Symbolic Execution

Kevin Wallace, CSE504

2010-04-28

Problem

- Attacker-facing code must be written to guard against all possible inputs
- There are many execution paths; not a single one should lead to a vulnerability
- Current techniques are helpful, but have weaknesses

Symbolic Execution

- Insight: code can generate its own test cases
- Run program on 'symbolic' input
- When execution path diverges, fork, adding constraints on symbolic values
- When we terminate (or crash), use a constraint solver to generate concrete input

Advantages

- Tests many code paths
- Generates concrete attacks
- Zero false positives

Fuzzing

- Idea: randomly apply mutations to well-formed inputs, test for crashes or other unexpected behavior
- Problem: usually, mutations have very little guidance, providing poor coverage
  - if(x == 10) bug(); -- fuzzing has a 1 in 2^{32} chance of triggering a bug

Today

- EXE
  - Fast - uses a custom constraint-to-SAT converter (STP)
- Whitebox fuzz testing (SAGE)
  - Targeted execution - focuses search around a user-provided execution path

EXE: Automatically Generating Inputs of Death

Using EXE

- Mark which regions of memory hold symbolic data
- Instrument code with exe-cc source-to-source translator
- Compile instrumented code with gcc, run

```
#include <assert.h>
int main(void) {
  unsigned i, t, a[4] = { 1, 3, 5, 2 };
  make_symbolic(&i);
  if(i >= 4)
    exit(0);
  // cast + symbolic offset + symbolic mutation
  char *p = (char *)a + i * 4;
  *p = *p − 1; // Just modifies one byte!

  // ERROR: EXE catches potential overflow i=2
  t = a[*p];
  // At this point i != 2.

  // ERROR: EXE catches div by 0 when i = 0.
  t = t / a[i];
  // At this point: i != 0 && i != 2.

  // EXE determines that neither assert fires.
  if(t == 2)
    assert(i == 1);
  else
    assert(i == 3);
}
```

