Data-driven Recurrent Set Learning for Non-termination Analysis

Zhilei Han, Fei He | School of Software, Tsinghua University

May 18, 2023

Non-termination Bugs

A program is **non-terminating** if there exist some inputs that cause the program to execute indefinitely.

- Non-termination is usually considered a bug that could lead to severe consequence.
- ~800 reported DoS vulnerabilities result from infinite loops.
- A recent empirical study found **445** non-termination bugs from **199** real-world OSS projects.

[X. Shi, X. Xie, Y. Li, Y. Zhang, S. Chen, and X. Li · Large-scale analysis of non-termination bugs in real-world OSS projects · ESEC/FSE 2022]

Non-termination Bugs

A program is **non-terminating** if there exist some inputs that cause the program to execute indefinitely.

- Non-termination is usually considered a bug that could lead to severe consequence.
- ~800 reported DoS vulnerabilities result from infinite loops.
- A recent empirical study found **445** non-termination bugs from **199** real-world OSS projects.

[X. Shi, X. Xie, Y. Li, Y. Zhang, S. Chen, and X. Li · Large-scale analysis of non-termination bugs in real-world OSS projects · ESEC/FSE 2022]

Non-termination Bugs

A program is **non-terminating** if there exist some inputs that cause the program to execute indefinitely.

- Non-termination is usually considered a bug that could lead to severe consequence.
- ~800 reported DoS vulnerabilities result from infinite loops.
- A recent empirical study found **445** non-termination bugs from **199** real-world OSS projects.

[X. Shi, X. Xie, Y. Li, Y. Zhang, S. Chen, and X. Li · Large-scale analysis of non-termination bugs in real-world OSS projects · ESEC/FSE 2022]

Non-termination Bugs

A program is **non-terminating** if there exist some inputs that cause the program to execute indefinitely.

- Non-termination is usually considered a bug that could lead to severe consequence.
- ~800 reported DoS vulnerabilities result from infinite loops.
- A recent empirical study found **445** non-termination bugs from **199** real-world OSS projects.

[X. Shi, X. Xie, Y. Li, Y. Zhang, S. Chen, and X. Li · Large-scale analysis of non-termination bugs in real-world OSS projects · ESEC/FSE 2022]

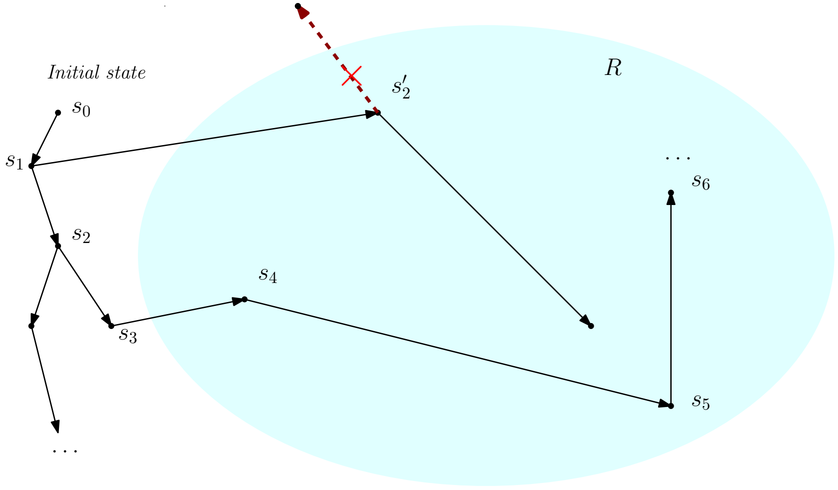
State-of-the-art: Recurrent Set

The standard non-termination analysis method to date is to synthesize a **recurrent set**.

A (closed) recurrent set is a set of states R such that

1. R is reachable from an initial state,
2. If a state s in R is reached, all successors of s remain in R,
3. Any state in R has at least a successor

The existence of R proves non-termination of a program.



[H.-Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. O'Hearn· Proving Nontermination via Safety · TACAS 2014]

State-of-the-art: White-box Synthesis

Existing tools utilize white-box methods to synthesize a recurrent set by templatization and SMT solving.

```c
void foo(int i) {
    while (i >= 0) {
        i++;
    }
}
```

$$\forall i, R(i) \longrightarrow i \geqslant 0 \wedge R(i+1)$$

Templatization $\downarrow$

$$\forall i, a \cdot i \leqslant b \longrightarrow i \geqslant 0 \wedge a \cdot (i+1) \leqslant b$$

Farkas' Lemma $\downarrow$

$$\exists \delta_1, \delta_2, \begin{pmatrix} \delta_1 \\ \delta_2 \end{pmatrix} a = \begin{pmatrix} -1 \\ a \end{pmatrix} \wedge \begin{pmatrix} \delta_1 \\ \delta_2 \end{pmatrix} b \leqslant \begin{pmatrix} 0 \\ b-a \end{pmatrix}$$

