A Bounded Symbolic-Size Model for Symbolic Execution

David Trabish

Shachar Itzhaky

Noam Rinetzky

Tel-Aviv University, Israel

Technion, Israel

KLEE Workshop 2022

(ESEC/FSE 2021)

Motivation

- Size of input affects program behavior

Motivation

- Size of input affects program behavior

$$|input| \geq 5$$

```
int osip_via_parse(const char *hvalue) {
  if (hvalue == NULL) return OSIP_BADPARAMETER;
  const char *version = strchr(hvalue, '/');
  if (version == NULL) return OSIP_SYNTAXERROR;
  const char *protocol = strchr(version + 1, '/');
  if (protocol == NULL) return OSIP_SYNTAXERROR;
  if (protocol - version < 2) return OSIP_SYNTAXERROR;
  ...
  const char *host = strchr(protocol + 1, ' ');
  if (host == NULL) return OSIP_SYNTAXERROR;
  if (host == protocol + 1) {
    while (0 == strncmp(host, " ", 1)) {
      host++;
      if (strlen(host) == 1) return OSIP_SYNTAXERROR;
    }
    host = strchr(host + 1, ' ');
  }
  ...
```

BUG

$$|input| = 1$$

```
int osip_uri_parse_headers(const char *headers) {
  const char *equal = strchr(headers, '=');
  const char *_and = strchr(headers + 1, '&');
  ...
}
```

BUG

Motivation

- Size of input affects program behavior
- The problem: concrete-size model

$|input| \geq 5$

```
int osip_via_parse(const char *hvalue) {
  if (hvalue == NULL) return OSIP_BADPARAMETER;
  const char *version = strchr(hvalue, '/');
  if (version == NULL) return OSIP_SYNTAXERROR;
  const char *protocol = strchr(version + 1, '/');
  if (protocol == NULL) return OSIP_SYNTAXERROR;
  if (protocol - version < 2) return OSIP_SYNTAXERROR;
  ...
  const char *host = strchr(protocol + 1, ' ');
  if (host == NULL) return OSIP_SYNTAXERROR;
  if (host == protocol + 1) {
    while (0 == strncmp(host, " ", 1)) {
      host++;
      if (strlen(host) == 1) return OSIP_SYNTAXERROR;
    }
    host = strchr(host + 1, ' ');
  }
  ...
```

BUG

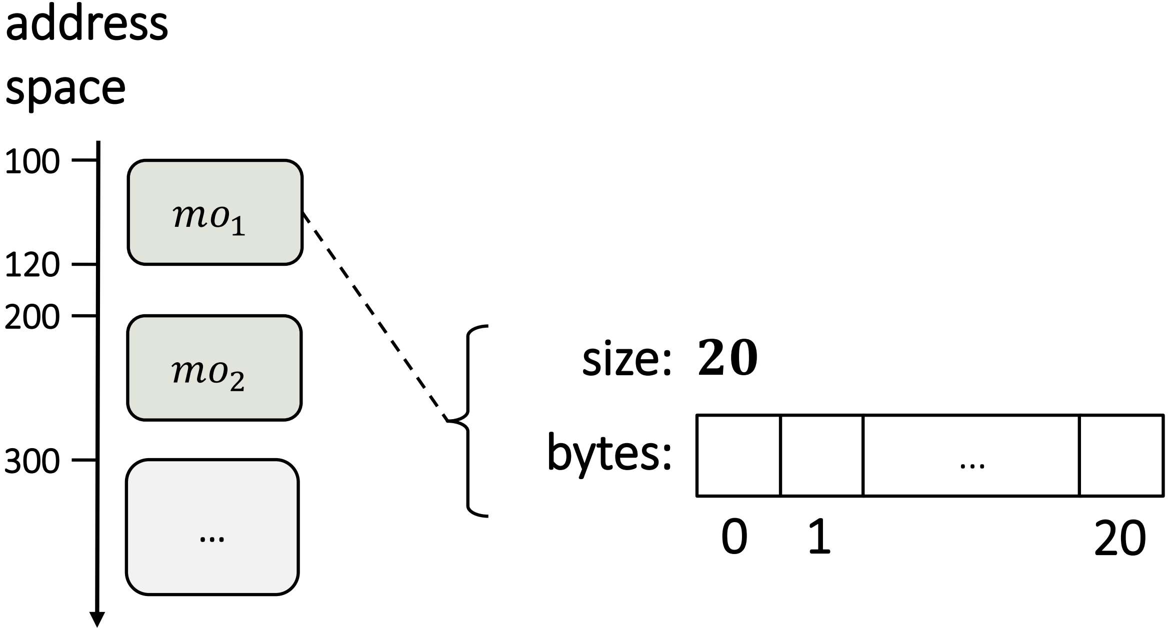
$|input| = 1$

```
int osip_uri_parse_headers(const char *headers) {
  const char *equal = strchr(headers, '=');
  const char *_and = strchr(headers + 1, '&');
  ...
}
```

BUG

Concrete-Size Model

- Linear address space
- Explicit encoding *(QF_ABV)*

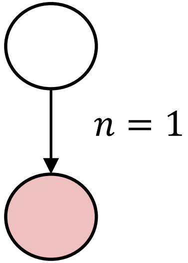


Concrete-Size Model

```
int n, z; // symbolic

char *p = malloc(n);
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```

Concrete-Size Model

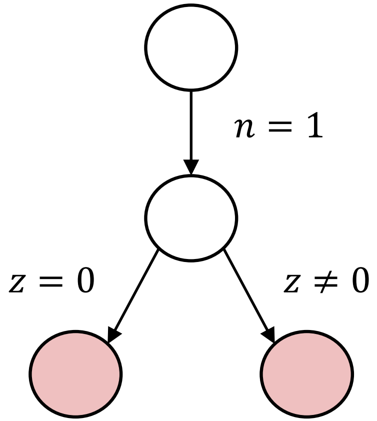
```
int n, z; // symbolic