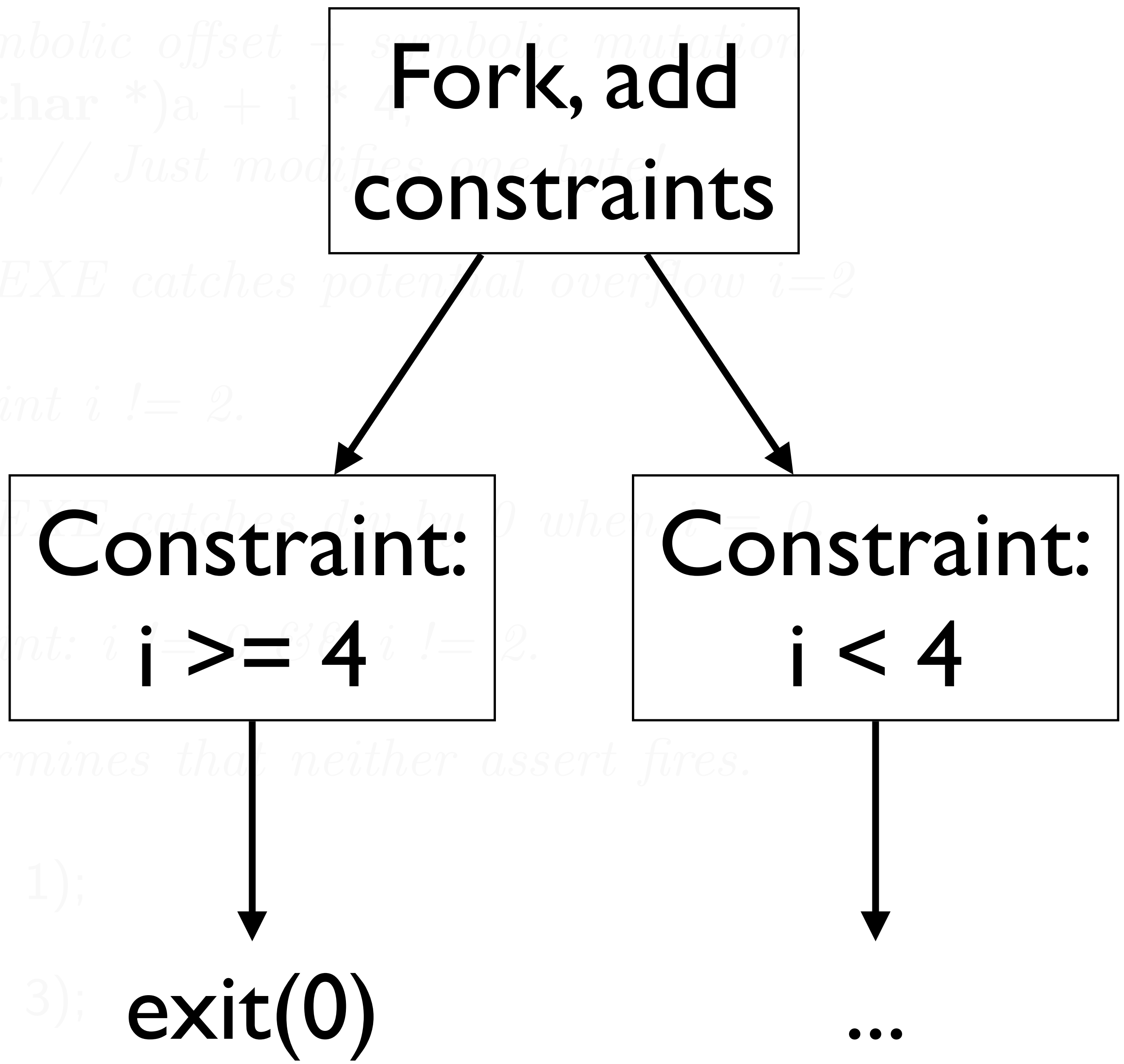
Mark *i* as symbolic

```
#include <assert.h>
int main(void) {
  unsigned i, t, a[4] = { 1, 3, 5, 2 };
  make_symbolic(&i);
  if(i >= 4)
    exit(0);
  // cast + symbolic offset + symbolic mutation
  char *p = (char *)a + i * 4;
  *p = *p − 1; // Just modifies one byte!

  // ERROR: EXE catches potential overflow i=2
  t = a[*p];
  // At this point i != 2.

  // ERROR: EXE catches div by 0 when i = 0.
  t = t / a[i];
  // At this point: i != 0 && i != 2.

  // EXE determines that neither assert fires.
  if(t == 2)
    assert(i == 1);
  else
    assert(i == 3);
}
```



```
#include <assert.h>
int main(void) {
  unsigned i, t, a[4] = { 1, 3, 5, 2 };
  make_symbolic(&i);
  if(i >= 4)
    exit(0);
  // cast + symbolic offset + symbolic mutation
  char *p = (char *)a + i * 4;
  *p = *p − 1; // Just modifies one byte!

  // ERROR: EXE catches potential overflow i=2
  t = a[*p];
  // At this point i != 2.

  // ERROR: EXE catches div by 0 when i = 0.
  t = t / a[i];
  // At this point i != 0 && i != 2.

  // EXE determines that neither assert fires.
  if(t == 2)
    assert(i == 1);
  else
    assert(i == 3);
}
```

Add constraints:
“p equals (char*)a + i * 4”
“p[0]’ equals p[0] - 1”

```c
#include <assert.h>
int main(void) {
  unsigned i, t, a[4] = { 1, 3, 5, 2 };
  make_symbolic(&i);
  if(i >= 4)
    exit(0);
  // cast + symbolic offset + symbolic mutation
  char *p = (char *)a + i * 4;
  *p = *p − 1; // Just modifies one byte!

  // ERROR: EXE catches potential overflow i=2
  t = a[*p];
  // At this point i != 2.

  // ERROR: EXE catches div by 0 when i = 0.
  t = t / a[i];
  // At this point: i != 0 && i != 2.

  // EXE determines that neither assert fires.
  if(t == 2)

  else

}
```

