Incorporating Native String Reasoning in Symbolic Execution of C Programs

Rachel Cleaveland & Clark Barrett

Meeting on String Constraints and Applications

Zagreb, Croatia

July 22, 2025

Stanford University

Motivating Example

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

Motivating Example

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

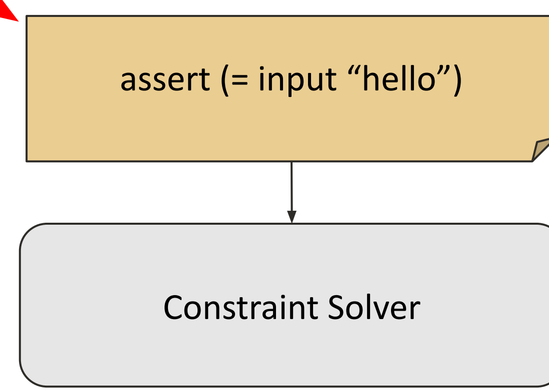
Motivating Example

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

assert (= input “hello”)

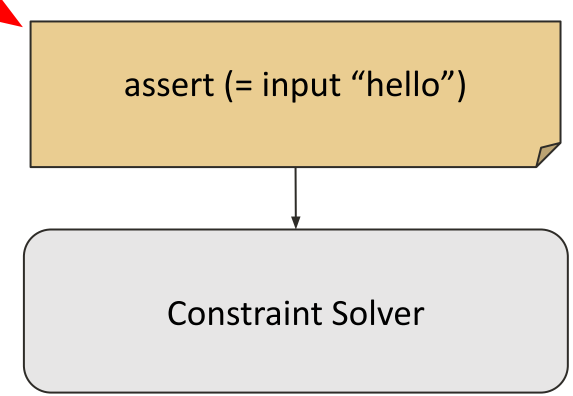
Motivating Example

```
if (strcmp(input,“hello”) == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```



Motivating Example

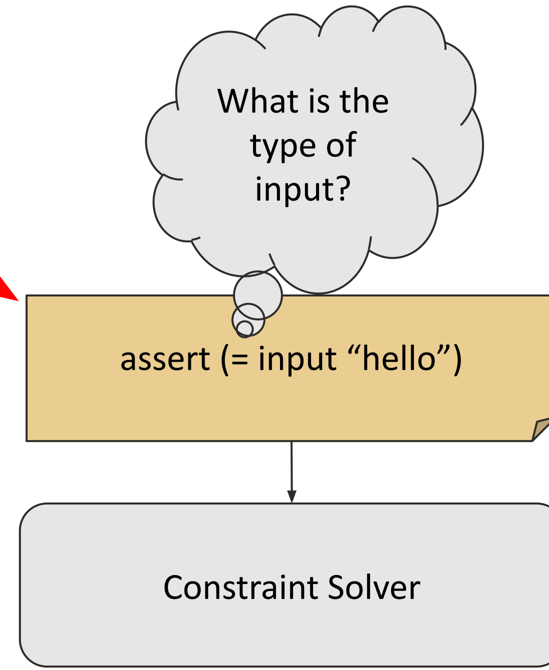
```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```



Via **symbolic execution:**
new input = "hello"

Motivating Example

```
if (strcmp(input,“hello”) == 0) {
    // Code A – contains a bug!
} else {
    // Code B
}
```



Via **symbolic execution:**
new input = “hello”

Motivating Example

State of the art engines:
theory of bitvectors

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

Motivating Example

State of the art engines:
theory of bitvectors

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

(declare-fun input () (_ BitVec **??**))

(assert (= input **??**))

Motivating Example

State of the art engines:
theory of bitvectors

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

h	e	l	l	o
0x68	0x65	0x6C	0x6C	0x6F

```
(declare-fun input () (_ BitVec ??))

(assert (= input ??))
```

Motivating Example

State of the art engines:
theory of bitvectors

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

h	e	l	l	o
0x68	0x65	0x6C	0x6C	0x6F

```
(declare-fun input () (_ BitVec ??))

(assert (= input #x68656C6C6F))
```

Motivating Example

State of the art engines:
theory of bitvectors

```
if (strcmp(input,“hello”) == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

h	e	l	l	o
0x68	0x65	0x6C	0x6C	0x6F

5 characters = 40 bits

```
(declare-fun input () (_ BitVec 40))

(assert (= input #x68656C6C6F))
```

Motivating Example

(declare-fun input () (_ BitVec **40**))

(assert (= input **#x68656C6C6F**))

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else if (strcmp(input,"helloworld") == 0) {
    // Code B
} else {
    // Code C
}
```

Motivating Example

```
(declare-fun input () (_ BitVec 40))

(assert (= input #x68656C6C6F776F726C64))
```

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else if (strcmp(input,"helloworld") == 0) {
    // Code B
} else {
    // Code C
}
```

Motivating Example

```
(declare-fun input () (_ BitVec 40))

(assert (= input #x68656C6C6F776F726C64))
```

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else if (strcmp(input,"helloworld") == 0) {
    // Code B
} else {
    // Code C
}
```

Motivating Example

(declare-fun input () (_ BitVec 40))

(assert (= input **#x68656C6C6F776F726C64**))

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else if (strstr(input,"helloworld")) {
    // Code B
} else {
    // Code C
}
```

Find first occurrence of "helloworld" in input

Motivating Example

(declare-fun input () (_ BitVec 40))

(assert (**??**))

```
if (strcmp(input,"hello") == 0) {
    // Code A - contains a bug!
} else if (strstr(input,"helloworld")) {
    // Code B
} else {
    // Code C
}
```

Find first occurrence of "helloworld" in input

Strings in Symbolic Execution

Symbolic execution engines should be able to **encode the full semantics of a program**, which includes being able to natively reason about **inputs of an unbounded length** and **string functions**.

Solution: use a string solver to represent symbolic data as strings

Prior Approaches

Many string-based symbolic engines have already been developed.

- Shannon’07, Shannon’09, Saxena’10, Redelinghuys’12, Ghosh’13, Bang’16, Loring’17, Reynolds’17 Amadini’19

Limitations:

1. All are written for Java or Python programs
2. Most use bespoke string solvers
3. Only evaluated on small programs and test suites
4. Some are limited to bounded-length strings

This Work

We present **SymCC-STR**, a symbolic execution engine for string-manipulating C programs that represents symbolic data as both bitvectors and strings.

Key Performance Features

- Hybrid data representation
- Manual assertion rewrites
- Custom string inequality solving pass
- Runtime string-backend disabling

Implementation: Concolic Execution

SymCC-STR builds on the **concolic execution** engine **SymCC** [Poeplau'20].

Implementation: Concolic Execution

SymCC-STR builds on the **concolic execution** engine **SymCC** [Poeplau'20].

Concrete input guides execution

Implementation: Concolic Execution

SymCC-STR builds on the **concolic execution** engine **SymCC** [Poeplau'20].

Concrete input guides execution

Input: "hello"

```
if (strcmp(buffer,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

Implementation: Concolic Execution

SymCC-STR builds on the **concolic execution** engine **SymCC** [Poeplau'20].

Concrete input guides execution

Input: "hello"

```
if (strcmp(buffer,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

Concrete input takes first branch

Implementation: Concolic Execution

SymCC-STR builds on the **concolic execution** engine **SymCC** [Poeplau'20].

Concrete input guides execution

Input: "hello"

```
if (strcmp(buffer,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

Concrete input takes first branch

New Input: "h"
(Takes else branch)

Implementation: Concolic Execution

SymCC-STR builds on the **concolic execution** engine **SymCC** [Poeplau'20].