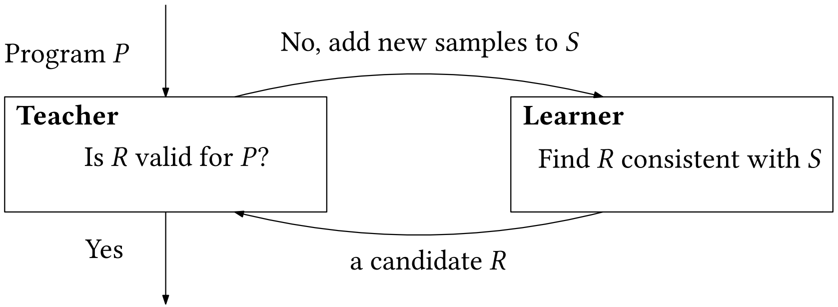
SMT Solver $\downarrow$

$a = -1, b = 0$ a.k.a $R(i) = i \geqslant 0$

However, this method does not work well on loops with non-linear assignments or complicated control-flow.

Black-box learning has been successfully applied to invariant generation and can handle complex programs.

Basic Idea (CEGIS)



Pros

1. Agnostic of the concrete program
2. Able to prove aperiodic non-termination

Cons

Termination is a **liveness** property. It seems impossible to obtain a non-terminating sample…

Challenges on Sample Generation

Recall the definition of recurrent set…

1. R is reachable from an initial state $\longrightarrow$?
2. If a state s in R is reached, all successors of s remain in R $\longrightarrow$ implicative sample
3. Any state in R has at least a successor $\longrightarrow$ negative sample

```
void foo(int i) {
      while (i >= 0) {
         i++;
      }
}
```

	Candidate	Valid?	Sample Set
Initial	-	-	$\emptyset$
Step 1	`True`	No	$\{(-1, \text{neg})\}$
Step 2	`False`	No	$\{(-1, \text{neg}), ???\}$
Step 3			

We cannot obtain a positve sample from an existential property!

Sample Speculation

A Walkthrough of Our Method: Overview

We analyze every loop L in the program and try to synthesize a recurrent set R.

```
while (k != 0) {
      j = -2 * (k - 1) * k
      k = j * k;
      j = 0;
}
```

$\longrightarrow$ $(\texttt{true}, k \neq 0, k' = -2 \cdot (k-1) \cdot k \cdot k \wedge j' = 0)$

We proceed by the standard method, using a specialized *decision tree* learner.

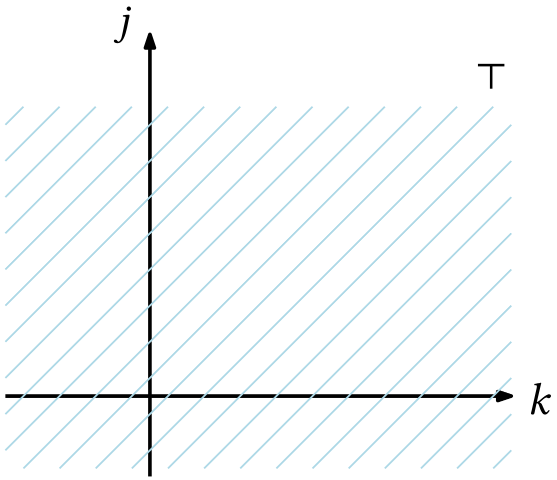
Recall that R should satisfy:

1. $\exists j, k. R(j,k)$
2. $\forall j, k. R(j,k) \longrightarrow k \neq 0$
3. $\forall j, k, j', k'. R(j,k) \wedge k' = -2 \cdot (k-1) \cdot k \cdot k \wedge j' = 0 \longrightarrow R(j', k')$

[Kincaid, Z.; Reps, T.; Cyphert, J.· Algebraic Program Analysis · CAV 2021]

Analyzed loop : $(\texttt{true}, k \neq 0, k' = -2 \cdot (k-1) \cdot k \cdot k \wedge j' = 0)$

Step 1



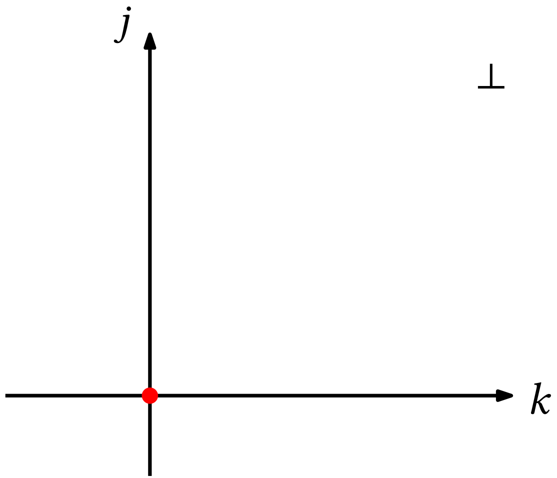
$\forall k, j. \top \longrightarrow k \neq 0$ is invalid!