Could cause invalid dereference or division.
Fork, add constraints for invalid/valid cases.

```c
#include <assert.h>
int main(void) {
  unsigned i, t, a[4] = { 1, 3, 5, 2 };
  make_symbolic(&i);
  if(i >= 4)
    exit(0);
  // cast + symbolic offset + symbolic mutation
  char *p = (char *)a + i * 4;
  *p = *p − 1; // Just modifies one byte!

  // ERROR: EXE catches potential overflow i=2
  t = a[*p];
  // At this point i != 2.

  // ERROR: EXE catches div by 0 when i = 0.
  t = t / a[i];
  // At this point: i != 0 && i != 2.

  // EXE determines that neither assert fires.
  if(t == 2)
    assert(i == 1);
  else
    assert(i == 3);
}
```

Fork, add constraints.
On false branch, emit error

Using exe-cc

```
% exe-cc simple.c
% ./a.out
% ls exe-last
  test1.forks test2.out     test3.forks  test4.out
  test1.out   test2.ptr.err test3.out    test5.forks
  test2.forks test3.div.err test4.forks  test5.out
% cat exe-last/test3.div.err
  ERROR: simple.c:16 Division/modulo by zero!
% cat exe-last/test3.out
  # concrete byte values:
  0 # i[0]
  0 # i[1]
  0 # i[2]
  0 # i[3]
% cat exe-last/test3.forks
  # take these choices to follow path
  0 # false branch (line 5)
  0 # false (implicit: pointer overflow check on line 9)
  1 # true (implicit: div-by-0 check on line 16)
% cat exe-last/test2.out
  # concrete byte values:
  2 # i[0]
  0 # i[1]
  0 # i[2]
  0 # i[3]
```

Constraint solving: STP

- Insight: if memory is a giant array of bits, constraint solving can be reduced to SAT
- Idea: turn set of constraints on memory regions into a set of boolean clauses in CNF
- Feed this into an off-the-shelf SAT solver (MiniSAT)

Caveat - pointers

- STP doesn’t directly support pointers
- EXE takes a similar approach to CCured and tags each pointer with a ‘home’ region
- Double-dereferences resolved with *concretization*, at the cost of soundness

STP results

Solver	Total Time	Timeouts
CVCL	60,366s	546
STP (no optimizations)	3,378s	36
STP (substitution)	1,216s	1
STP (refinement)	624s	1
STP (simplifications)	336s	0
STP (subst+refinement)	513s	1
STP (simplif+subst)	233s	0
STP (simplif+refinement)	220s	0
STP (all optimizations)	110s	0

(Pentium 4 machine at 3.2 GHz, with 2 GB of RAM and 512 KB of cache)

EXE Results

	bpf	expat	pcre	tcpdump	udhcpd
Test cases	7333	360	866	2140	328
None	30.6	28.4	31.3	28.2	30.4
Caching	32.6	30.8	34.4	27.0	36.4
Independence	17.8	25.2	10.0	24.9	30.5
All	10.3	26.3	7.5	23.6	32.1
STP cost	6.9	24.6	2.8	22.4	23.1

(number of test cases generated, times in minutes on a dual-core 3.2 GHz Intel Pentium D machine with 2 GB of RAM, and 2048 KB of cache)

Results (detail)

		bpf	expat	pcre	tcpdump	udhcpd
1	Symbolic input size (bytes)	96	10	16	84	548
2	Total statements run (not unique)	298,195	41,345	423,182	40,097	15,258
3	% of statements symbolic	29.2%	8.5%	34.7%	41.7%	23.6% %
4	Explicit symbolic branch points	77,024	1,969	98,138	11,425	888
5	% with both branches feasible	11.3%	19.3%	0.9%	19.4%	52.8%
6	Avg. # symbolic branches per path	38.33	43.44	55.72	103.37	200.14
7	Symbolic checks	1,490	904	4,451	552	1,535
8	Pointer concretizations	0	0	0	73	0
9	Symbolic args. to uninstr. calls	0	0	0	0	0

Table 5: Dynamic counts from EXE execution runs.