char *p = malloc(n);
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```



concretize symbolic size n

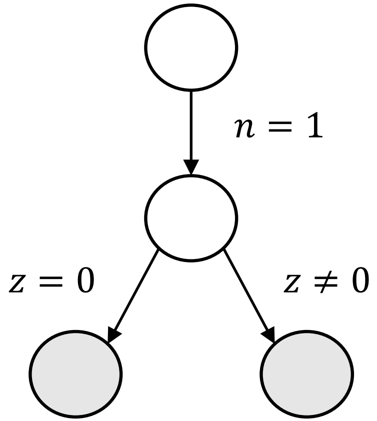
Concrete-Size Model

```
int n, z; // symbolic

char *p = malloc(n);
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```



Concrete-Size Model

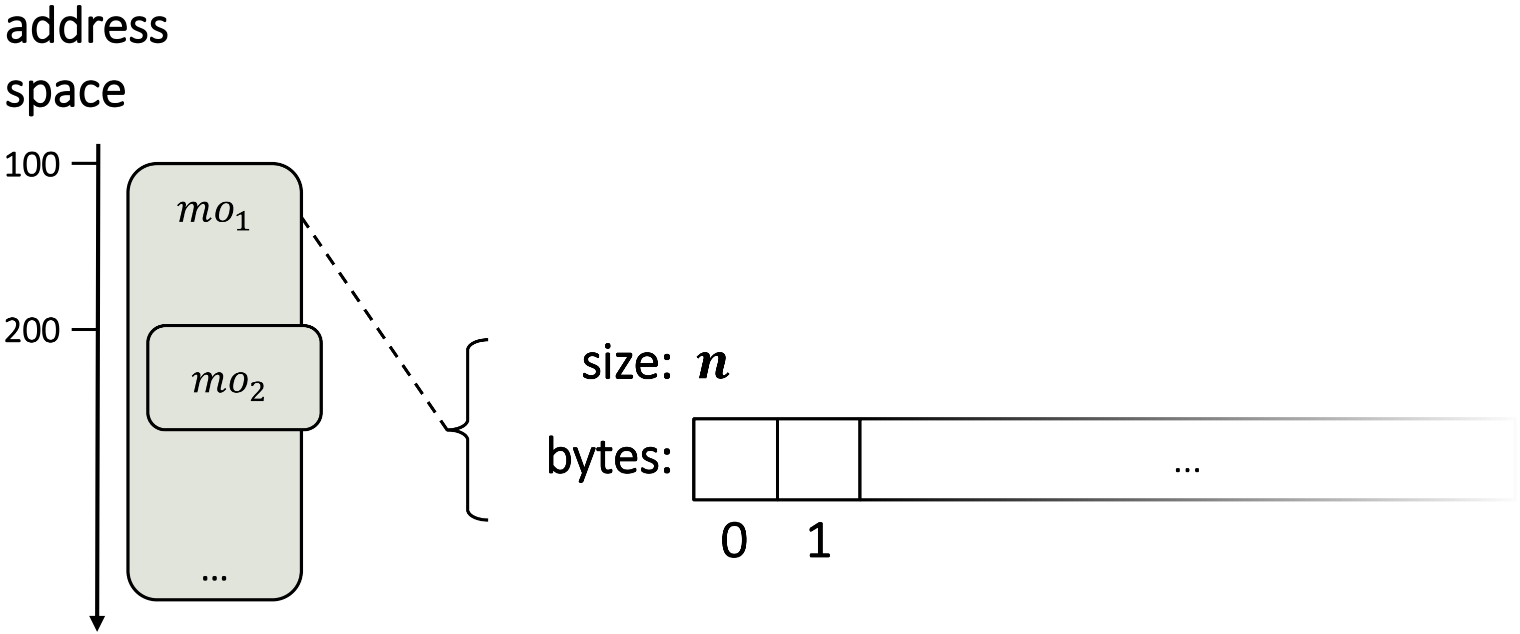
```
int n, z; // symbolic

char *p = malloc(n);
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```



only 2 paths explored

Unbounded Symbolic-Size Model

- Linear address space → overlapping
- Explicit encoding *(QF_ABV)* → unbounded memory consumption



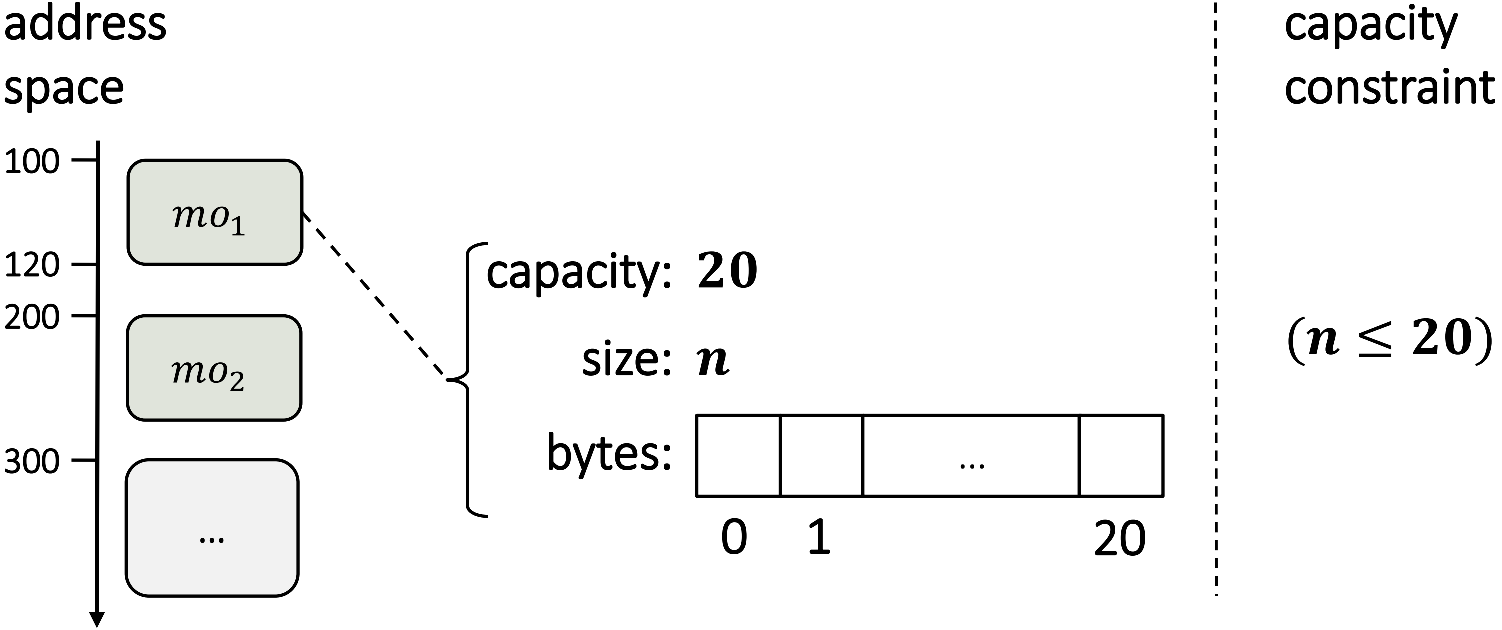
Bounded Symbolic-Size Model

Every memory object has:

- A fixed (concrete) capacity $\boldsymbol{c}$
- A symbolic size $\boldsymbol{n}$ such that: $\boldsymbol{n} \leq \boldsymbol{c}$

Bounded Symbolic-Size Model

- Linear address space → supported
- Explicit encoding *(QF_ABV)* → controllable memory consumption

