# State Merging with Quantifiers in Symbolic Execution



David Trabish[1]


Noam Rinetzky[1]


Sharon Shoham[1]


Vaibhav Sharma[2]

[1]Tel Aviv University, Israel

[2]University of Minnesota, USA


**ESEC/FSE 2023**

# Symbolic Execution: Introduction

Program analysis technique

- Systematically explores paths
- Checks feasibility using SMT

Main challenges

- Path explosion
- Constraint solving

# Symbolic Execution: State Merging

- Mitigates path explosion by joining exploration paths
- Often leads to:
  - Large disjunctive constraints
  - Costly constraint solving

# Main Contributions

- State merging using compact quantified constraints
- Specialized solving procedure

$$(\ldots) \vee (\ldots) \vee \ldots\ldots\ldots \vee (\ldots)$$

$$\Downarrow$$

$$\forall x. (\ldots)$$

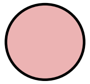
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```

# Example

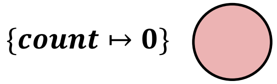
```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```

$\{count \mapsto 0\}$



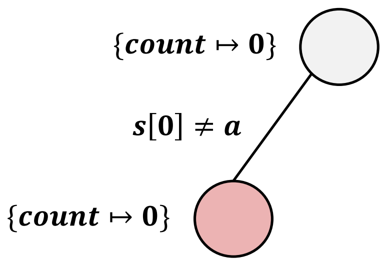
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```



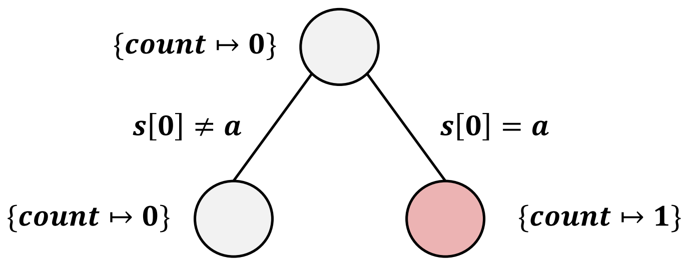
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, 'a');
int m = strspn(s + n, 'b');
...
```



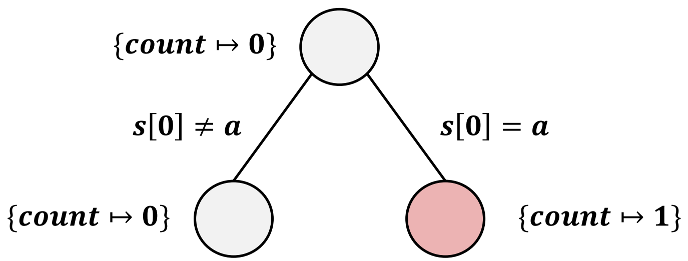
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```



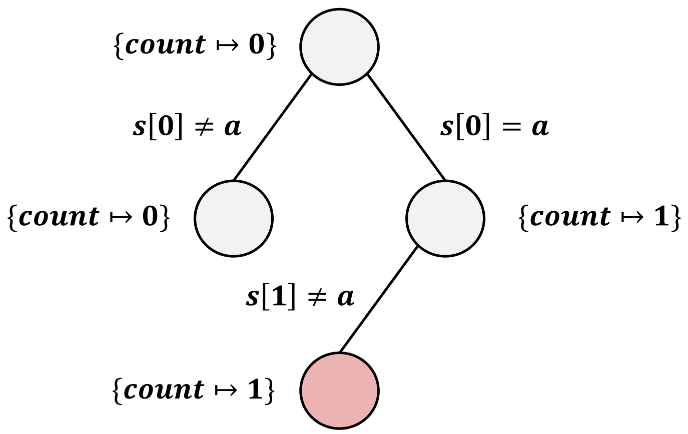
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```



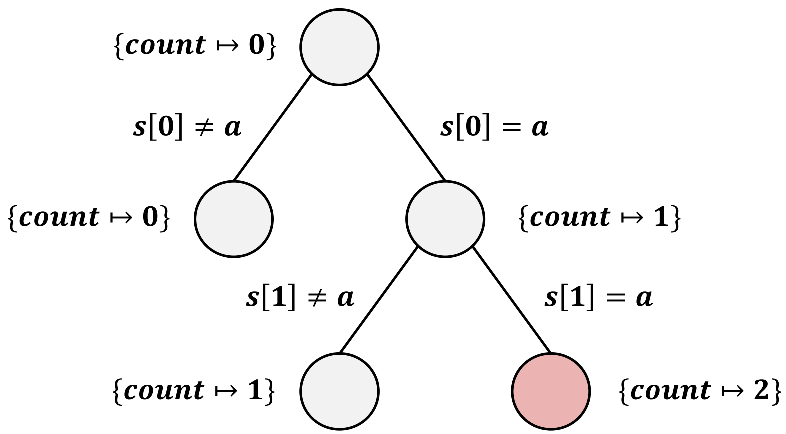
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```