Results (detail)

		bpf	expat	pcre	tcpdump	udhcpd
1	Symbolic input size (bytes)	96	10	16	84	548
2	Total statements run (not unique)	298,195	41,345	423,182	40,097	15,258
3	% of statements symbolic	29.2%	8.5%	34.7%	41.7%	23.6% %
4	Explicit symbolic branch points	77,024	1,969	98,138	11,425	888
5	% with both branches feasible	11.3%	19.3%	0.9%	19.4%	52.8%
6	Avg. # symbolic branches per path	38.33	43.44	55.72	103.37	200.14
7	Symbolic checks	1,490	904	4,451	552	1,535
8	Pointer concretizations	0	0	0	73	0
9	Symbolic args. to uninstr. calls	0	0	0	0	0

Table 5: Dynamic counts from EXE execution runs.

Search heuristics

- Need to limit the number of simultaneously running forked processes
  - (unless you like forkbombs)
- What order do we run forked processes in?
- Currently using a modified best-first search

Search heuristics

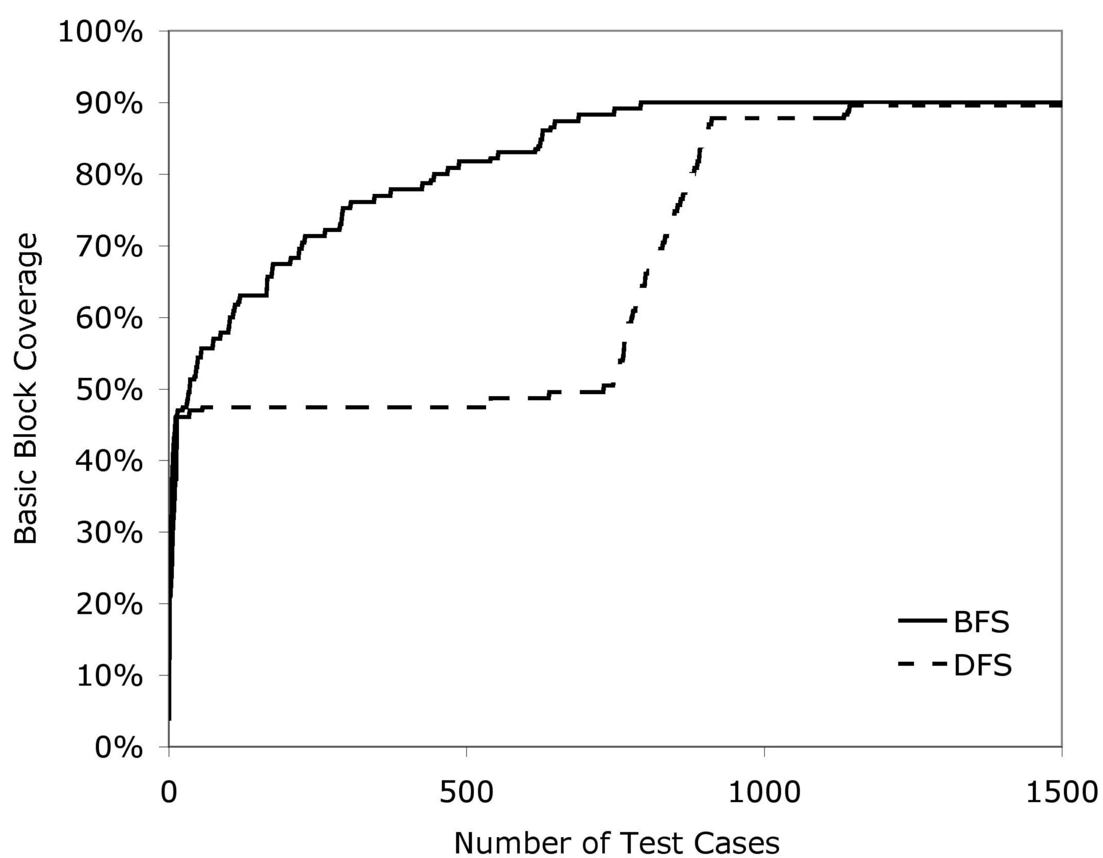


Figure 4: Best-first search vs. depth-first search.