Concrete input guides execution

Input: "hello"

```
if (strcmp(buffer,"hello") == 0) {
    // Code A - contains a bug!
} else {
    // Code B
}
```

Concrete input takes first branch

Code B not explored on this execution!

New Input: "h"
(Takes else branch)

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: "hello"

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] == 'g') {
    // Code B
}
```

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: "hello"

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] == 'g') {
    // Code B
}
```

(declare-fun input-bv () (_ BitVec 40))

(declare-fun input-str () String)

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: “hello”

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] == 'g') {
    // Code B
}
```

(declare-fun input-bv () (_ BitVec 40))

(assert (= ((_ extract 7 0) input-bv) #x67))

(declare-fun input-str () String)

(assert (= (str.substr input-str 0 1) “g”))

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: "hello"

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] ...
    // Code B
}
```

(declare-fun input-bv () (_ BitVec 40))

(assert (= ((_ extract 7 0) input-bv) #x67))

(declare-fun input-str () String)

(assert (= (str.substr input-str 0 1) "g"))

String assertions are only solved if the bitvector assertions are unsatisfiable.

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: “hello”

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] 
    // Code B
}
```

(declare-fun input-bv () (_ BitVec 40))

(assert (= ((_ extract 7 0) input-bv) #x67))

SAT: “g”

String assertions are only solved if the bitvector assertions are unsatisfiable.

(declare-fun input-str () String)

(assert (= (str.substr input-str 0 1) “g”))

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: "hello"

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] ...
    // Code B
}
```

(declare-fun input-bv () (_ BitVec 40))

(assert (= ((_ extract 7 0) input-bv) #x67))

SAT: "g"

(declare-f... r

(assert (= (str.s ... tr 0 1) "g"))

String assertions are only solved if the bitvector assertions are unsatisfiable.

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: “hello”

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] == 'g') {
    // Code B
}
```

(declare-fun input-bv () (_ BitVec 40))

(assert (not (= ((_ extract 7 0) input-bv) #x67)))

(declare-fun input-str () String)

(assert (not (= (str.substr input-str 0 1) “g”)))

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: “hello”

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] == 'g') {
    // Code B
}
```

```
(declare-fun input-bv () (_ BitVec 40))

(assert (not (= ((_ extract 7 0) input-bv) #x67)))
(assert (= #x00 #x67))
```

```
(declare-fun input-str () String)

(assert (not (= (str.substr input-str 0 1) “g”)))
(assert (= (str.substr input-str 5 1) “g”))
```

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: "hello"

Symbolic inputs are the length of the concrete input

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] == 'g') {
    // Code B
}
```

```
(declare-fun input-bv () (_ BitVec 40))

(assert (not (= ((_ extract 7 0) input-bv)
#x67)))
(assert (= #x00 #x67))
```

```
(declare-fun input-str () String)

(assert (not (= (str.substr input-str 0 1)
"g")))
(assert (= (str.substr input-str 5 1) "g"))
```

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: "hello"

Symbolic inputs are the length of the concrete input

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] == 'g') {
    // Code B
}
```

```
(declare-fun input-bv () (_ BitVec 40))

(assert (not (= ((_ extract 7 0) input-bv) #x67)))
(assert (= #x00 #x67))
```

UNSAT

```
(declare-fun input-str () String)

(assert (not (= (str.substr input-str 0 1) "g")))
(assert (= (str.substr input-str 5 1) "g"))
```

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: “hello”

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5] == 'g') {
    // Code B
}
```

Symbolic inputs are the length of the concrete input

(declare-fun input-bv () (_ BitVec 40))

(assert (not (= ((_ extract 7 0) input-bv) #x67)))
(assert (= #x00 #x67))

UNSAT

(declare-fun input-str () String)

(assert (not (= (str.substr input-str 0 1) “g”)))
(assert (= (str.substr input-str 5 1) “g”))

SAT: “hellog”

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: “hello”

```
// Read in user input
...
if (input[0] == 'g') {
    // Code A
}
if (input[5]
    // Code B
}
```

Symbolic inputs are the length of the concrete input

(declare-fun input-bv () (_ BitVec 40))

(assert (not (= ((_ extract 7 0) input-bv) #x67)))
(assert (= #x00 #x67))

UNSAT

Advantage: performance of bitvector solver with expressivity of string solver.

(declare-fun input-str () String)

(assert (not (= (str.substr input-str 0 1) “g”)))
(assert (= (str.substr input-str 5 1) “g”))

SAT: “hellog”

Implementation: Manual Rewrites

In practice, string assertions rapidly became complex and solving them became intractable.

Observation: assertions could be simplified given the knowledge from prior assertions.

Implementation: Manual Rewrites

In practice, string assertions rapidly became complex and solving them became intractable.

Observation: assertions could be simplified given the knowledge from prior assertions.

```
len(ITE(cond,A,B))
```

Implementation: Manual Rewrites

In practice, string assertions rapidly became complex and solving them became intractable.

Observation: assertions could be simplified given the knowledge from prior assertions.

$$\texttt{len(ITE(cond,A,B))} \xrightarrow{\texttt{if len(A) == len(B)}} \texttt{len(A)}$$

Implementation: Manual Rewrites

In practice, string assertions rapidly became complex and solving them became intractable.

Observation: assertions could be simplified given the knowledge from prior assertions.

$$\texttt{len(ITE(cond,A,B))} \xrightarrow{\texttt{if len(A) == len(B)}} \texttt{len(A)}$$

Implementation: Manual Rewrites

In practice, string assertions rapidly became complex and solving them became intractable.

Observation: assertions could be simplified given the knowledge from prior assertions.

$$\texttt{len(ITE(cond,A,B))} \xrightarrow{\texttt{if len(A) == len(B)}} \texttt{len(A)}$$

“Mini-check”:

Implementation: Manual Rewrites

In practice, string assertions rapidly became complex and solving them became intractable.

Observation: assertions could be simplified given the knowledge from prior assertions.

`len(ITE(cond,A,B))` —— `if len(A) == len(B)` ——→ `len(A)`

"Mini-check":

`!(len(A) == len(B))` ——→ **String Solver**

Implementation: Manual Rewrites

In practice, string assertions rapidly became complex and solving them became intractable.

Observation: assertions could be simplified given the knowledge from prior assertions.

`len(ITE(cond,A,B))` —`if len(A) == len(B)` ?→ `len(A)`

"Mini-check":

`!(len(A) == len(B))` → **String Solver**

(Timeout: 100ms)

Implementation: Manual Rewrites

In practice, string assertions rapidly became complex and solving them became intractable.

Observation: assertions could be simplified given the knowledge from prior assertions.

$$\texttt{len(ITE(cond,A,B))} \xrightarrow{\texttt{if len(A) == len(B)}\ ?} \texttt{len(A)}$$

"Mini-check":

$$\texttt{!(len(A) == len(B))} \longrightarrow \boxed{\text{String Solver}} \longrightarrow \textbf{UNSAT}$$

(Timeout: 100ms)

Implementation: Manual Rewrites

In practice, string assertions rapidly became complex and solving them became intractable.

Observation: assertions could be simplified given the knowledge from prior assertions.

`len(ITE(cond,A,B))` —`if len(A) == len(B)`?→ `len(A)`

"Mini-check":

`!(len(A) == len(B))` → **String Solver** → **UNSAT** → Perform the rewrite

(Timeout: 100ms)

Implementation: Manual Rewrites

String Rewrites:

$$
\begin{aligned}
\texttt{indexOf}(\texttt{con}(A,B),0,c) &\rightarrow -1 && \text{if } \texttt{indexOf}(A,0,c) = -1 \wedge \texttt{indexOf}(B,0,c) = -1 \\
\texttt{indexOf}(\texttt{con}(A,B),0,c) &\rightarrow \texttt{indexOf}(A,0,c) && \text{if } \texttt{indexOf}(A,0,c) \neq -1 \\
\texttt{indexOf}(\texttt{con}(A,B),0,c) &\rightarrow \texttt{len}(A) + \texttt{indexOf}(B,0,c) && \text{if } x + y \leq \texttt{len}(A) \\
\texttt{substr}(A,x,y) &\rightarrow A && \text{if } x = 0 \wedge \texttt{len}(A) = y \\
\texttt{substr}(\texttt{con}(A,B),x,y) &\rightarrow \texttt{substr}(A,x,y) && \text{if } x + y \leq \texttt{len}(A) \\
\texttt{substr}(\texttt{con}(A,B),x,y) &\rightarrow \texttt{substr}(B,x - \texttt{len}(A),y) && \text{if } x \geq \texttt{len}(A) \\
\texttt{substr}(\texttt{substr}(A,x,y),a,b) &\rightarrow \texttt{substr}(A,x+a,\texttt{min}(y,b)) && \text{if } \top \\
\texttt{len}(\texttt{substr}(A,x,y)) &\rightarrow y && \text{if } \texttt{len}(A) \geq x + y \\
\texttt{len}(\texttt{ITE}(cond,A,B)) &\rightarrow \texttt{len}(A) && \text{if } \texttt{len}(A) = \texttt{len}(B) \\
A == B &\rightarrow \texttt{false} && \text{if } \texttt{len}(B) == 1 \wedge !\texttt{contains}(A,B)
\end{aligned}
$$

Arithmetic Rewrites: short-circuit integer overflow conditionals, sign extension conditionals, etc.

Implementation: Disabling the String Backend

In practice, **most new inputs generated** by the string backend are done so **early in the program**. Later in the program, most string queries time out.

Design Feature: string backend gets disabled when less than a user-defined percentage of the mini-checks are succeeding.

Implementation: Custom String Solving Pass