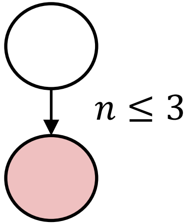


Symbolic-Size Model

```
int n, z; // symbolic

char *p = malloc(n); // capacity = 3
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```

Symbolic-Size Model

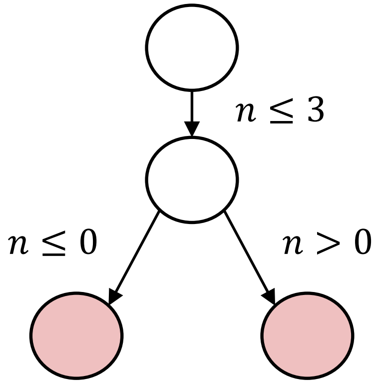
```
int n, z; // symbolic

char *p = malloc(n); // capacity = 3
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```



add capacity constraint

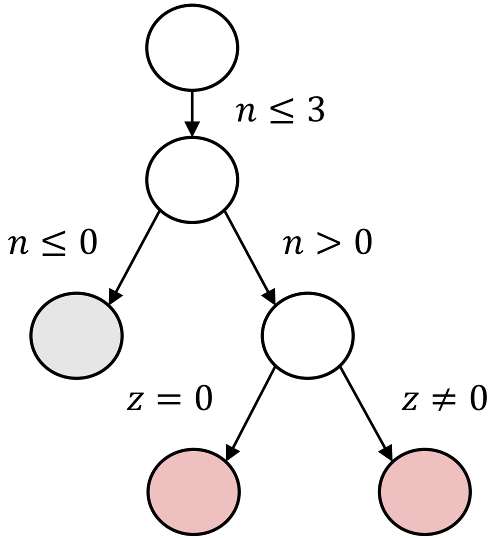
Symbolic-Size Model

```
int n, z; // symbolic

char *p = malloc(n); // capacity = 3
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```



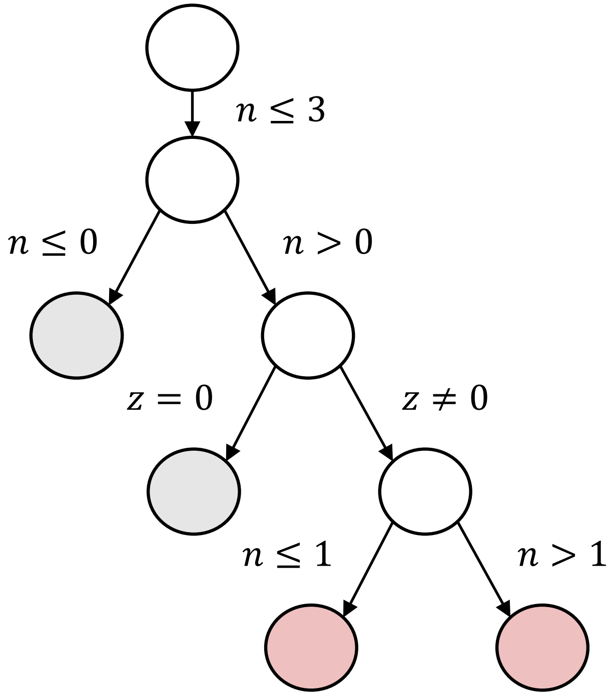
Symbolic-Size Model

```
int n, z; // symbolic

char *p = malloc(n); // capacity = 3
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```



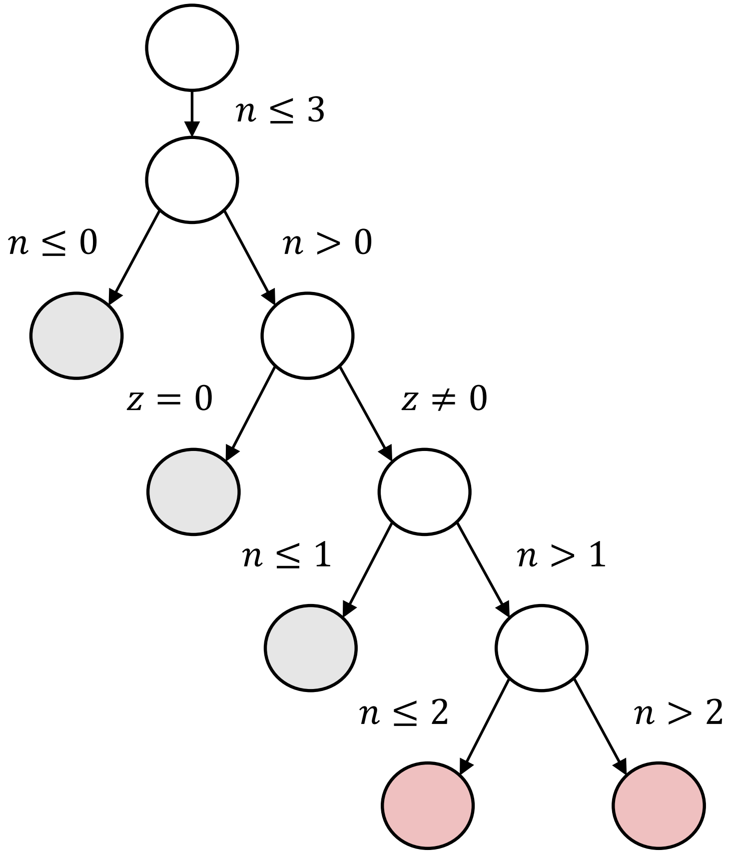
Symbolic-Size Model

```
int n, z; // symbolic

char *p = malloc(n); // capacity = 3
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```



Symbolic-Size Model

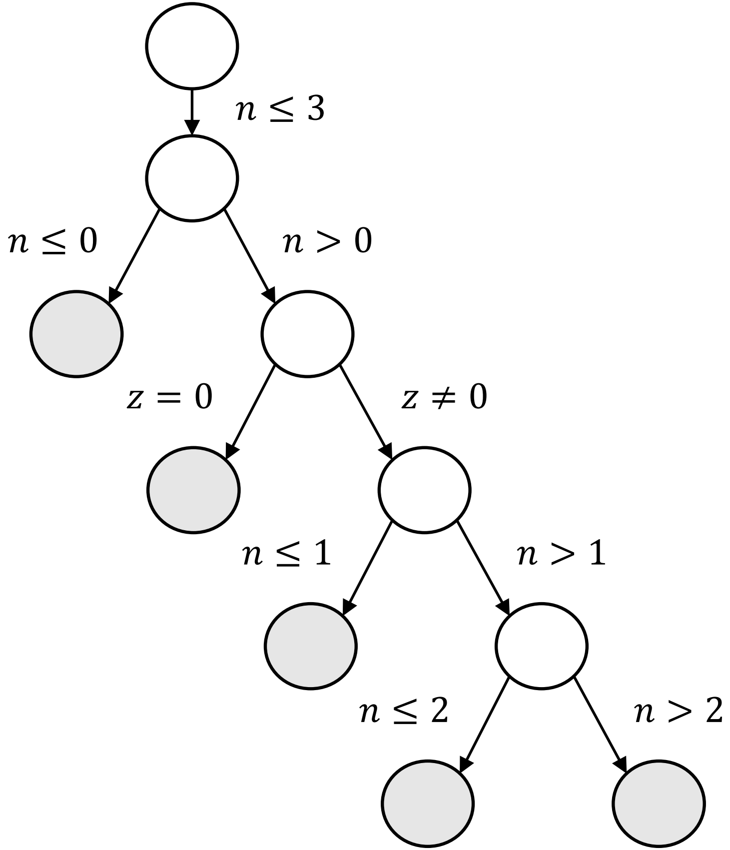
```
int n, z; // symbolic

char *p = malloc(n); // capacity = 3
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```