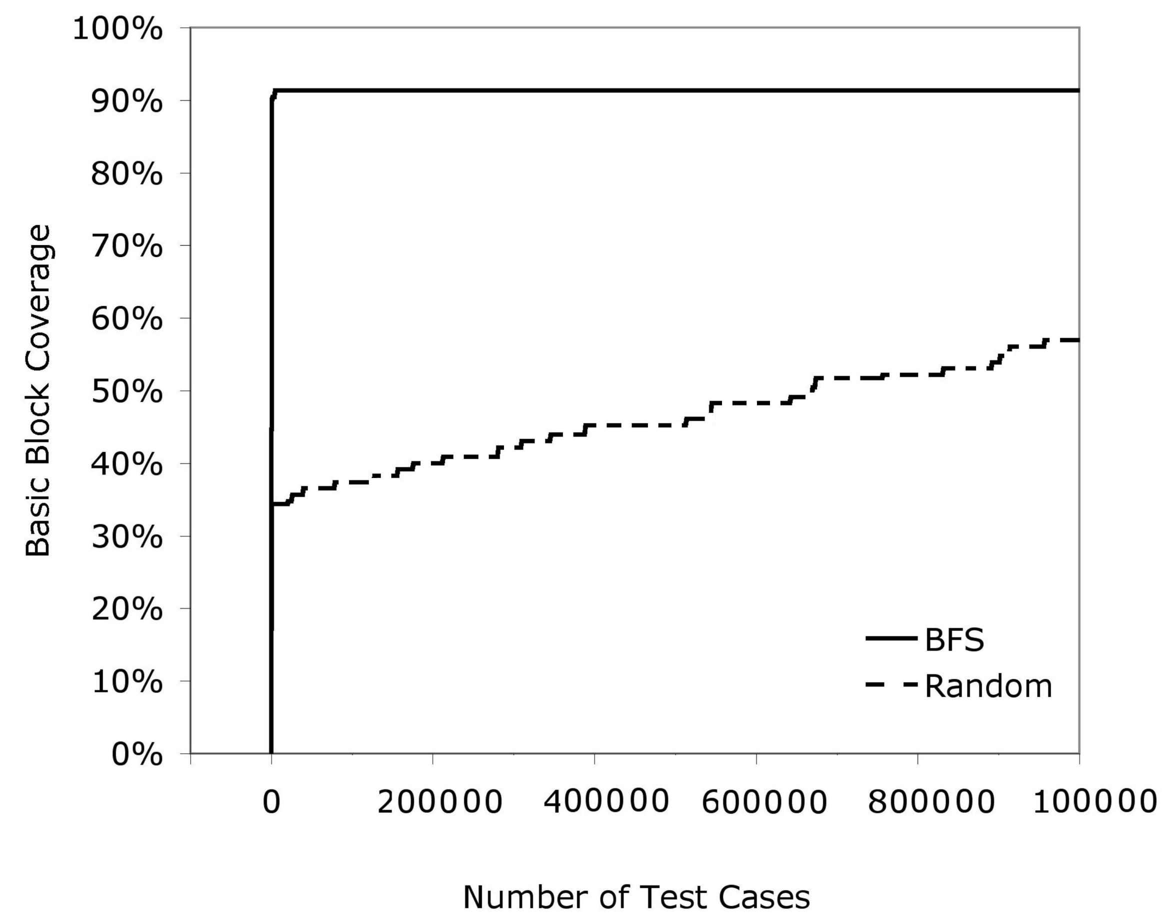


Figure 5: EXE with best-first search vs. random testing.