```
1: substr(S,0,9) < "Document"
2: "Feature" < substr(S,0,8)
```

Implementation: Custom String Solving Pass

```
1: substr(S,0,9) < "Document"
2: "Feature" < substr(S,0,8)
```

UNSAT: `substr(S,0,9) < "Document" < "Feature" < substr(S,0,8)` **≤** `substr(S,0,9)`

Implementation: Custom String Solving Pass

```
1: substr(S,0,9) < "Document"
2: "Feature" < substr(S,0,8)
```

UNSAT: `substr(S,0,9) < "Document" < "Feature" < substr(S,0,8)` **≤** `substr(S,0,9)`

TIMEOUT (10 seconds)

Implementation: Custom String Solving Pass

1: substr(S,0,9) < "Document"

2: "Feature" < substr(S,0,8)

UNSAT: substr(S,0,9) < "Document" < "Feature" < substr(S,0,8) ≤ substr(S,0,9)

TIMEOUT (10 seconds)

```
(set-logic ALL)
(set-option :produce-models true)

(declare-fun |s1| () String)
(declare-fun |s2| () String)

(assert (not (=> (str.<= (str.substr s1 0 9) s2)
                 (str.<= (str.substr s1 0 8) s2))))

(check-sat)
(get-model)
```

Implementation: Custom String Solving Pass

1: substr(S,0,9) < “Document”
2: “Feature” < substr(S,0,8)

UNSAT: substr(S,0,9) < “Document” < “Feature” < substr(S,0,8) ≤ substr(S,0,9)

TIMEOUT (10 seconds)

```
(set-logic ALL)
(set-option :produce-models true)

(declare-fun |s1| () String)
(declare-fun |s2| () String)

(assert (not (=> (str.<= (str.substr s1 0 9) s2)
                 (str.<= (str.substr s1 0 8) s2))))

(check-sat)
(get-model)
```

Is it possible that s1[0:9] ≤ s2, but s1[0:8] > s2?

Implementation: Custom String Solving Pass

1: substr(S,0,9) < "Document"
2: "Feature" < substr(S,0,8)

UNSAT: substr(S,0,9) < "Document" < "Feature" < substr(S,0,8) ≤ substr(S,0,9)

TIMEOUT (10 seconds)

```
(set-logic ALL)
(set-option :produce-models true)

(declare-fun |s1| () String)
(declare-fun |s2| () String)

(assert (not (=> (str.<= (str.substr s1 0 9) s2)
                 (str.<= (str.substr s1 0 8) s2))))

(check-sat)
(get-model)
```

Is it possible that s1[0:9] ≤ s2, but s1[0:8] > s2?

TIMEOUT (1 hour)

Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) < "Document"
2: "Feature" < substr(S,0,8)
```

Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) < "Document"
2: "Feature" < substr(S,0,8)
```

String Solving Pass:

Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) < "Document"
2: "Feature" < substr(S,0,8)
```

String Solving Pass:

1. Strings become graph nodes

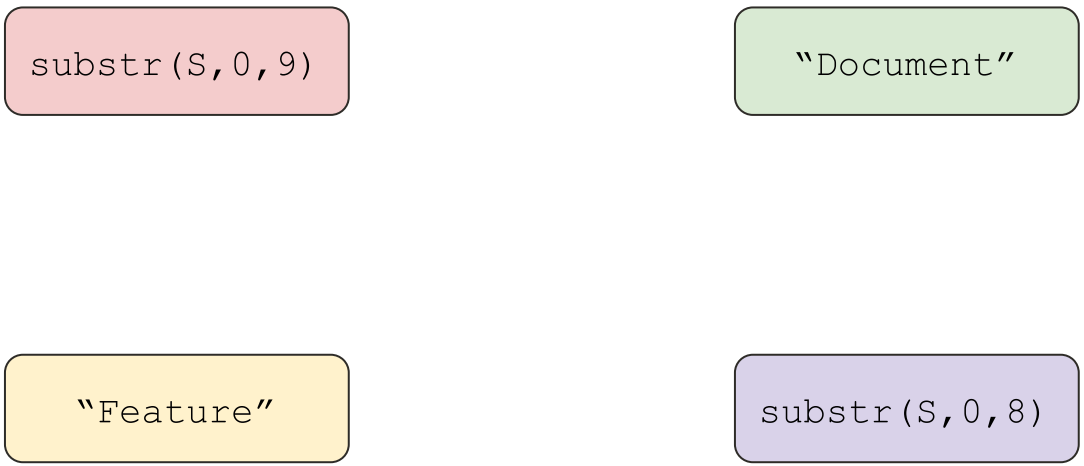
Implementation: Solving UNSAT Inequalities

1: `substr(S,0,9)` ≪ `“Document”’`

2: `“Feature”` ≪ `substr(S,0,8)`

String Solving Pass:

