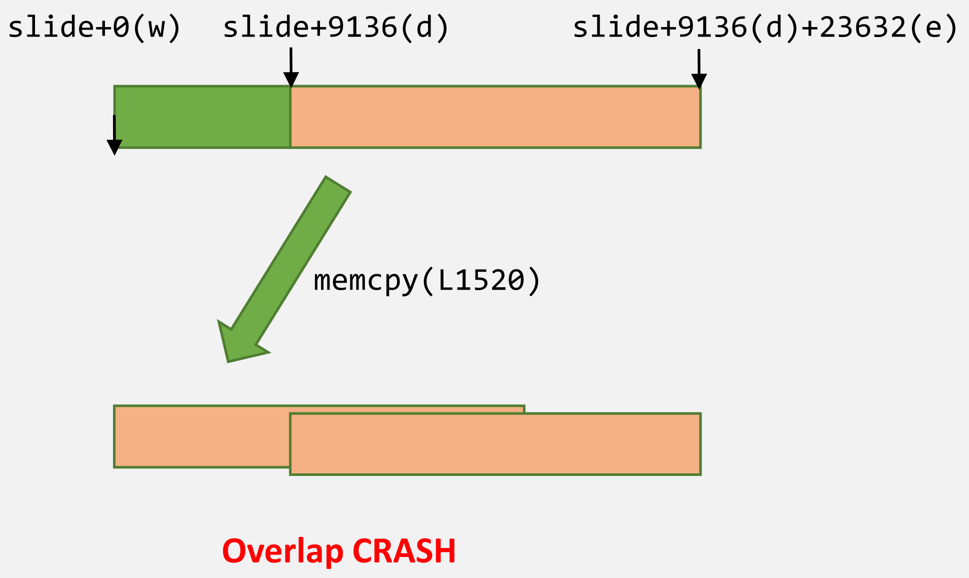
Crash Analysis
ICIO: Invalid Comparison Caused by Integer Overflow

- Integer overflow may cause some **unexpected branches** to be covered **under error conditions**, causing some crashes when executing that branch.



```
1448    register unsigned e;
1449    unsigned n, d;
1450    unsigned w;
```

```
1518          if (w - d >= e)         /* (this test
1519          {
1520            memcpy(slide + w, slide + d, e);
1521            w += e;
1522            d += e;
1523          }
```



`memcpy(des, src, n)`
Functionality: Copy n bytes from `src` to `des`.
Crash: if **`des+n>src`**, memory overlap crash will happen

- L1518: both `w`, `d` and `e` are **unsigned** variable, so (unsigned)(0-9136) =4294958160 > 23632 and the "then" branch is executed.
- L1520: **memory overlap crash** happens when executing `memcpy`.