EXE finds real bugs

```
s[0].code = BPF_STX; // also: (BPF_LDX|BPF_MEM)
s[0].k    = 0xfffffff0UL;
s[1].code = BPF_RET;
```

Figure 6: A BPF filter of death

```
// Code extracted from bpf_validate. Rejects
// filter if opcode's memory offset is more than
// BPF_MEMWORDS.
// Forgets to check opcodes LDX and STX!
if((BPF_CLASS(p->code) == BPF_ST
  || (BPF_CLASS(p->code) == BPF_LD &&
       (p->code & 0xe0) == BPF_MEM))
  && p->k >= BPF_MEMWORDS )
   return 0;
...
// Code extracted from bpf_filter: pc points to current
// instruction. Both cases can overflow mem[pc->k].
   case BPF_LDX|BPF_MEM:
      X = mem[pc->k]; continue;
   ...
   case BPF_STX:
      mem[pc->k] = X; continue;
```

Figure 7: The BPF code Figure 6's filter exploits.

- FreeBSD BPF accepts filter rules in custom opcode format
- Forgets to check memory read/write offset in some cases, leading to arbitrary kernel memory access

EXE finds real bugs

- 2 buffer overflows in BSD Berkeley Packet Filter
- 4 errors in Linux packet filter
- 5 errors in udhcpd
- A class of errors in pcre
- Errors in ext2, ext3, JFS drivers in Linux

Automated Whitebox Fuzz Testing

Whitebox fuzz testing

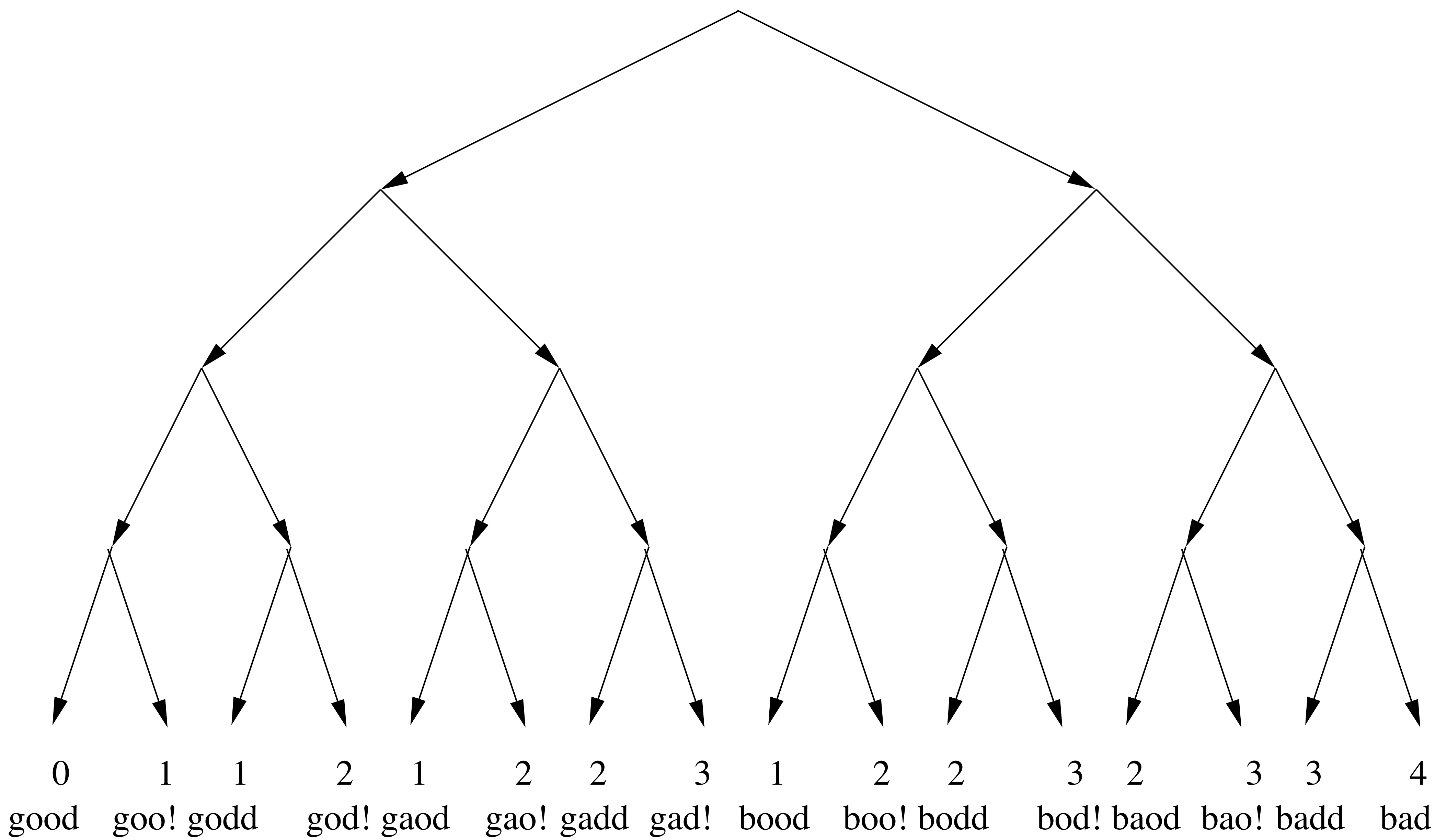
- Insight: valid input gets us close to the interesting code paths
- Idea: execute with valid input, record constraints that were made along the way
- Systematically negate these constraints one-by-one, and observe the results