1. Strings become graph nodes

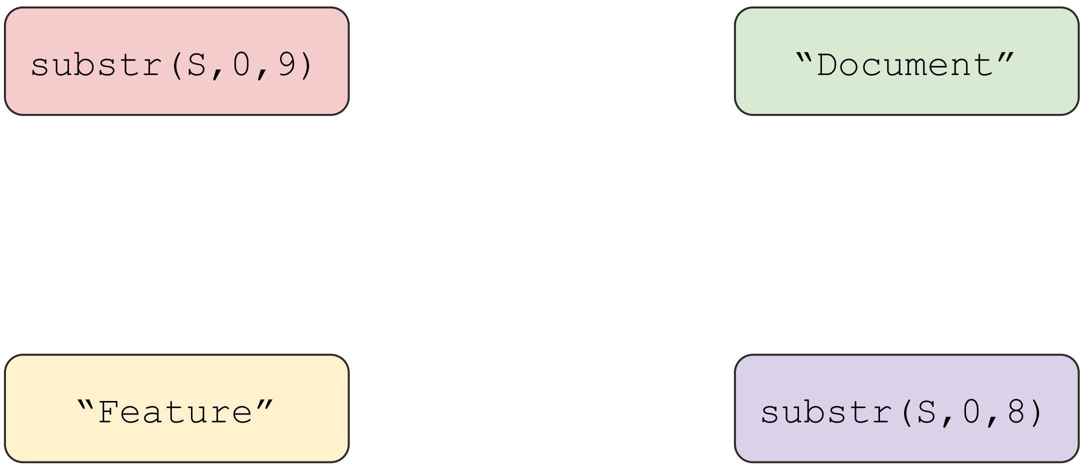


Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) ≪ "Document"
2: "Feature" ≪ substr(S,0,8)
```

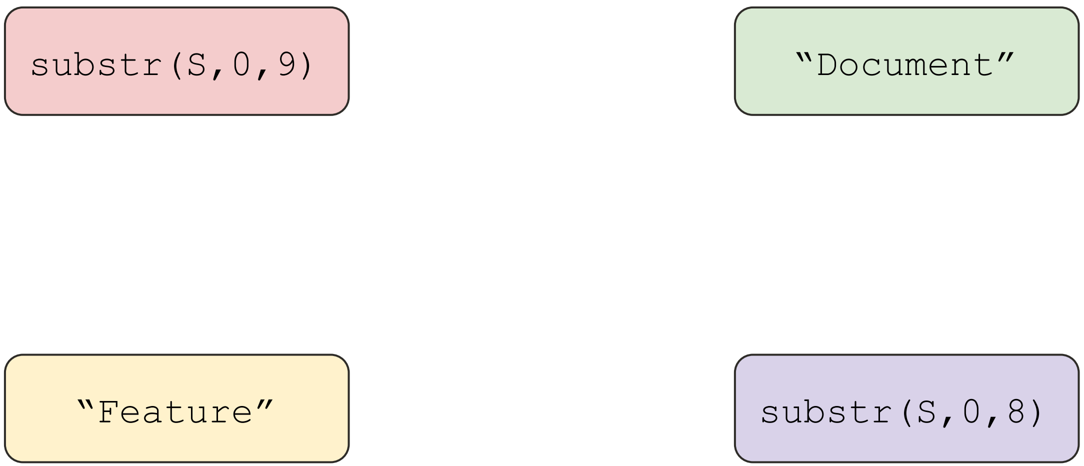
String Solving Pass:

1. Strings become graph nodes
2. Add "strong" (<) and "weak" (≤) edges



Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) ≪ "Document"
2: "Feature" ≪ substr(S,0,8)
```

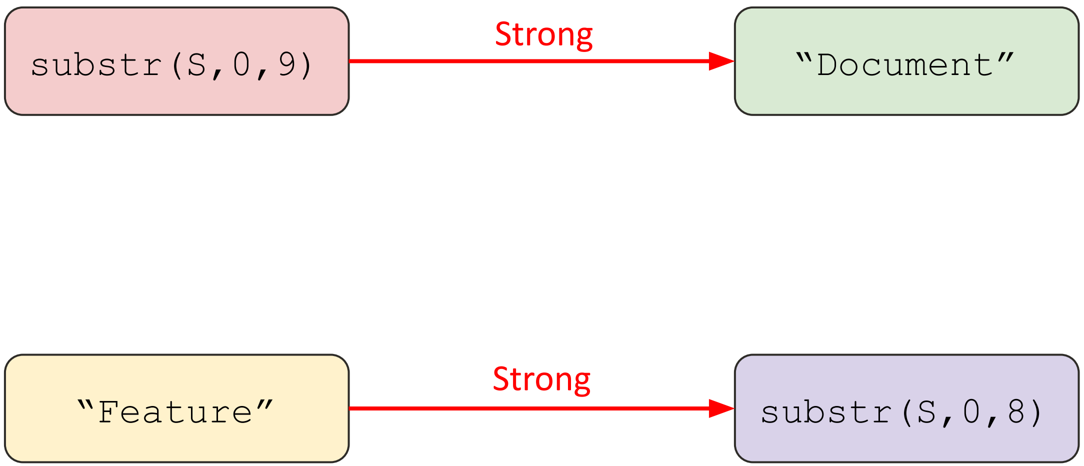


String Solving Pass:

1. Strings become graph nodes
2. Add "strong" (<) and "weak" (≤) edges
    a. From the assertions

Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) ≪ "Document"
2: "Feature" ≪ substr(S,0,8)
```

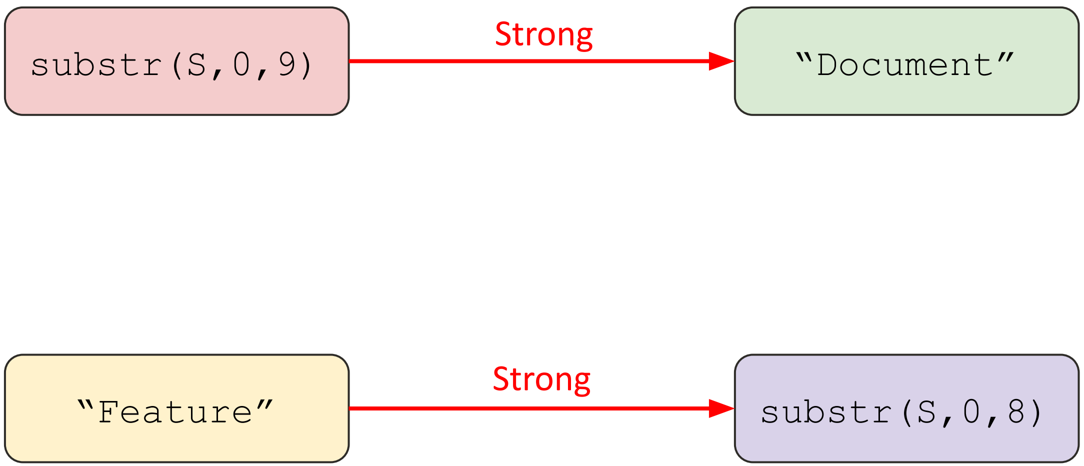


String Solving Pass:

1. Strings become graph nodes
2. Add "strong" (<) and "weak" (≤) edges
    a. From the assertions

Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) ≪ "Document"
2: "Feature" ≪ substr(S,0,8)
```