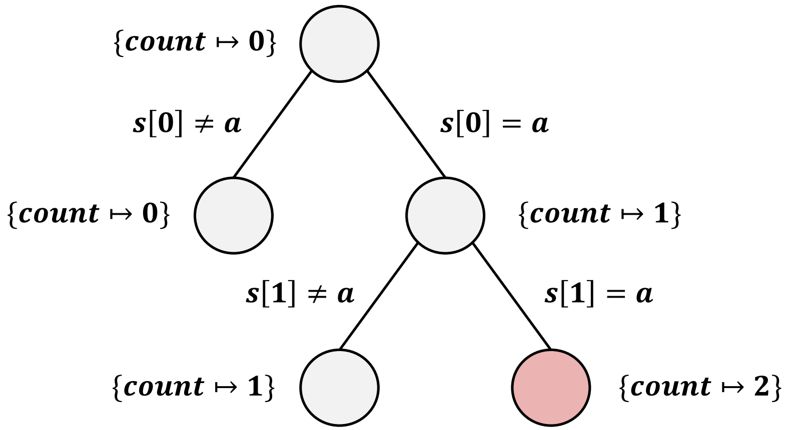
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```



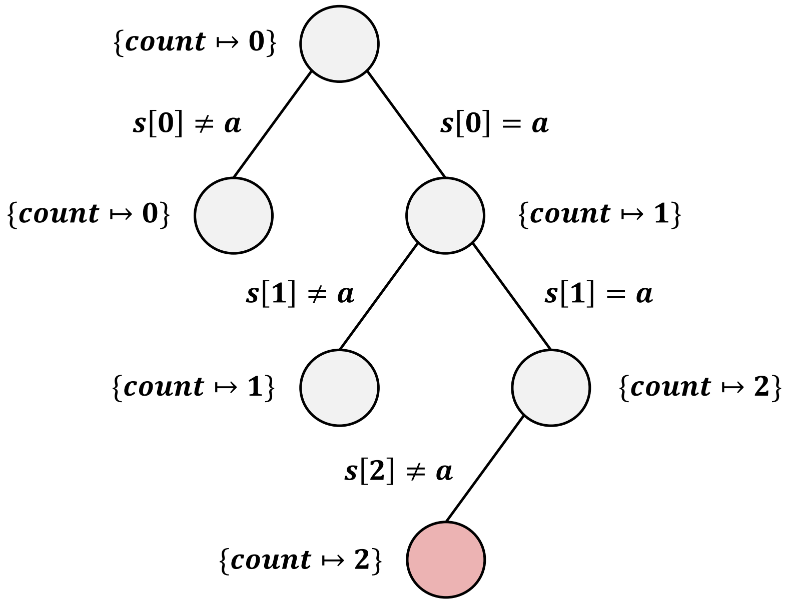
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```



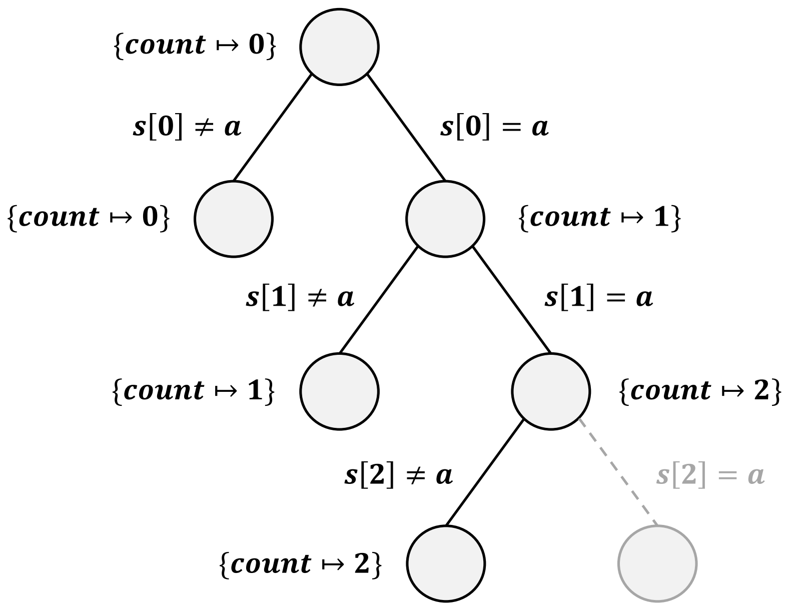
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```



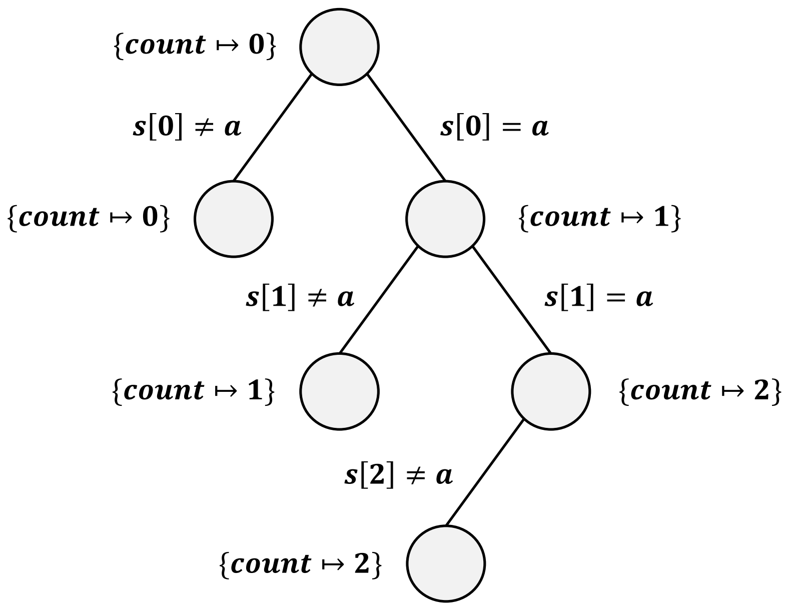
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```