Symbolic-Size Model

```
int n, z; // symbolic

char *p = malloc(n); // capacity = 3
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```

5 paths explored



Arising Challenges

- Additional symbolic-size expressions
- Amplifies path explosion
  - Especially with **size-dependent loops**

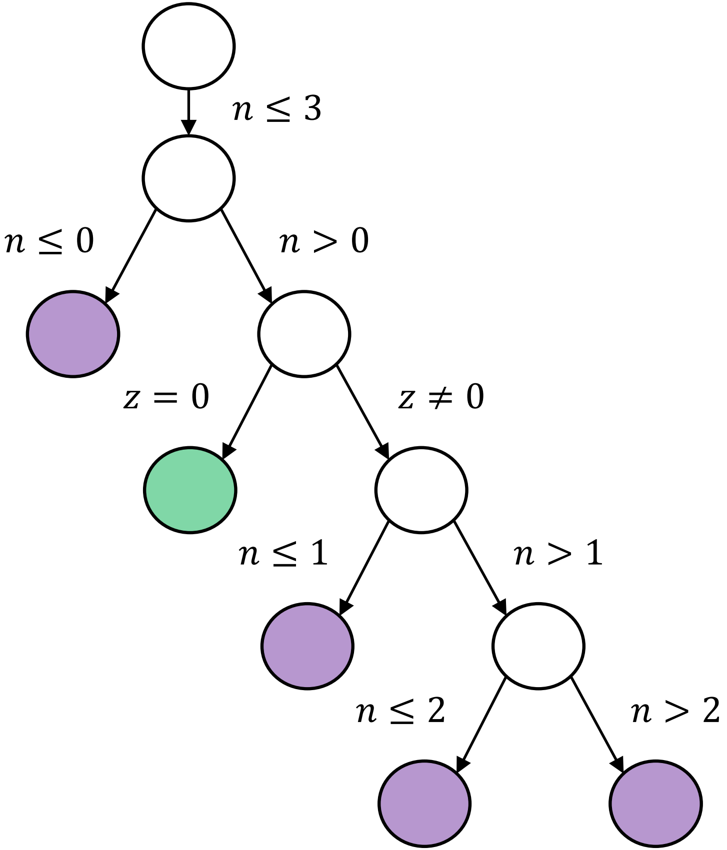
Merging Approach

- Detect **symbolic-size dependent** loops
- Execute the loop till **full exploration**
- **Merge** the resulting states

Merging Approach