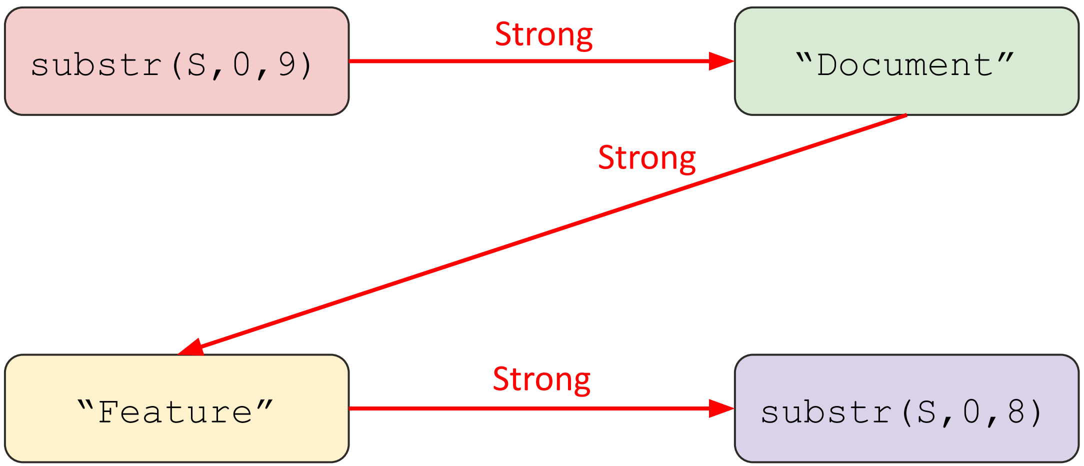


String Solving Pass:

1. Strings become graph nodes
2. Add "strong" (<) and "weak" (≤) edges
   a. From the assertions
   b. From concrete strings

Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) ≪ "Document"
2: "Feature" ≪ substr(S,0,8)
```

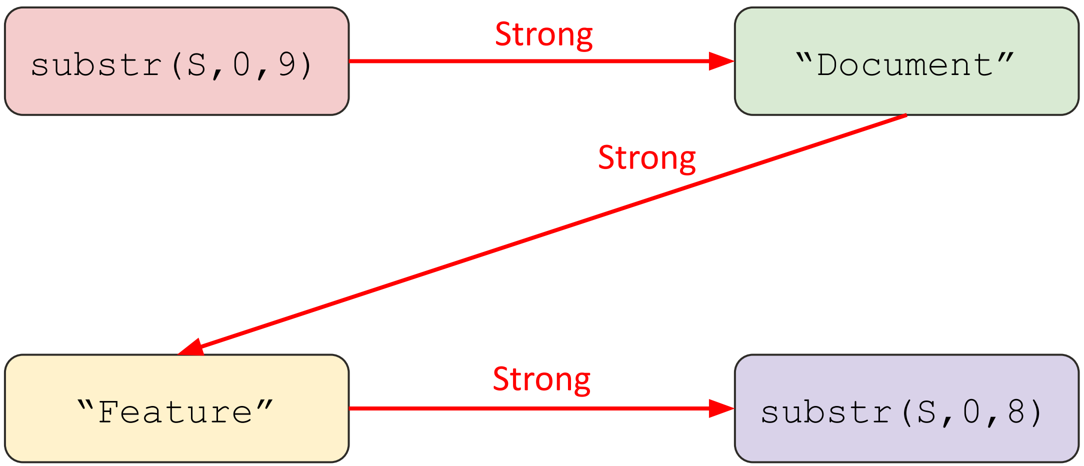


String Solving Pass:

1. Strings become graph nodes
2. Add "strong" (<) and "weak" (≤) edges
    a. From the assertions
    b. From concrete strings

Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) ≪ "Document"
2: "Feature" ≪ substr(S,0,8)
```

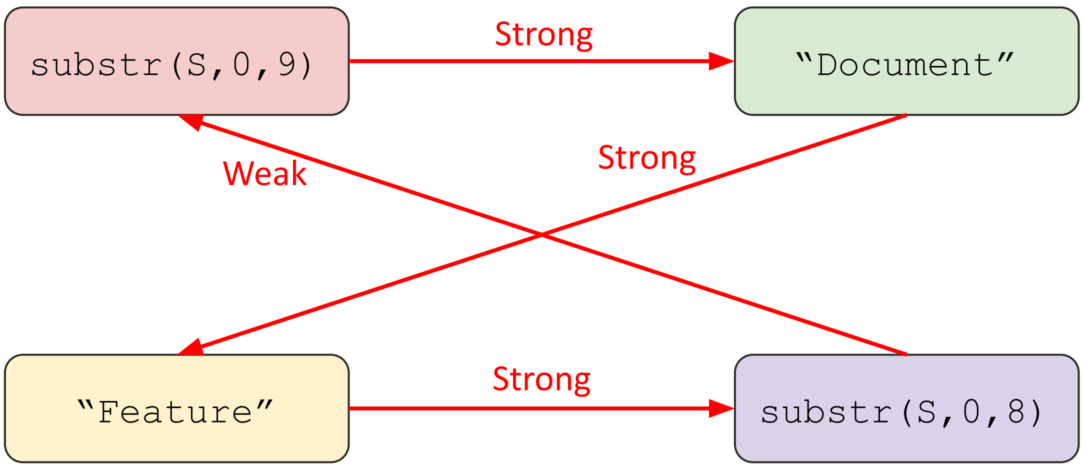


String Solving Pass:

1. Strings become graph nodes
2. Add "strong" (<) and "weak" (≤) edges
   a. From the assertions
   b. From concrete strings
   c. From substrings

Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) ≪ "Document"
2: "Feature" ≪ substr(S,0,8)
```

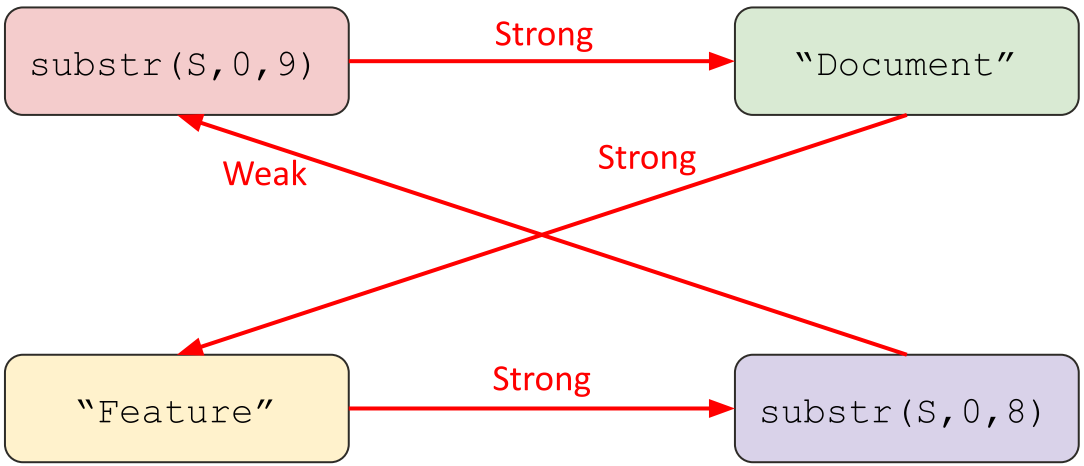


String Solving Pass:

1. Strings become graph nodes
2. Add "strong" (<) and "weak" (≤) edges
    a. From the assertions
    b. From concrete strings
    c. From substrings

Implementation: Solving UNSAT Inequalities