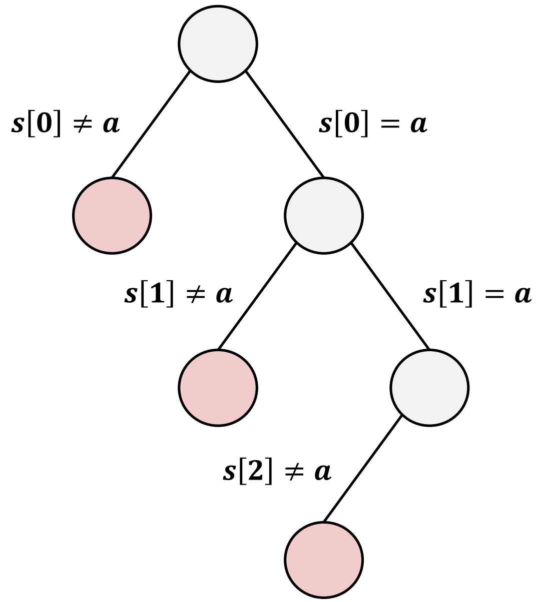
# Example

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```



# Standard State Merging

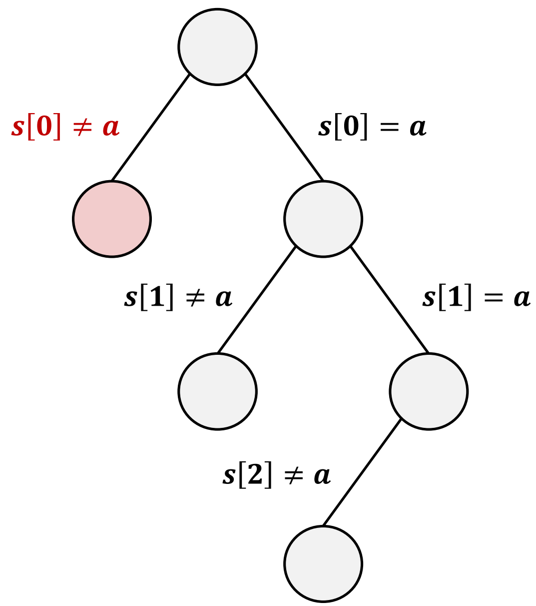
Merging the **path constraints**



# Standard State Merging

Merging the **path constraints**

$s[0] \neq a$

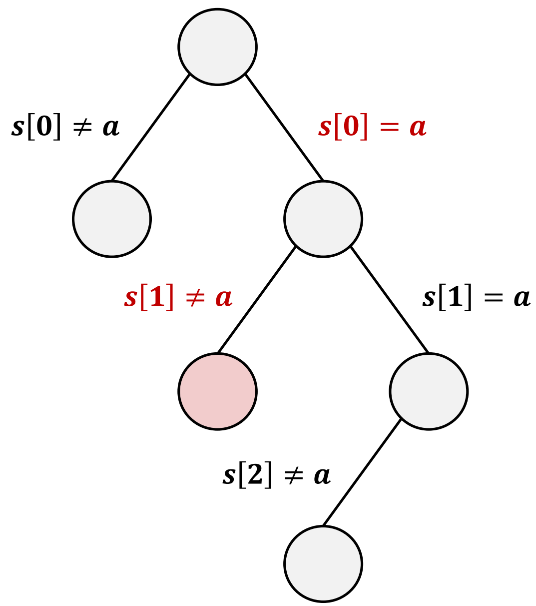


# Standard State Merging

Merging the **path constraints**

$s[0] \neq a$

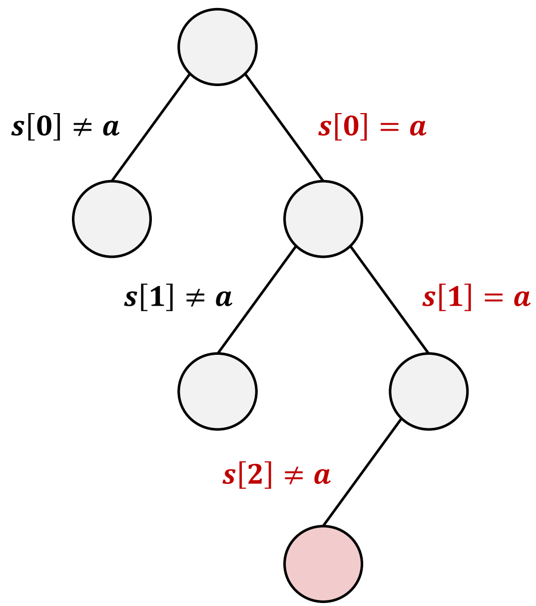
$s[0] = a \wedge s[1] \neq a$



# Standard State Merging

Merging the **path constraints**

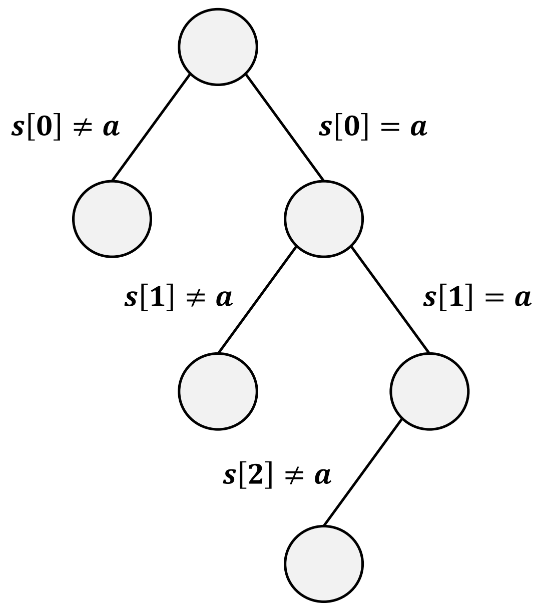
$s[0] \neq a$