Example

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) abort(); // error
}
```

- With input "good", we collect the constraints $i_0 \neq b$, $i_1 \neq a$, $i_2 \neq d$, $i_3 \neq !$
- Generate all inputs that don't match this, choose one to use as next input, repeat

Search space



Limitations

- Path explosion
  - n constraints leads to 2^n paths to explore
  - Must prioritize
- Imperfect symbolic execution
  - Calls to libraries/OS, pointer tricks, etc. make perfect symbolic execution difficult

Generational search

```
Search(inputSeed){
  inputSeed.bound = 0;
  workList = {inputSeed};
  Run&Check(inputSeed);
  while (workList not empty) {//new children
    input = PickFirstItem(workList);
    childInputs = ExpandExecution(input);
    while (childInputs not empty) {
      newInput = PickOneItem(childInputs);
      Run&Check(newInput);
      Score(newInput);
      workList = workList + newInput;
    }
  }
}
```

```
ExpandExecution(input) {
  childInputs = {};
  // symbolically execute (program,input)
  PC = ComputePathConstraint(input);
  for (j=input.bound; j < |PC|; j++) {
    if((PC[0..(j-1)] and not(PC[j]))
                   has a solution I){
      newInput = input + I;
      newInput.bound = j;
      childInputs = childInputs + newInput;
  }
  return childInputs;
}
```

- BFS with a heuristic to maximize block coverage
- Score returns the number of new blocks covered

ANI bug

```
RIFF...ACONLIST
B...INFOINAM....
3D Blue Alternat
e v1.1..IART....
................
1996..anih$...$.
................
................
..rate..........
..........seq ..
................
..LIST....framic
on.........  ..
```