```
1: substr(S,0,9) ≪ "Document"
2: "Feature" ≪ substr(S,0,8)
```



String Solving Pass:

1. Strings become graph nodes
2. Add "strong" (<) and "weak" (≤) edges
   a. From the assertions
   b. From concrete strings
   c. From substrings
3. Assertions are UNSAT if a cycle exists with at least one strong edge.

Evaluation

Used benchmarks from the OSS-Fuzz open-source fuzzing project

- Targeted those that contained a high number of string functions (strcmp, strlen, strchr, etc)
- All are some sort of parser of various file formats
- **Benchmarks**: inih, libspectre, libtasn1, libyaml, openjpeg, speex.

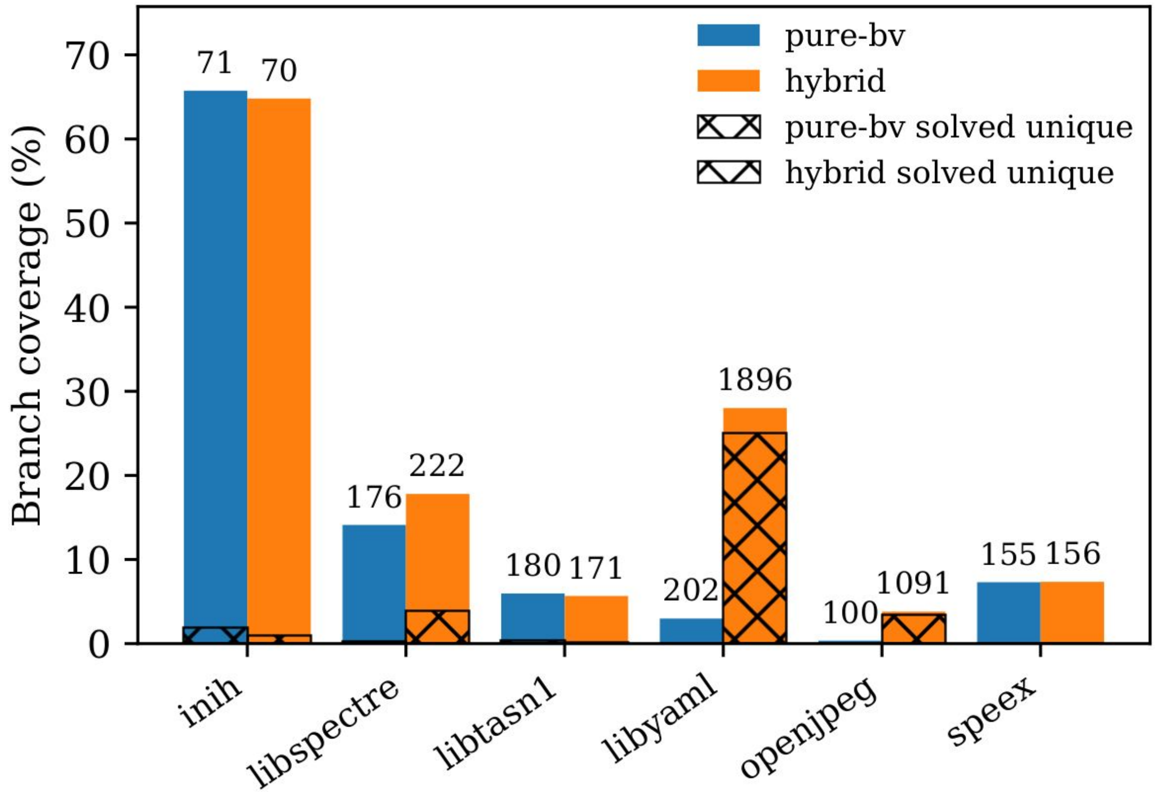
Experiments

RQ1: How does the branch coverage achieved by **hybrid SymCC-STR** compare to the branch coverage achieved by **pure-bv SymCC-STR**?

Results (RQ1: branch coverage)

Experimental Setup

- Feed an initial input into each engine and repeatedly rerun with the new inputs they generate
- Backend solver: cvc5
- Total timeout: 12 hours
- Single-run timeout: 5 minutes
- Disable string solver when only 25% of the mini-checks are being solved.



Experiments

RQ1: How does the branch coverage achieved by **hybrid SymCC-STR** compare to the branch coverage achieved by **pure-bv SymCC-STR**?

RQ2: How do **hybrid**, **pure-str**, and **pure-bv** SymCC-STR compare on a single run?

Results (RQ2: pure-bv, pure-str, hybrid comparison)

Experiment: ran each benchmark with a single input

	pure-bv		pure-str		hybrid	
	SAT	Runtime (s)	SAT	Runtime (s)	SAT	Runtime (s)
inih	701	0.7	107	TO	**760**	25634
libspectre	**419**	3.3	18	TO	346	TO
libtasn1	2687	85	83	538	**2702**	589
libyaml	**2785**	11102	62	TO	1636	TO
openjpeg	27	0.3	33	446	**37**	193
speex	69	3	16	10284	**79**	8414

Most cases: hybrid SymCC-STR finds **more SAT instances** than pure-bv at a **shorter runtime** than pure-str.

Experiments

RQ1: How does the branch coverage achieved by **hybrid SymCC-STR** compare to the branch coverage achieved by **pure-bv SymCC-STR**?

RQ2: How do **hybrid**, **pure-str**, and **pure-bv** SymCC-STR compare on a single run?

RQ3: How do our **manual rewrites** and **string solving pass** affect performance?

Results (RQ2: rewrite performance)

Experiment: ran each benchmark through pure-string SymCC-STR with a single input, with and without assertion rewrites.

	no-rewrite		rewrite	
	SAT	Runtime (s)	SAT	Runtime (s)
inih	36	TO	**107**	TO
libspectre	18	TO	18	TO
libtasn1	**86**	2628	83	**538**
libyaml	**64**	TO	62	TO
openjpeg	33	476	33	**446**
speex	**18**	**9668**	16	10284

Results (RQ3: string solving pass performance)

Experiment: ran each benchmark through pure-string SymCC-STR with a single input, with and without our custom string solving pass.

	no-pass		pass	
	UNSAT	Runtime (s)	UNSAT	Runtime
libspectre	807	TO	818	TO
libtasn1	5443	607	5446	536

Experiments

RQ1: How does the branch coverage achieved by **hybrid SymCC-STR** compare to the branch coverage achieved by **pure-bv SymCC-STR**?

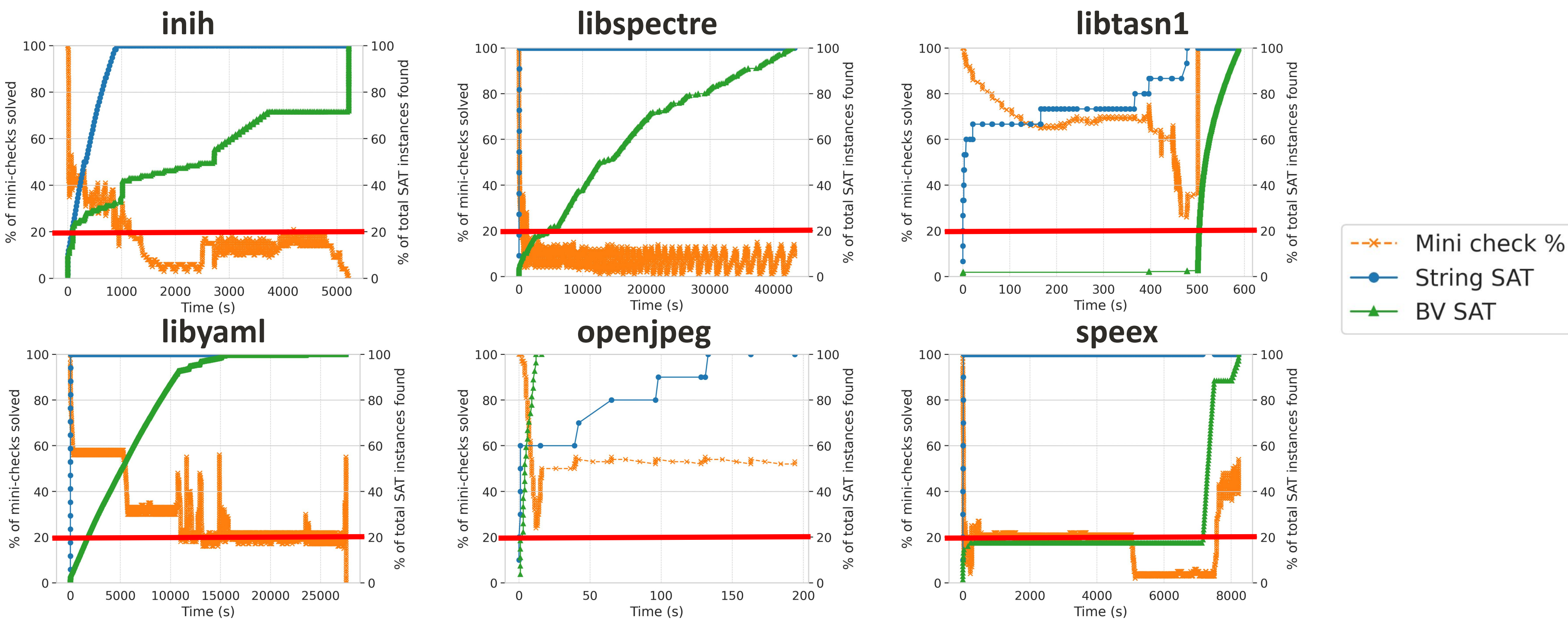
RQ2: How do **hybrid**, **pure-str**, and **pure-bv** SymCC-STR compare on a single run?

RQ3: How do our **manual rewrites** and **string solving pass** affect performance?

RQ4: At what point is the string backend no longer able to solve for new inputs (i.e., what is the **optimal point at which to disable the string backend**)?

Results (RQ4: string backend disabling)

Experiment: ran each benchmark through hybrid SymCC-STR with a single input, recording percentage of mini-checks solved and SAT instances found by the BV and string solvers over time.



Results (RQ4: string backend disabling)

Experiment: ran each benchmark through hybrid SymCC-STR with a single input, disabling the solver after less than 20% of the mini-checks are solvable.

	Baseline		20%	
	SAT	RT (s)	SAT	RT (s)
inih	760	25634	760	**983**
libspectre	346	TO	**430**	**372**
libtasn1	2702	589	2702	587
libyaml	1636	TO	**2802**	**20744**
openjpeg	37	193	37	192
speex	79	8414	79	**31**

Average speedup: 81%

What Do We Need From String Solvers?

- Better rewrites and simplifications of assertions
  - Working on a new string solving benchmark set
- Faster incremental string solving
- Better handling of conversions between bitvectors and strings
- String inequality decision procedure

Conclusion & Future Work

SymCC-STR: concolic execution engine for string-manipulating C programs that represents data as bitvectors and strings

- Scales to real world software
- Achieves better branch coverage than pure-bitvector SymCC-STR

Ongoing and future work:

- Identifying more opportunities for rewrites
- New string benchmark set
- Static code analysis to optimize for which solver to query

Thank You

Results (RQ2: pure-bv, pure-str, hybrid comparison)

Experiment: ran each benchmark with a single input

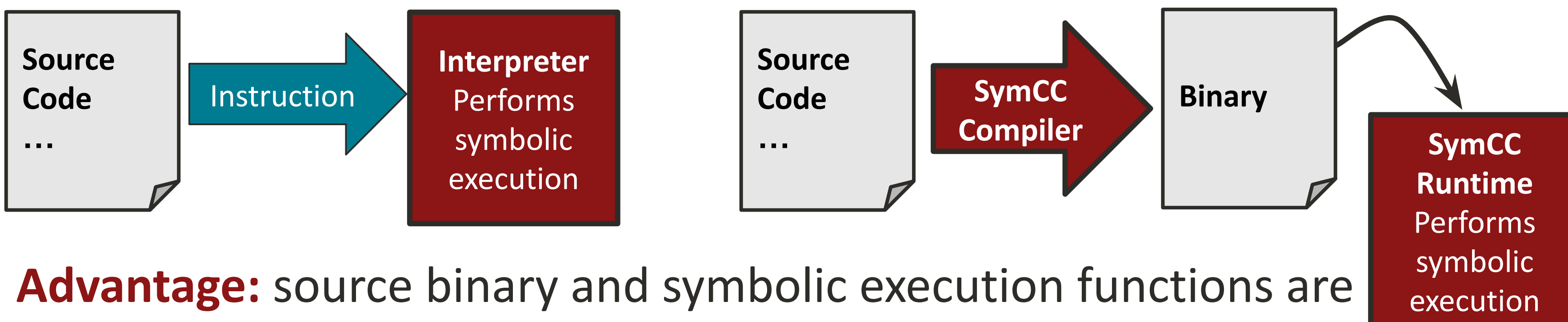
	pure-bv		pure-str		hybrid	
	SAT	Runtime (s)	SAT	Runtime (s)	SAT	Runtime (s)
inih	701	0.7	105	1806	**760**	25634
libspectre	**419**	3.3	18	TO	346	TO
libtasn1	2687	85	83	538	**2702**	589
libyaml	**2785**	11102	62	TO	1636	TO
openjpeg	27	0.3	33	446	**37**	193
speex	69	3	16	10284	**79**	8414

Most cases: hybrid SymCC-STR finds **more SAT instances** than pure-bv at a **shorter runtime** than pure-str.

Implementation: SymCC Design

SymCC achieves **better performance than other engines** by compiling the functions that perform symbolic execution directly into the binary.

Standard engines **SymCC**



Advantage: source binary and symbolic execution functions are performed natively instead of via an interpreter.

Implementation: Concolic Execution

SymCC achieves **better performance than other engines** by compiling the functions that perform symbolic execution directly into the binary.

Code Snippet: if (x + 1 > 2) {...}