$s[0] = a \wedge s[1] \neq a$
$s[0] = a \wedge s[1] = a \wedge s[2] \neq a$



# Standard State Merging

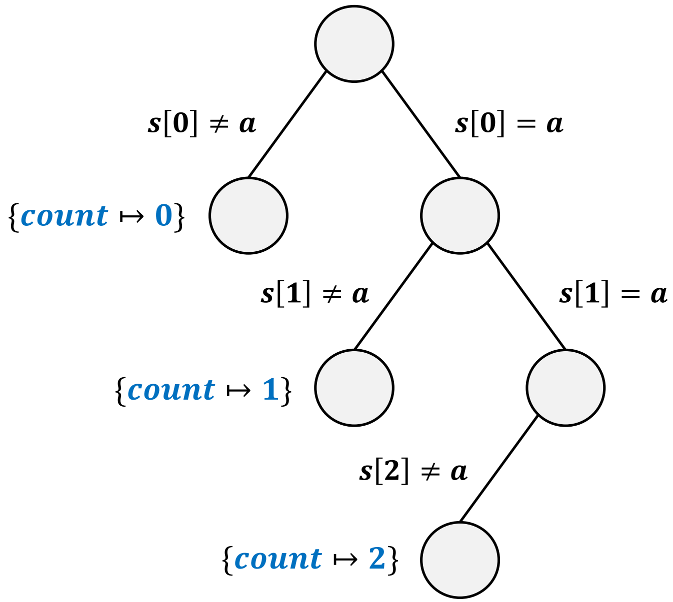
Merging the **path constraints**

$(s[0] \neq a) \vee$
$(s[0] = a \wedge s[1] \neq a) \vee$
$(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)$



# Standard State Merging

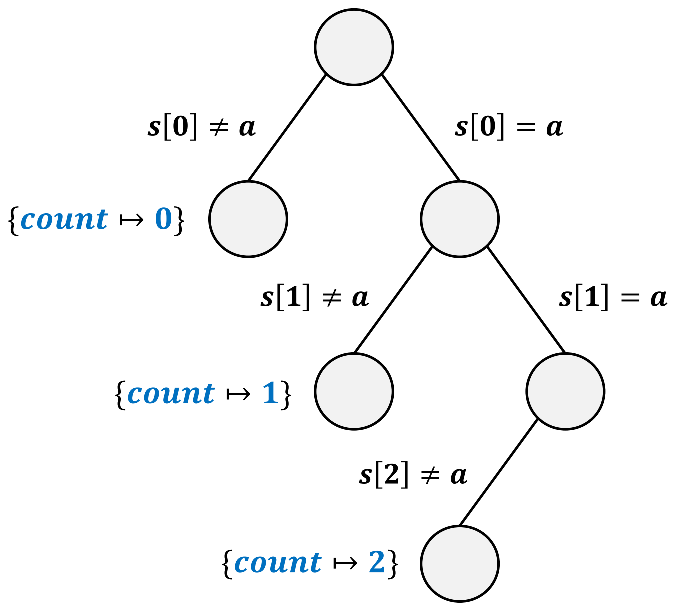
Merging the **memory**



# Standard State Merging

Merging the **memory**

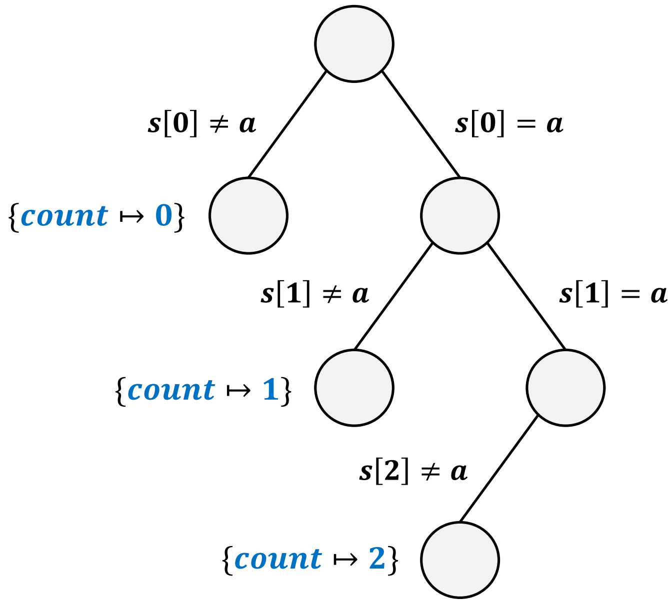
$$\underbrace{\begin{array}{l} ite( \\ \quad s[0] \neq a, \\ \quad 0, \\ \quad \ldots \\ ) \end{array}}_{\text{merged value of count}}$$