```
int n; // symbolic
int z; // symbolic

char *p = malloc(n); // capacity = 3
for (unsigned i = 0; i < n; i++) {
  if (z == 0) {
    break;
  }
  p[i] = i;
}
```

group states by loop exit



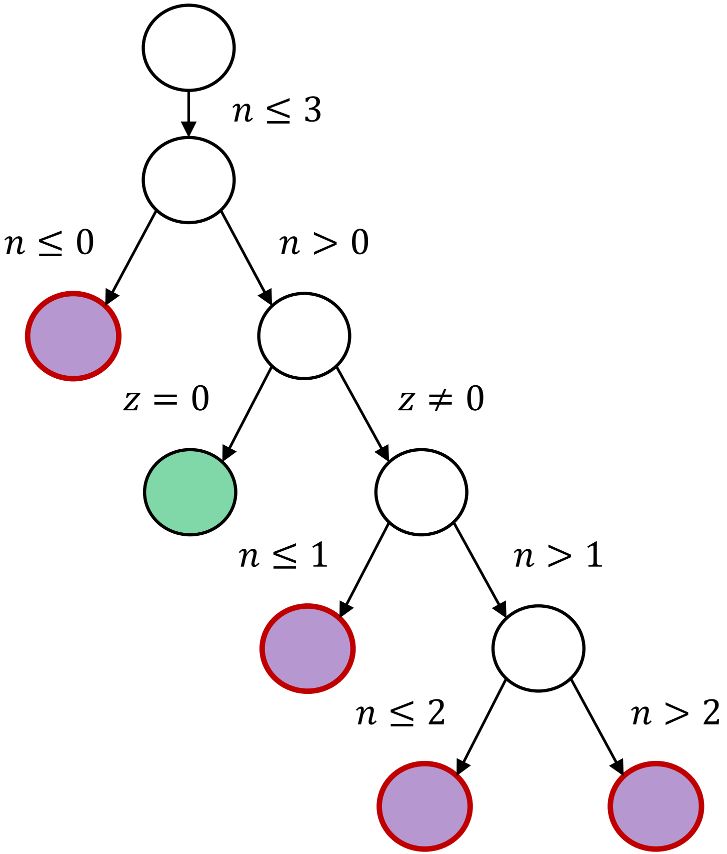
Merging Approach

$(n \leq 0) \lor$
$(n > 0 \land z \neq 0 \land n \leq 1) \lor$
$(n > 0 \land z \neq 0 \land n > 1 \land n \leq 2) \lor$
$(n > 0 \land z \neq 0 \land n > 1 \land n > 2)$

merged constraint

$(n > 0 \land z = 0)$

merged constraint



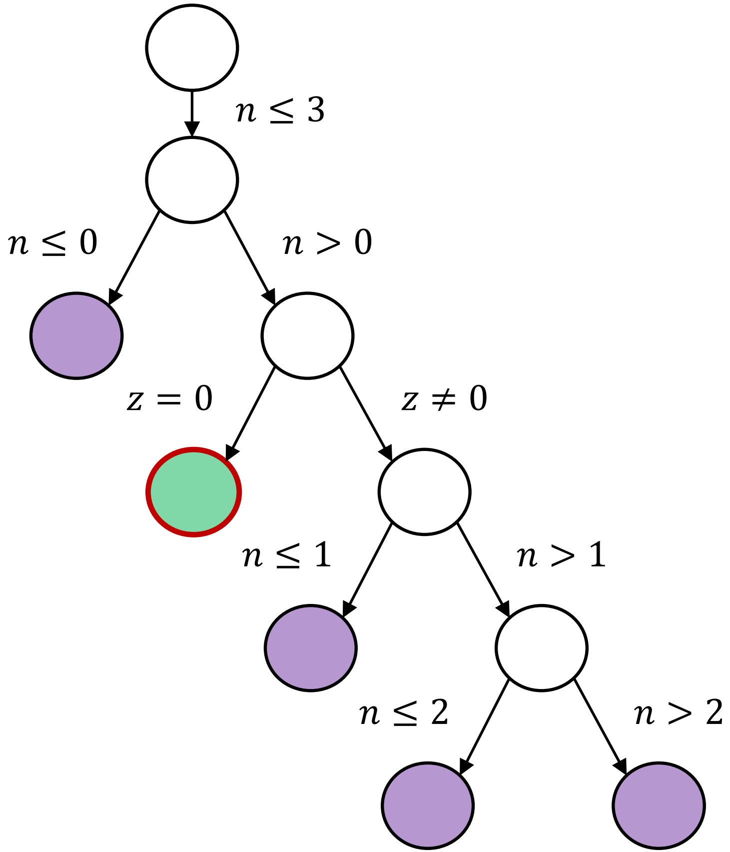
Merging Approach

$(n \leq 0) \vee$
$(n > 0 \wedge z \neq 0 \wedge n \leq 1) \vee$
$(n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n \leq 2) \vee$
$(n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n > 2)$

merged constraint

$(n > 0 \wedge z = 0)$

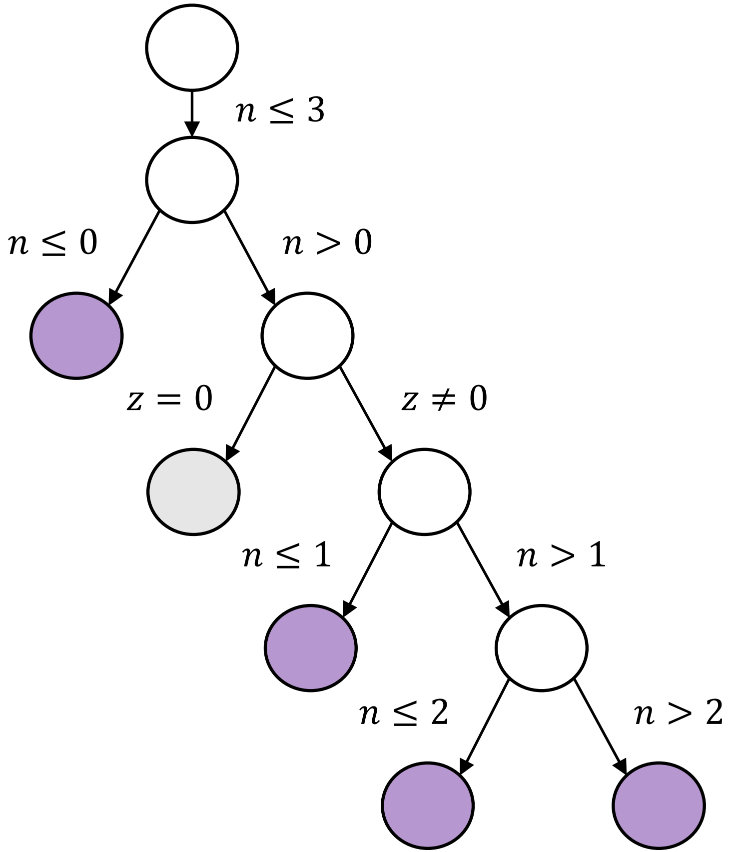
merged constraint



Merging Optimization

$(n \leq 0) \lor$
$(n > 0 \land z \neq 0 \land n \leq 1) \lor$
$(n > 0 \land z \neq 0 \land n > 1 \land n \leq 2) \lor$
$(n > 0 \land z \neq 0 \land n > 1 \land n > 2)$

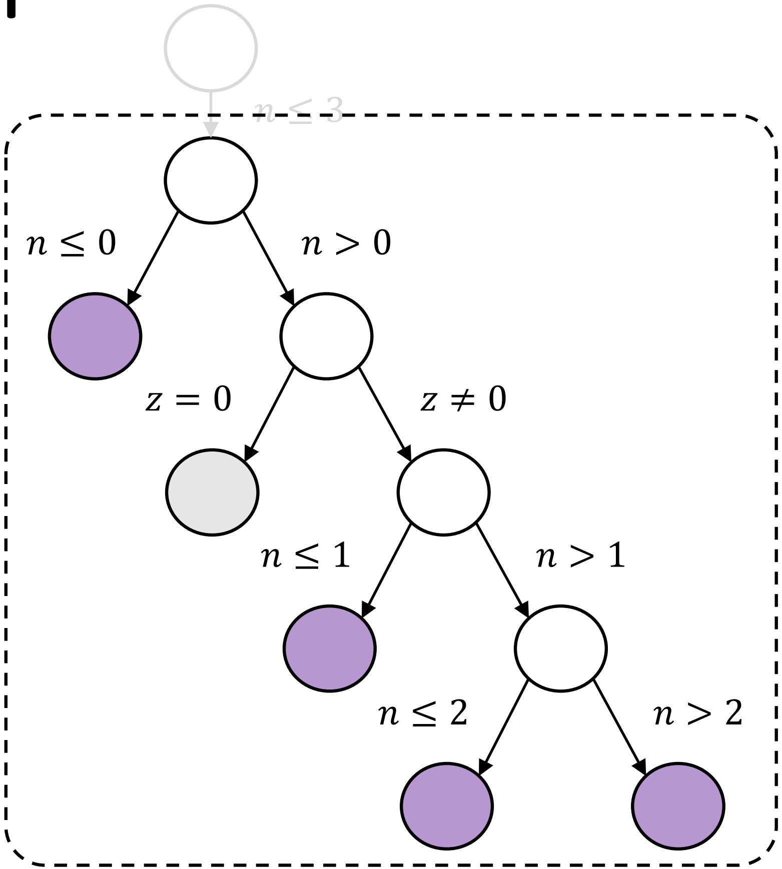
merged constraint



Merging Optimization

$(n \leq 0) \vee$
$(n > 0 \wedge z \neq 0 \wedge n \leq 1) \vee$
$(n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n \leq 2) \vee$