```
add    %r12,1
```

SymCC Compiler

```
mov    %r14,%rdi
mov    0x20(%rsp),%rsi
call   401180 <_sym_build_add@plt>
mov    %rax,-0xc8(%rbp)
add    %r12,1
```

Symbolic expression of x

Symbolic expression of 1

SymCC runtime library performs symbolic addition of x and 1

Advantage: functions that create and solve symbolic assertions can be performed natively instead of via an interpreter.

Implementation: Concolic Execution

SymCC components: LLVM pass and runtime library

Code Snippet: if (x + 1 > 2) {...}

```
add    %r12,1
cmp    $0x2,%r12
setg   %al
test   $0x1,%al
jne    40171a <main+0x48a>
```

SymCC Compiler

Symbolic expression of x

Symbolic expression of 1

Symbolic addition of x+1

```
mov    %r14,%rdi
mov    0x20(%rsp),%rsi
call   401180 <_sym_build_add@plt>
mov    %rax,-0xc8(%rbp)
add    %r12,1 # Concretely add one to x
mov    %r12,-0xcc(%rbp)
mov    $0x2,%rdi
call   401130 <_sym_build_integer@plt>
mov    -0xc8(%rbp),%rdi
mov    %rax,%rsi
call   4010f0 <_sym_build_signed_greater@plt># Build bra
mov    %rax,-0xe8(%rbp)
mov    -0xcc(%rbp),%eax
cmp    $0x2,%eax # Evaluate concrete branch condition
setg   %al
mov    %al,-0xd9(%rbp)
mov    -0xe8(%rbp),%rdi
mov    %al,%esi
mov    $0x1bfe9f0,%edx
call   4010b0 <_sym_push_path_constraint@plt># Generate
mov    -0xd9(%rbp),%al
test   $0x1,%al
jne    40171a <main+0x48a># Take concrete branch
```

Implementation: SymCC-STR

SymCC-STR represents symbolic data as **both bitvectors and strings**.

Input: “input”