# Standard State Merging

Merging the **memory**

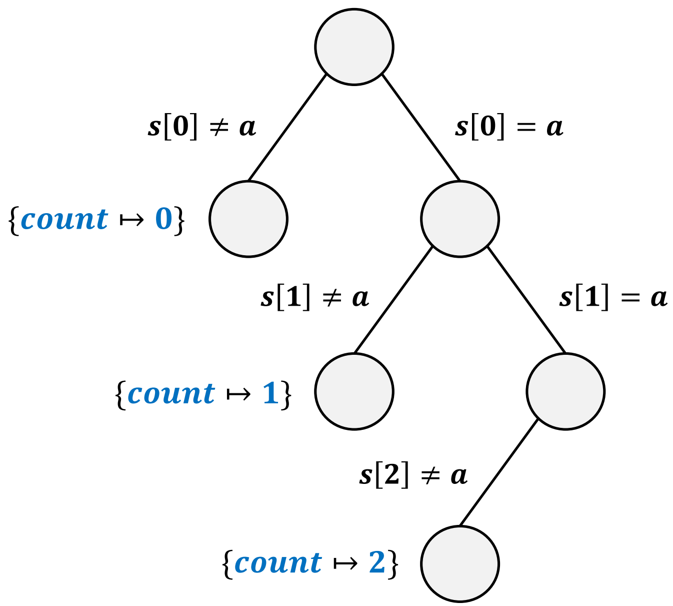
$$\underbrace{\begin{array}{l} ite( \\ \quad s[0] \neq a, \\ \quad 0, \\ \quad ite( \\ \quad\quad s[0] = a \land s[1] \neq a, \\ \quad\quad 1, \\ \quad\quad \ldots \\ \quad ) \\ ) \end{array}}_{\text{merged value of count}}$$



# Standard State Merging

Merging the **memory**

$$\underbrace{\begin{array}{l} ite( \\ \quad s[0] \neq a, \\ \quad 0, \\ \quad ite( \\ \quad\quad s[0] = a \wedge s[1] \neq a, \\ \quad\quad 1, \\ \quad\quad 2 \\ \quad ) \\ ) \end{array}}_{\text{merged value of count}}$$



# Standard State Merging

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```

$$\left\{ \begin{array}{l} ite( \\ \quad s[0] \neq a, \\ \quad 0, \\ \quad ite( \\ \quad\quad s[0] = a \land s[1] \neq a, \\ \quad\quad 1, \\ \quad\quad 2 \\ \quad ) \\ ) \end{array} \right.$$

# Standard State Merging

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```

$$
\left\{
\begin{array}{l}
ite( \\
\quad s[0] \neq a, \\
\quad 0, \\
\quad ite( \\
\quad\quad s[0] = a \land s[1] \neq a, \\
\quad\quad 1, \\
\quad\quad 2 \\
\quad ) \\
)
\end{array}
\right.
$$