$(n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n > 2)$

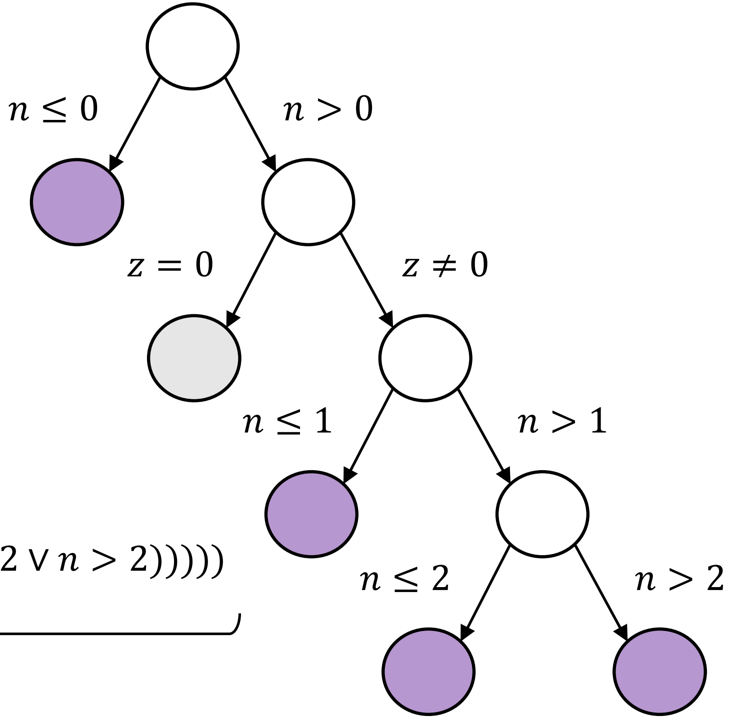
merged constraint



Merging Optimization

$(n \leq 0) \vee$
$(n > 0 \wedge z \neq 0 \wedge n \leq 1) \vee$
$(n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n \leq 2) \vee$
$(n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n > 2)$

merged constraint



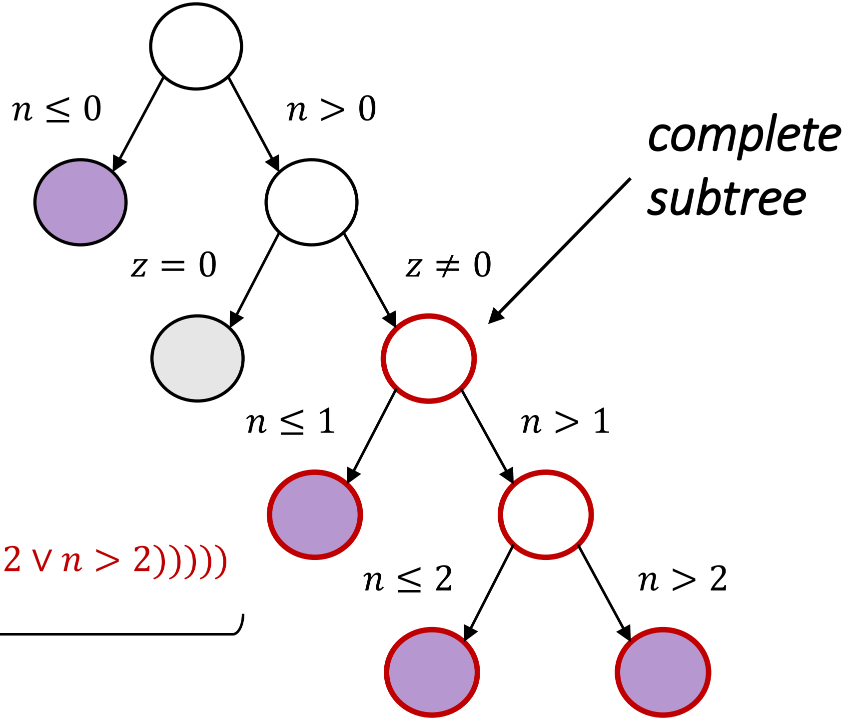
$(n \leq 0 \vee (n > 0 \wedge z \neq 0 \wedge (n \leq 1 \vee (n > 1 \wedge (n \leq 2 \vee n > 2)))))$

merged constraint

Merging Optimization

$(n \leq 0) \vee$
$(n > 0 \wedge z \neq 0 \wedge n \leq 1) \vee$
$(n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n \leq 2) \vee$
$(n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n > 2)$

merged constraint



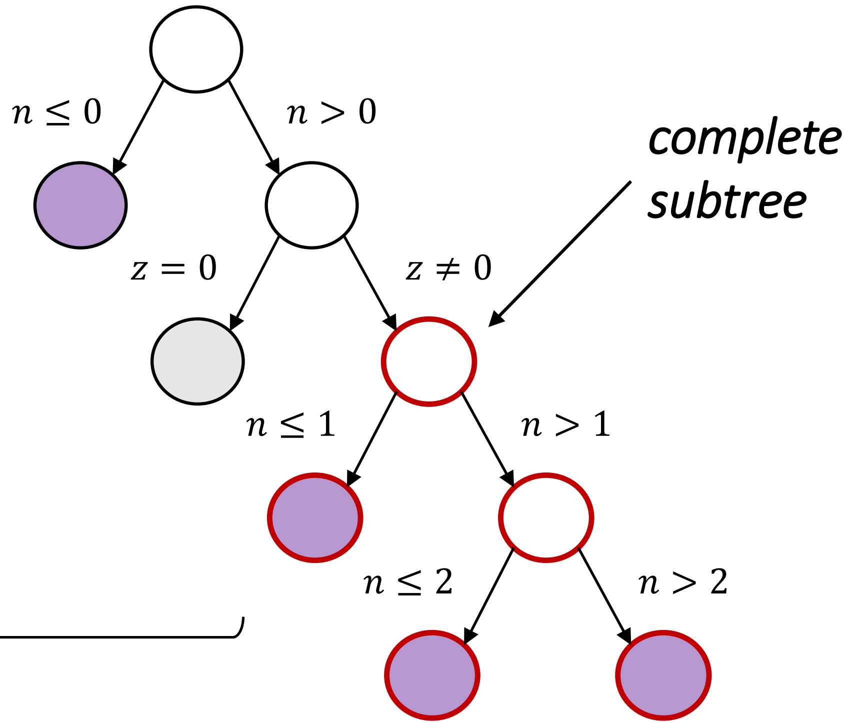
$(n \leq 0 \vee (n > 0 \wedge z \neq 0 \wedge (n \leq 1 \vee (n > 1 \wedge (n \leq 2 \vee n > 2)))))$

merged constraint

Merging Optimization

$(n \leq 0) \vee$
$(n > 0 \wedge z \neq 0 \wedge n \leq 1) \vee$
$(n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n \leq 2) \vee$
$(n > 0 \wedge z \neq 0 \wedge n > 1 \wedge n > 2)$

merged constraint



$(n \leq 0 \vee (n > 0 \wedge z \neq 0))$

merged constraint

Limitations (Model)