```
char buffer[11];
memset(buffer,0,11);
read(STDIN_FILENO, buffer, sizeof(buffer)
if (buffer[0] == ‘g’) {
    // C
}
if (str                    {
    // C
}
```

libc string functions are concretized by bitvector-SymCC

Advantage: performance of bitvector solver with expressivity of string solver.

(declare-fun buffer-bv () (_ BitVec 40))

(assert (not (= buffer-bv #x67)))
(assert (= 1 0))

UNSAT

(declare-fun buffer-str () String)

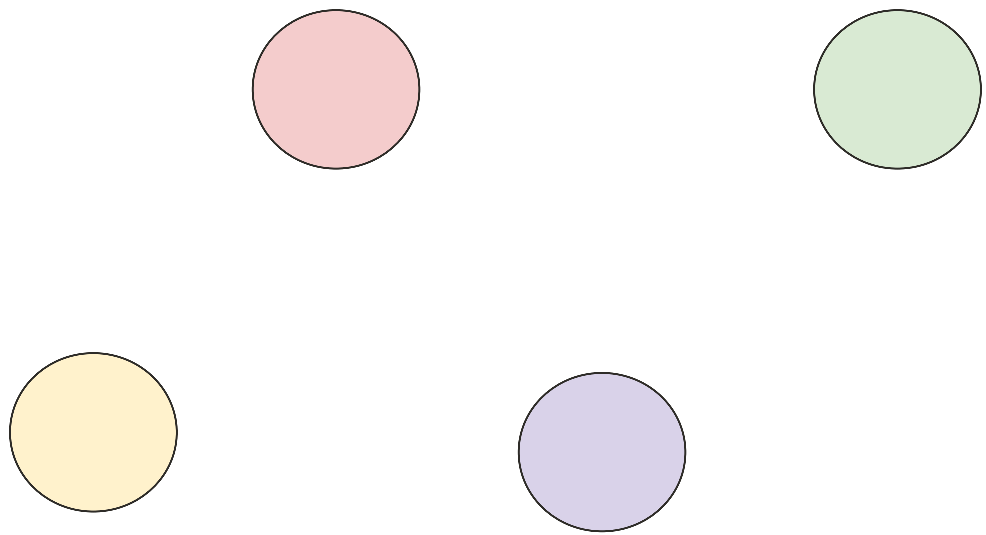
(assert (not (= (str.substr buffer-str 0 1) “g”)))
(assert (= buffer-str “hello”))

SAT: “hello”

Details – Implementation (string solving pass)

1: `substr(S,0,9)` ≪ `“Document”`

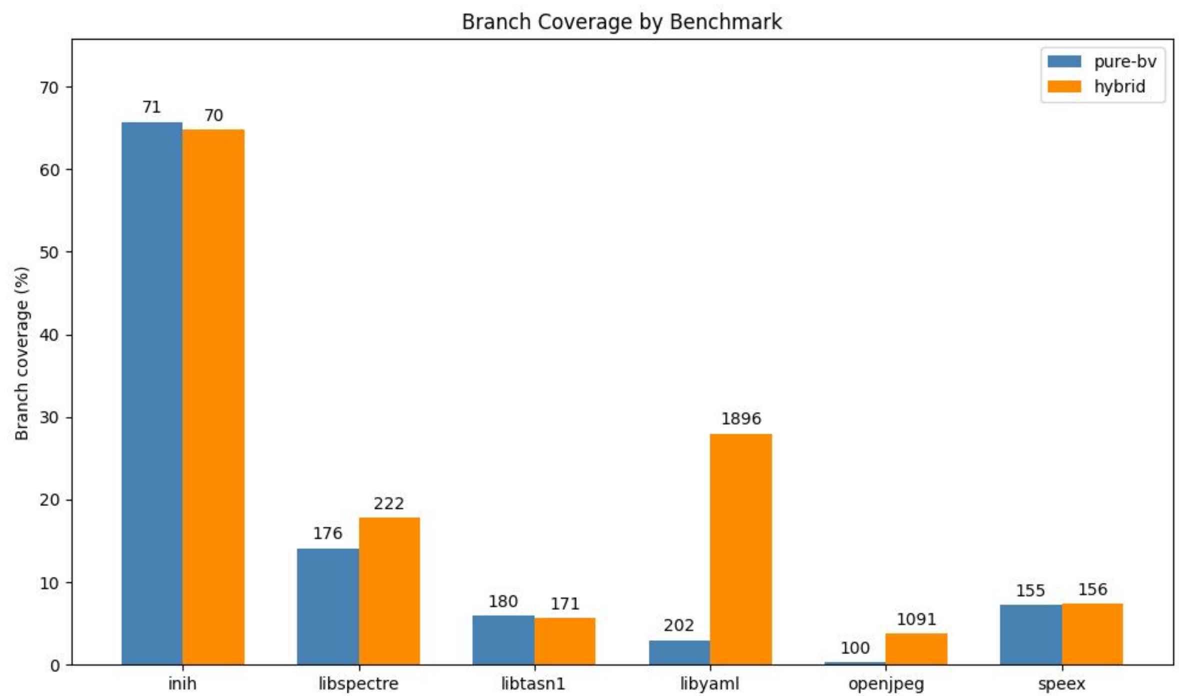
2: `“Feature”` ≪ `substr(S,0,8)`



Results (RQ1: branch coverage)

Experimental Setup

- Feed an initial input into each engine and repeatedly rerun with the new inputs they generate
- Backend solver: cvc5
- Total timeout: 12 hours
- Single-run timeout: 5 minutes
- Disable string solver when only 25% of the mini-checks are able to be solved.



Results (RQ2: rewrite performance)

Experiment: ran each benchmark through pure-string SymCC-STR with a single input, with and without assertion rewrites.

	no-rewrite			rewrite		
	SAT	UNSAT	Runtime (s)	SAT	UNSAT	Runtime (s)
inih	36	925	TO	**105**	373	**1806**
libspectre	18	814	TO	18	818	TO
libtasn1	86	8818	2628	83	5444	**538**
libyaml	64	718	TO	62	717	TO
openjpeg	33	10	476	33	9	**446**
speex	18	80	**9668**	16	89	10284

Results (RQ4: string backend disabling)

Experiment: ran each benchmark through hybrid SymCC-STR with a single input, disabling the solver after less than different percentages of the mini-checks are solvable.

	1%		5%		10%		25%		50%	
	SAT	RT (s)	SAT	RT (s)	SAT	RT (s)	SAT	RT (s)	SAT	RT (s)
inih	712	1441	712	1015	712	893	712	587	710	72
libspectre	430	12129	430	1889	430	135	430	128	430	125
libtasn1	2812	1151	2812	1150	2812	1150	2812	1150	2812	482
libyaml	2802	35542	2802	35573	2802	35463	2802	20756	2802	16241
openjpeg	34	191	34	192	24	191	30	12	30	9
speex	82	7634	82	7601	82	3920	82	130	75	11