# Standard State Merging

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, 'a');
int m = strspn(s + n, 'b');
...
```

$$\left\{\begin{array}{l} ite( \\ \quad s[0] \neq a, \\ \quad 0, \\ \quad ite( \\ \quad\quad s[0] = a \wedge s[1] \neq a, \\ \quad\quad 1, \\ \quad\quad 2 \\ \quad ) \\ ) \end{array}\right.$$

# Standard State Merging

## Path constraints

$\ldots \wedge$
$(s[ite(\mathrm{s}[0] \neq a, 0, ite(s[0] = a \wedge s[1] \neq a, 1,2)) + 0] \neq a) \vee$
$(s[ite(\mathrm{s}[0] \neq a, 0, ite(s[0] = a \wedge s[1] \neq a, 1,2)) + 0] = a \wedge s[ite(\mathrm{s}[0] \neq a, 0, ite(s[0] = a \wedge s[1] \neq a, 1,2)) + 1] \neq a) \vee$
$(s[ite(\mathrm{s}[0] \neq a, 0, ite(s[0] = a \wedge s[1] \neq a, 1,2)) + 0] = a \wedge s[ite(\mathrm{s}[0] \neq a, 0, ite(s[0] = a \wedge s[1] \neq a, 1,2)) + 1] = a \wedge s[ite(\mathrm{s}[0] \neq a, 0, ite(s[0] = a \wedge s[1] \neq a, 1,2)) + 2] \neq a)$

## Value of `m`

$ite($
$\mathrm{s}[ite(\mathrm{s}[0] \neq a, 0, ite(s[0] = a \wedge s[1] \neq a, 1,2)) + 0] \neq a,$
$0,$
$ite($
$s[ite(\mathrm{s}[0] \neq a, 0, ite(s[0] = a \wedge s[1] \neq a, 1,2)) + 0] = a \wedge s[ite(\mathrm{s}[0] \neq a, 0, ite(s[0] = a \wedge s[1] \neq a, 1,2)) + 1] \neq a,$
$1,$
$2$
$)$
$)$

# State Merging with Quantifiers

Merging the path constraints

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```

# State Merging with Quantifiers

Merging the path constraints

$$\begin{aligned}&(s[0] \neq a) \vee \\&(s[0] = a \wedge s[1] \neq a) \vee \\&(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)\end{aligned}$$

# State Merging with Quantifiers

Merging the path constraints

$$
\begin{aligned}
&(s[0] \neq a) \vee \\
&(s[0] = a \wedge s[1] \neq a) \vee \\
&(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)
\end{aligned}
$$

# State Merging with Quantifiers

Merging the path constraints

$$(s[0] \neq a) \lor$$
$$(s[0] = a \land s[1] \neq a) \lor$$
$$(s[0] = a \land s[1] = a \land s[2] \neq a)$$

$$s[0] = a \land \cdots \land s[i-1] = a \land s[i] \neq a$$

# State Merging with Quantifiers

Merging the path constraints

$$(s[0] \neq a) \vee$$
$$(s[0] = a \wedge s[1] \neq a) \vee$$
$$(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)$$

$$s[0] = a \wedge \cdots \wedge s[i-1] = a \wedge s[i] \neq a$$

$$\Updownarrow$$

$$(\forall x.\, 1 \leq x \leq i \rightarrow s[x-1] = a) \wedge s[i] \neq a$$

*bound* variable

# State Merging with Quantifiers

Merging the path constraints

$$(s[0] \neq a) \lor$$
$$(s[0] = a \land s[1] \neq a) \lor$$
$$(s[0] = a \land s[1] = a \land s[2] \neq a)$$

$$\Updownarrow$$

$$\big((\forall x.\, 1 \leq x \leq 0 \rightarrow s[x-1] = a) \land s[0] \neq a\big) \lor$$
$$\big((\forall x.\, 1 \leq x \leq 1 \rightarrow s[x-1] = a) \land s[1] \neq a\big) \lor$$
$$\big((\forall x.\, 1 \leq x \leq 2 \rightarrow s[x-1] = a) \land s[2] \neq a\big)$$

# State Merging with Quantifiers

Merging the path constraints

$$
\begin{aligned}
&(s[0] \neq a) \lor \\
&(s[0] = a \land s[1] \neq a) \lor \\
&(s[0] = a \land s[1] = a \land s[2] \neq a)
\end{aligned}
$$

$$\Updownarrow$$

$$
\begin{aligned}
&\big((\forall x.\, 1 \leq x \leq 0 \rightarrow s[x-1] = a) \land s[0] \neq a\big) \lor \\
&\big((\forall x.\, 1 \leq x \leq 1 \rightarrow s[x-1] = a) \land s[1] \neq a\big) \lor \\
&\big((\forall x.\, 1 \leq x \leq 2 \rightarrow s[x-1] = a) \land s[2] \neq a\big)
\end{aligned}
$$

$$\Updownarrow$$

$$0 \leq i \leq 2 \land (\forall x.\, 1 \leq x \leq i \rightarrow s[x-1] = a) \land s[i] \neq a$$

*fresh free* variable

# State Merging with Quantifiers

Merging memory

$$0 \le i \le 2 \wedge (\forall x.\, 1 \le x \le i \rightarrow s[x-1] = a) \wedge s[i] \ne a$$

$$\text{merged value of } \mathsf{n} \left\{ \begin{array}{l} ite( \\ \quad \mathrm{s}[0] \ne a, \\ \quad 0, \\ \quad ite( \\ \qquad s[0] = a \wedge s[1] \ne a, \\ \qquad 1, \\ \qquad 2 \\ \quad ) \\ ) \end{array} \right.$$

# State Merging with Quantifiers

Merging memory

$0 \leq i \leq 2 \land (\forall x. 1 \leq x \leq i \rightarrow s[x-1] = a) \land s[i] \neq a$

$$\text{merged value of } \mathbf{n} \begin{cases} ite( \\ \quad \mathrm{s}[0] \neq a, \\ \quad 0, \\ \quad ite( \\ \quad\quad s[0] = a \land s[1] \neq a, \\ \quad\quad 1, \\ \quad\quad 2 \\ \quad ) \\ ) \end{cases}$$

# State Merging with Quantifiers

Merging memory

$$0 \leq i \leq 2 \land (\forall x. 1 \leq x \leq i \rightarrow s[x-1] = a) \land s[i] \neq a$$

merged value of **n** $\left\{ \begin{array}{l} ite( \\ \quad s[0] \neq a, \\ \quad 0, \\ \quad ite( \\ \quad\quad s[0] = a \land s[1] \neq a, \\ \quad\quad 1, \\ \quad\quad 2 \\ \quad ) \\ ) \end{array} \right. \quad \Longrightarrow \quad i$

# State Merging with Quantifiers

Merging the path constraints

```
int strspn(char *s, char c) {
  int count = 0;
  while (s[count] == c) {
    count++;
  }
  return count;
}