```
size_t n; // symbolic
char *p = calloc(n, 1); // capacity = 100
if (n > 0 && n < 3) {
  p[n - 1] = 7;
  if (p[0]) {
    ...
  }
}
```

Limitations (Model)

```
size_t n; // symbolic
char *p = calloc(n, 1); // capacity = 100
if (n > 0 && n < 3) {
  p[n - 1] = 7;
  if (p[0]) {
    ...
  }
}
```

$$\underbrace{select(a[0 \rightarrow 0, 1 \rightarrow 0, 2 \rightarrow 0, 3 \rightarrow 0, \quad \ldots \quad , 99 \rightarrow 0, n-1 \rightarrow 7], 0)}_{\text{symbolic value of p[0]}}$$

Limitations (Model)

```
size_t n; // symbolic
char *p = calloc(n, 1); // capacity = 100
if (n > 0 && n < 3) {
  p[n - 1] = 7;
  if (p[0]) {
    ...
  }
}
```

redundant stores

$$select(a[0 \to 0, 1 \to 0, 2 \to 0, 3 \to 0, \quad \ldots \quad , 99 \to 0, n - 1 \to 7], 0)$$

symbolic value of p[0]

Limitations (State Merging)

High input capacity → **complex expressions**

merged path constraints:

$(n > 0 \land s[0] = 7) \lor$
$\quad (n > 0 \land s[0] \neq 7 \land n > 1 \land s[1] = 7) \lor$
$\quad\quad (n > 0 \land s[0] \neq 7 \land n > 1 \land s[1] \neq 7 \land n > 2 \land s[2] = 7)$
$\quad\quad\quad (n > 0 \land s[0] \neq 7 \land n > 1 \land s[1] \neq 7 \land n > 2 \land s[2] \neq 7 \land n > 3 \land s[3] = 7)$

merged value of `index`:

$ite(n > 0 \land s[0] = 7,$
$\quad 0,$
$\quad ite(n > 0 \land s[0] \neq 7 \land n > 1 \land s[1] = 7,$
$\quad\quad 1,$
$\quad\quad ite(n > 0 \land s[0] \neq 7 \land n > 1 \land s[1] \neq 7 \land n > 2 \land s[2] = 7,$
$\quad\quad\quad 2,$
$\quad\quad\quad 3)$