```
RIFF...ACONB
B...INFOINAM....
3D Blue Alternat
e v1.1..IART....
................
1996..anih$...$.
................
................
..rate..........
..........seq ..
................
..anih....framic
on.........  ..
```

Figure 5. On the left, an ASCII rendering of a prefix of the seed ANI file used for our search. On the right, the SAGE-generated crash for MS07-017. Note how the SAGE test case changes the `LIST` to an additional `anih` record on the next-to-last line.

- Failure to check the length of the *second* anih record
- Was blackbox fuzz tested, but no test case had more than one anih
- Zero-day exploit of this bug was used in the wild

Crash triage

- Idea: most found bugs can be uniquely identified by the call stack at time of error
- Crashes are bucketed by *stack hash*, which includes information about the functions on the call stack, and the address of the faulting instruction

Results

Media 1:	wff-1	wff-1nh	wff-2	wff-2nh	wff-3	wff-3nh	wff-4	wff-4nh
NULL	1 (46)	1 (32)	1(23)	1(12)	1(32)	1(26)	1(13)	1(1)
ReadAV	1 (40)	1 (16)	2(32)	2(13)	7(94)	4(74)	6(15)	5(45)
WriteAV	0	0	0	0	0	1(1)	1(3)	1(1)
SearchTime	10h7s	10h11s	10h4s	10h20s	10h7s	10h12s	10h34s	9h29m2s
AnalysisTime(s)	5625	4388	16565	11729	5082	6794	5545	7671

Media 1:	wff-5	wff-5nh	bogus-1	bogus-1nh	bogus-2	bogus-3	bogus-4	bogus-5
NULL	1(25)	1(15)	0	0	0	0	0	0
ReadAV	3(44)	3(56)	3(3)	1(1)	0	0	0	0
WriteAV	0	0	0	0	0	0	0	0
SearchTime	10h8s	10h4s	10h8s	10h14s	10h29s	9h47m15s	5m23s	5m39s
AnalysisTime(s)	21614	22005	11640	13156	3885	4480	214	234

Results

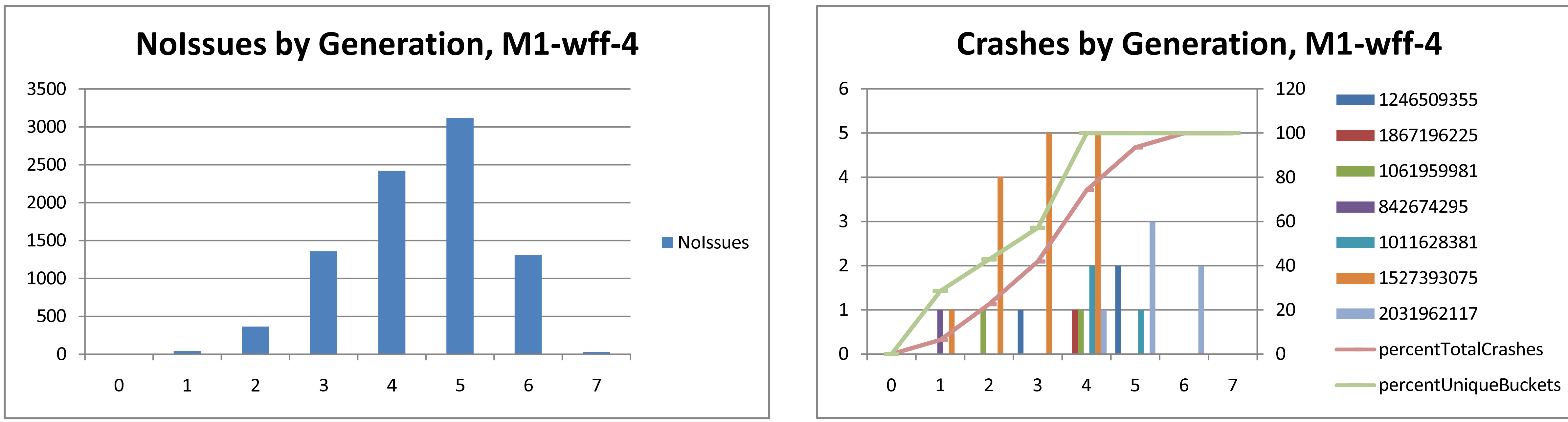


Figure 8. Histograms of test cases and of crashes by generation for Media 1 seeded with `wff-4`.

Most crashes found within a few generations

Discussion

- Generational search is better than DFS
- Bogus files find few bugs
- Different files find different bugs
- Block coverage heuristic doesn’t help much
  - Generation *much* better heuristic

Comparison

- Generational search vs. modified BFS
  - Bad input is usually only a few mutations away from good
  - Incomplete search, but can effectively find bugs in large applications without source
- EXE closer to sound - how much does this matter?