// symbolic, null-terminated
char s[3];
int n = strspn(s, ‘a’);
int m = strspn(s + n, ‘b’);
...
```

# State Merging with Quantifiers

Path constraints

$$\ldots \wedge 0 \leq j \leq 2 \wedge (\forall x.\, 1 \leq x \leq j \rightarrow s[i + x - 1] = b) \wedge s[i + j] \neq b$$

Value of `m`

$j$

# Synthesizing Quantified Constraints

path constrains

$(s[0] \neq a)$
$(s[0] = a \wedge s[1] \neq a)$
$(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)$

# Synthesizing Quantified Constraints

path constrains

$(s[0] \neq a)$
$(s[0] = a \wedge s[1] \neq a)$
$(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)$

$\Downarrow$

abstraction

$\beta$
$\alpha\beta$
$\alpha\alpha\beta$

# Synthesizing Quantified Constraints

path constrains

$(s[0] \neq a)$
$(s[0] = a \wedge s[1] \neq a)$
$(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)$

$$\Downarrow$$

abstraction

$$\begin{array}{ll} \beta & \alpha^0\beta \\ \alpha\beta & \alpha^1\beta \\ \alpha\alpha\beta & \alpha^2\beta \end{array} \left.\vphantom{\begin{array}{l} \beta \\ \beta \\ \beta \end{array}}\right\} \alpha^*\beta$$

# Synthesizing Quantified Constraints

path constrains

$(s[0] \neq a)$
$(s[0] = a \land s[1] \neq a)$
$(s[0] = a \land s[1] = a \land s[2] \neq a)$

$\Downarrow$

abstraction

$$\begin{array}{ll} \beta & \alpha^0\beta \\ \alpha\beta & \alpha^1\beta \\ \alpha\alpha\beta & \alpha^2\beta \end{array} \Bigg\} \ \alpha^*\beta$$

$\Rightarrow$

synthesis constraints

$$\left.\begin{array}{l} \varphi_\alpha(1) \stackrel{\text{def}}{=} s[0] = a \\ \varphi_\alpha(2) \stackrel{\text{def}}{=} s[1] = a \end{array}\right. \Rightarrow \varphi_\alpha(x) \stackrel{\text{def}}{=} s[x-1] = a$$

$$\left.\begin{array}{l} \varphi_\beta(0) \stackrel{\text{def}}{=} s[0] \neq a \\ \varphi_\beta(1) \stackrel{\text{def}}{=} s[1] \neq a \\ \varphi_\beta(2) \stackrel{\text{def}}{=} s[2] \neq a \end{array}\right. \Rightarrow \varphi_\beta(x) \stackrel{\text{def}}{=} s[x] \neq a$$

# Synthesizing Quantified Constraints

path constrains

$(s[0] \neq a)$
$(s[0] = a \wedge s[1] \neq a)$
$(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)$

$\Downarrow$

abstraction

$$\left.\begin{array}{ll} \beta & \alpha^0\beta \\ \alpha\beta & \alpha^1\beta \\ \alpha\alpha\beta & \alpha^2\beta \end{array}\right\} \alpha^*\beta \quad \Rightarrow$$

synthesis constraints

$$\begin{array}{l} \varphi_\alpha(1) \stackrel{\text{def}}{=} s[0] = a \\ \varphi_\alpha(2) \stackrel{\text{def}}{=} s[1] = a \end{array} \Rightarrow \varphi_\alpha(x) \stackrel{\text{def}}{=} s[x-1] = a$$