```
// symbolic
size_t n;
// symbolic, capacity = 4
char *s = malloc(n);
unsigned index = 0;
while (index < n) {
  if (s[index] == 7)
    break;
  index++;
}
```

Limitations (State Merging)

High input capacity → **complex expressions**

merged path constraints:

$$(n > 0) \wedge (s[0] = 7 \vee$$
$$(s[0] \neq 7 \wedge n > 1 \wedge (s[1] = 7 \vee$$
$$(s[1] \neq 7 \wedge n > 2 \wedge (s[2] = 7 \vee$$
$$(s[2] \neq 7 \wedge n > 3 \wedge s[3] = 7))))))$$

merged value of `index`:

$$ite(s[0] = 7,$$
$$0,$$
$$ite(s[1] = 7,$$
$$1,$$
$$ite(s[2] = 7,$$
$$2,$$
$$3)$$

```
// symbolic
size_t n;
// symbolic, capacity = 4
char *s = malloc(n);
unsigned index = 0;
while (index < n) {
  if (s[index] == 7)
    break;
  index++;
}
```

Evaluation

API Testing
- GNU libtasn1 *(17 API's)*
- libpng *(13 API's)*
- GNU oSIP *(48 API's)*

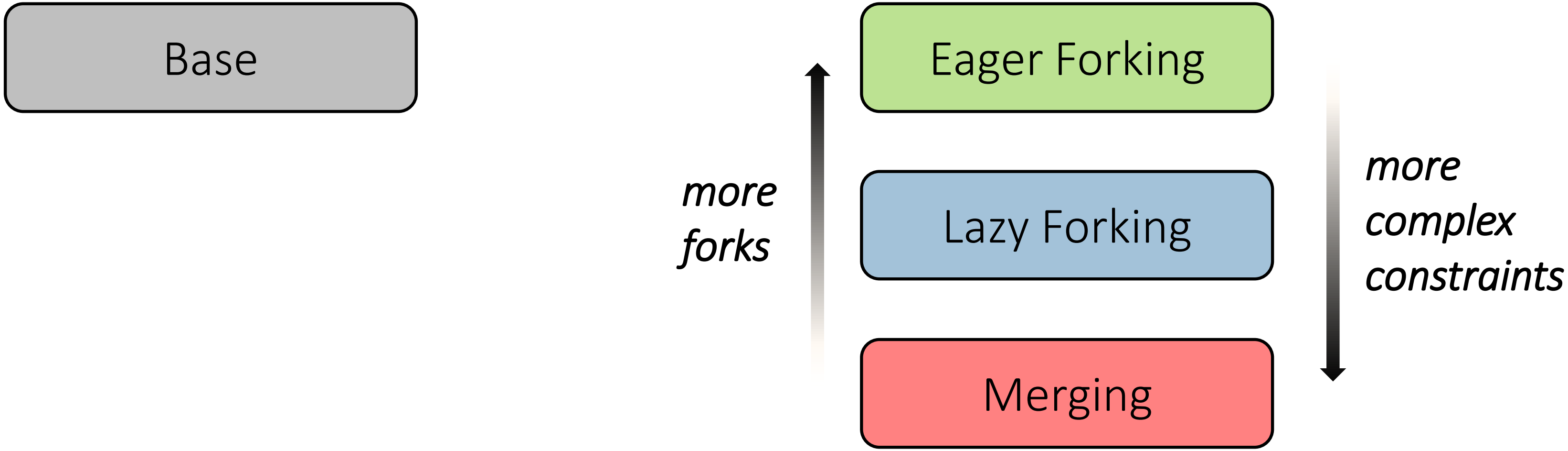
Whole-program testing
- GNU Coreutils *(99 programs)*

Implementation
- On top of ***KLEE***

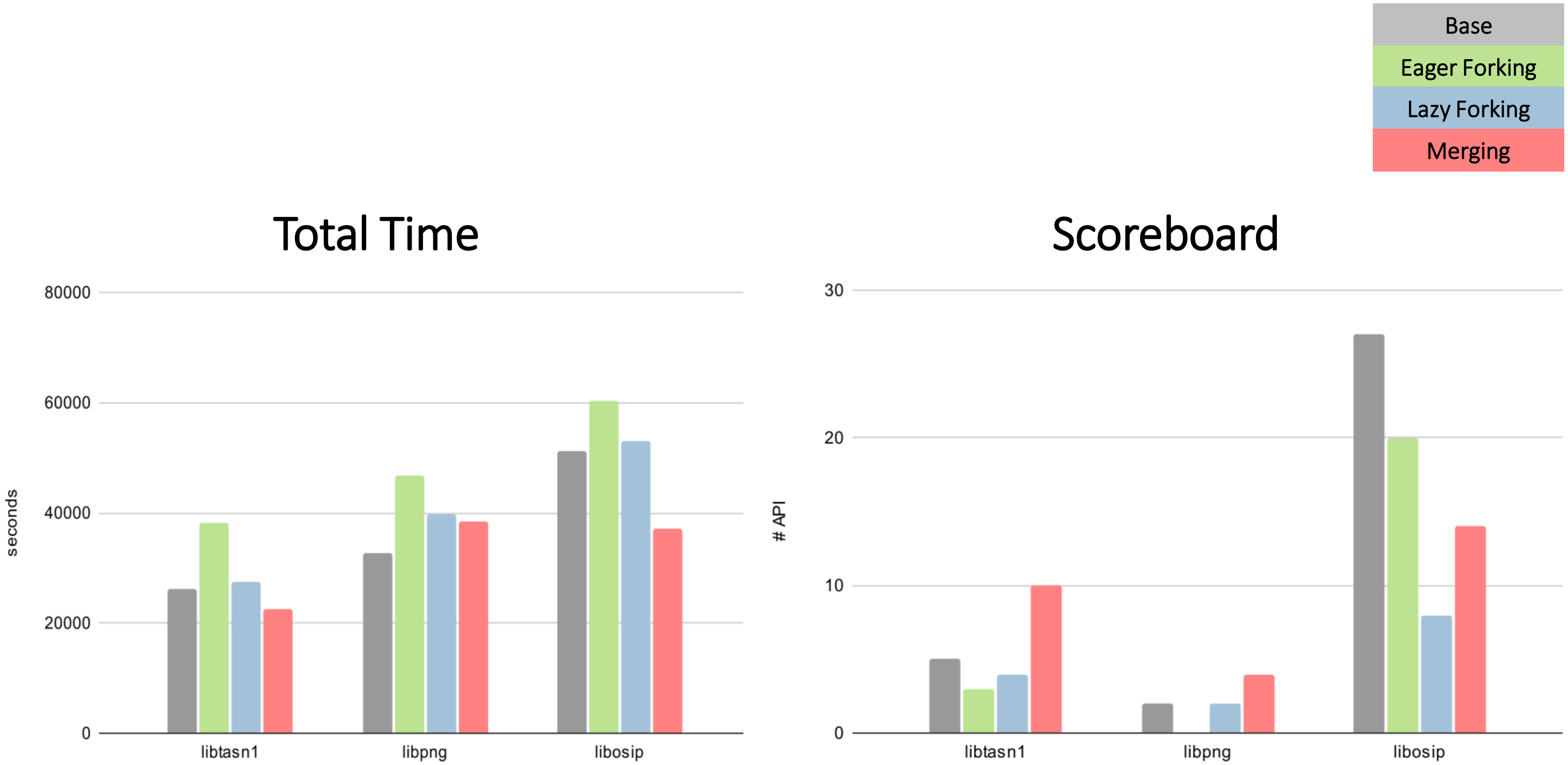
Evaluation: Approaches

Concrete-size Model

Bounded Symbolic-Size Model



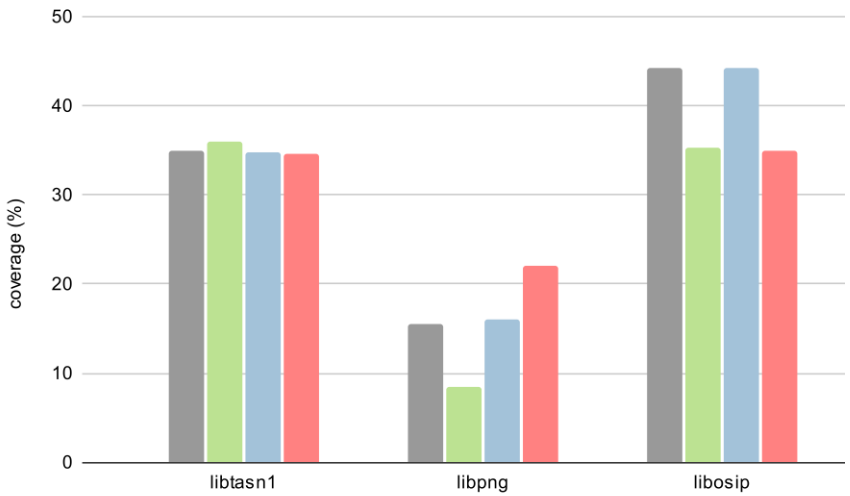
API Testing: Analysis Time



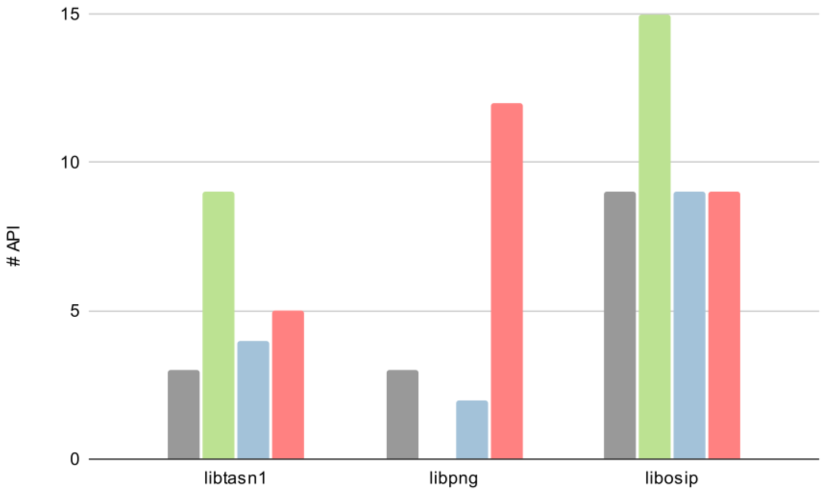
API Testing: Coverage

Base
Eager Forking
Lazy Forking
Merging

Total Coverage

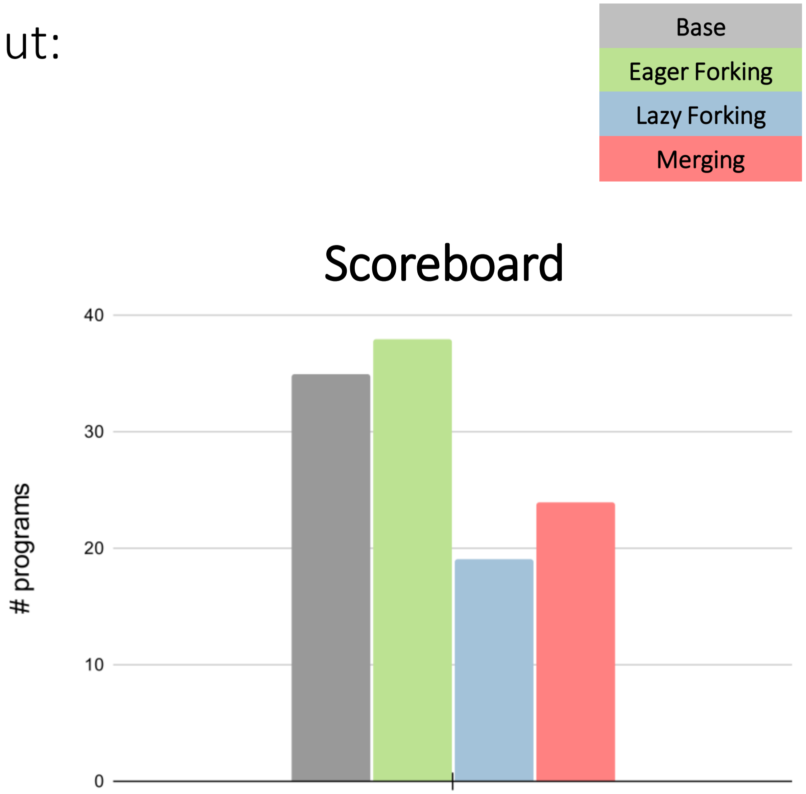


Scoreboard



Evaluation: GNU Coreutils

- In 94 programs, all approaches timeout:
  - Compare coverage
- In the rest 5 programs:
  - Merging approach is faster



Found Bugs

- GNU libtasn1
  - one *out-of-bound-read*
- GNU oSIP
  - three *out-of-bound-read’s*
  - one *integer-underflow*

All the bugs were confirmed and fixed.

Summary

- Bounded modeling of variable-size inputs
- Evaluated in API testing and whole-program testing
- Found previously unknown bugs

Future Work

- Applying in other domains (patch testing, program repair, …)
- Better encoding in state merging

https://github.com/davidtr1037/klee-symsize