$$\begin{array}{l} \varphi_\beta(0) \stackrel{\text{def}}{=} s[0] \neq a \\ \varphi_\beta(1) \stackrel{\text{def}}{=} s[1] \neq a \\ \varphi_\beta(2) \stackrel{\text{def}}{=} s[2] \neq a \end{array} \Rightarrow \varphi_\beta(x) \stackrel{\text{def}}{=} s[x] \neq a$$

$\Uparrow$

quantified path constraints

$0 \leq i \leq 2 \wedge (\forall x.\, 1 \leq x \leq i \rightarrow \varphi_\alpha[x]) \wedge \varphi_\beta[i]$

# Synthesizing Quantified Constraints

path constrains

$$(s[0] \neq a) \vee$$
$$(s[0] = a \wedge s[1] \neq a) \vee$$
$$(s[0] = a \wedge s[1] = a \wedge s[2] \neq a)$$

$\Leftrightarrow$

quantified path constraints

$$0 \leq i \leq 2 \wedge (\forall x. 1 \leq x \leq i \rightarrow \varphi_\alpha[x]) \wedge \varphi_\beta[i]$$

abstraction

$$\beta \quad \alpha^0\beta$$
$$\alpha\beta \quad \alpha^1\beta$$
$$\alpha\alpha\beta \quad \alpha^2\beta$$
$$\left.\begin{matrix}\alpha^0\beta \\ \alpha^1\beta \\ \alpha^2\beta\end{matrix}\right\} \alpha^*\beta$$

synthesis constraints

$$\varphi_\alpha(1) \stackrel{\text{def}}{=} s[0] = a$$
$$\varphi_\alpha(2) \stackrel{\text{def}}{=} s[1] = a \quad \Rightarrow \quad \varphi_\alpha(x) \stackrel{\text{def}}{=} s[x-1] = a$$

$$\varphi_\beta(0) \stackrel{\text{def}}{=} s[0] \neq a$$
$$\varphi_\beta(1) \stackrel{\text{def}}{=} s[1] \neq a \quad \Rightarrow \quad \varphi_\beta(x) \stackrel{\text{def}}{=} s[x] \neq a$$
$$\varphi_\beta(2) \stackrel{\text{def}}{=} s[2] \neq a$$

# Additional Contributions

Specialized solving procedure

- Efficiently solving quantified formulas

Incremental state merging

- Handling complex loops (exponential execution trees)

*More details in the paper…*

# Evaluation

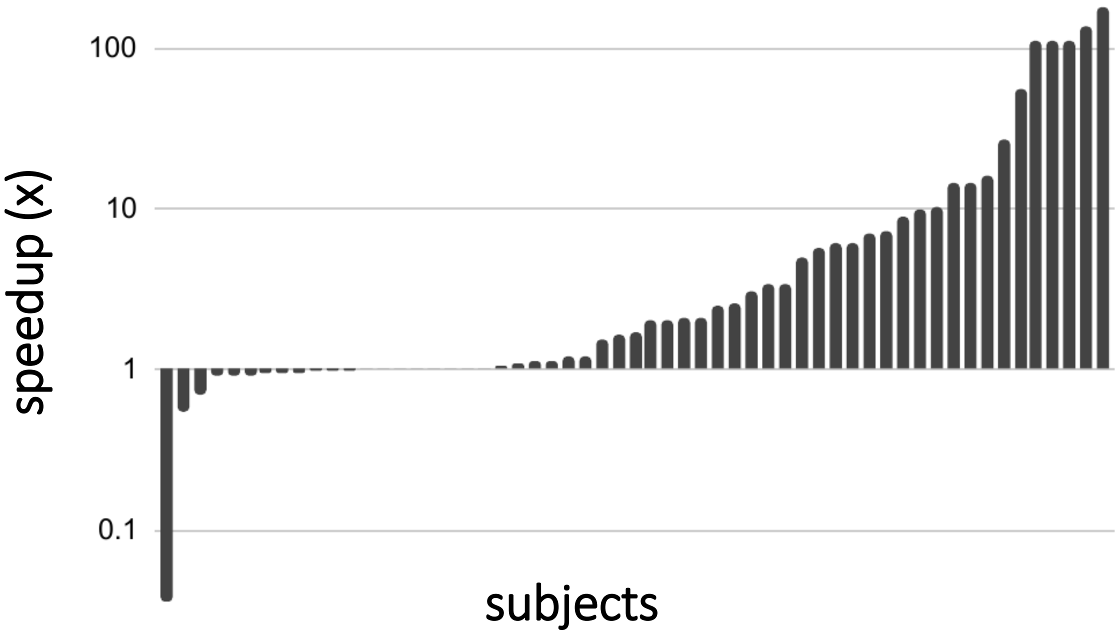
Implementation

- On top of ***KLEE***

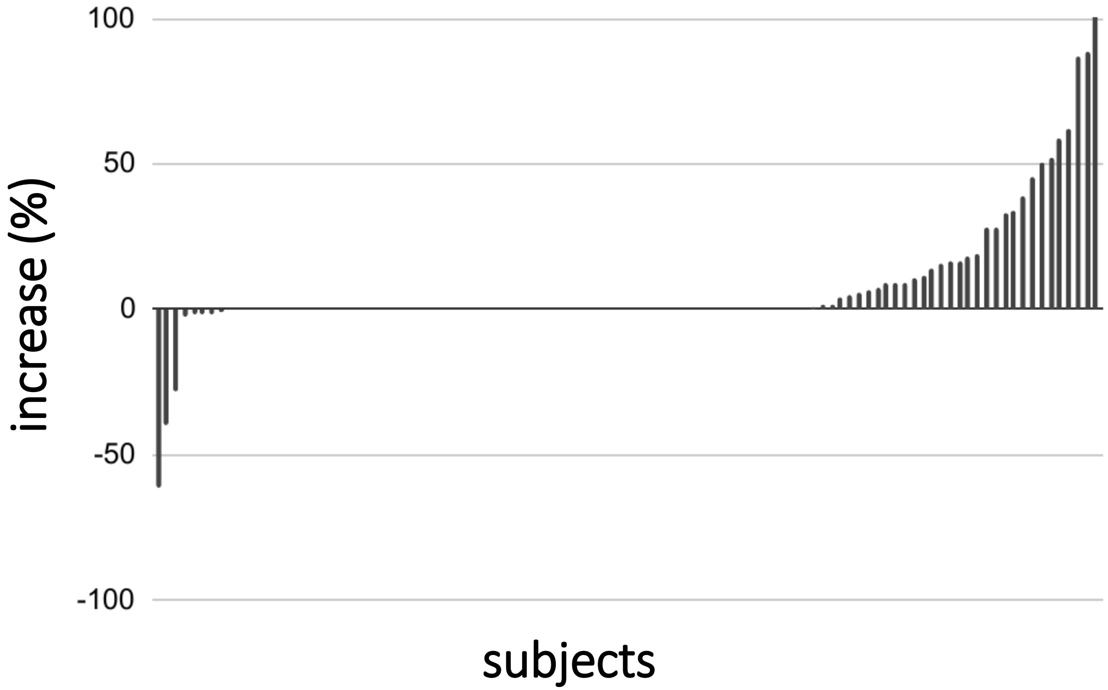
Benchmarks

- GNU oSIP *(35 subjects)*
- wget *(31 subjects)*
- GNU libtasn1 *(13 subjects)*
- libpng *(12 subjects)*
- APR (Apache Portable Runtime) *(20 subjects)*
- json-c *(5 subjects)*
- busybox *(30 subjects)*

# Evaluation: Analysis Time



# Evaluation: Coverage



# Found Bugs

Detected bugs in ***klee-uclibc*** in the experiments with ***busybox***

- Two *memory out-of-bound's*
- Confirmed and fixed

# Summary

- State merging using quantified constraints
- Specialized solving procedure for quantified constraints
- Evaluated on real-world benchmarks
- Found bugs

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