	Candidate	Valid?	Sample Set
Initial	-	-	$\emptyset$
Step 1	True	No	$\{((0,0), \text{neg})\}$

$(k = 0, j = 0)$ is a negative sample

Analysized loop : $(\texttt{true}, k \neq 0, k' = -2 \cdot (k-1) \cdot k \cdot k \wedge j' = 0)$

Step 2



$\exists k, j. \top \longrightarrow \perp$ is invalid

	Candidate	Valid?	Sample Set
Initial	-	-	$\emptyset$
Step 1	`True`	No	$(0,0)^-$
Step 2	`False`	No	$(0,0)^-, (-1,0)^+_?$

We speculate a state $(-1,0)$ from the reachable set as positive sample.

Sample Speculation

To reduce overhead, the positive sample is selected from the states that satisfy the following:

- it is reachable from an initial state,
- it does not belong to the set of already known terminating states, and
- it does not terminate within a fixed number of steps.

Sample Speculation

To reduce overhead, the positive sample is selected from the states that satisfy the following:

- it is reachable from an initial state,
- it does not belong to the set of already known terminating states, and
- it does not terminate within a fixed number of steps.

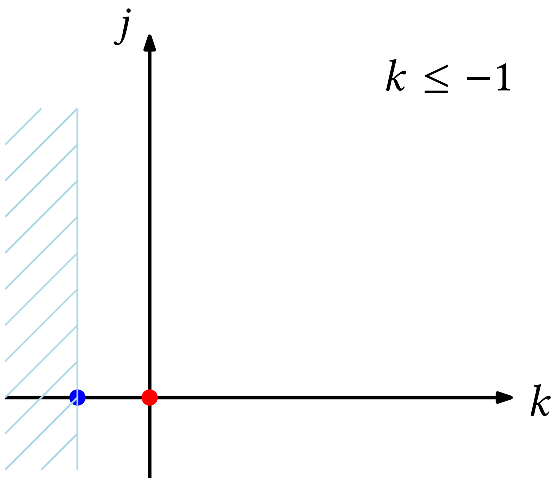
Sample Speculation

To reduce overhead, the positive sample is selected from the states that satisfy the following:

- it is reachable from an initial state,
- it does not belong to the set of already known terminating states, and
- it does not terminate within a fixed number of steps.

Analyzed loop : $(\texttt{true}, k \neq 0, k' = -2 \cdot (k-1) \cdot k \cdot k \wedge j' = 0)$

Step 3



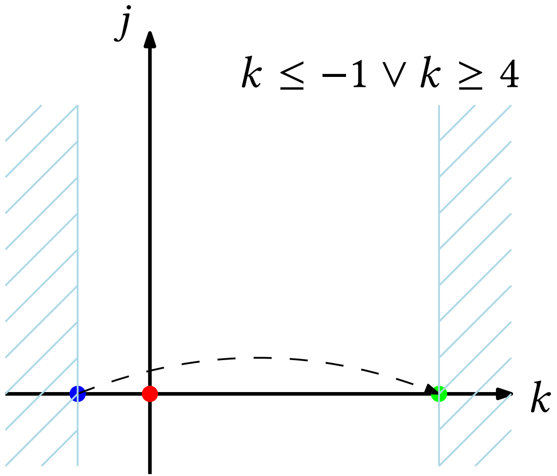
$\forall j, k, j', k'. k \leqslant -1 \wedge k' = -2 \cdot (k-1) \cdot k \cdot k \wedge j' = 0 \longrightarrow k' \leqslant -1$
is invalid

	Candidate	Valid?	Sample Set
Initial	-	-	$\varnothing$
Step 1	True	No	$(0,0)^-$
Step 2	False	No	$(0,0)^-, (-1,0)^+_?$
Step 3	$k \leqslant -1$	No	$(0,0)^-, (-1,0)^+_?, (-1,0) \longrightarrow (4,0)$

$(-1,0) \longrightarrow (4,0)$ is an implicative sample.

Analysized loop : $(\texttt{true}, k \neq 0, k' = -2 \cdot (k-1) \cdot k \cdot k \wedge j' = 0)$

Step 4



$k \leqslant -1 \vee k \geqslant 4$ is a valid recurrent set.

	Candidate	Valid?	Sample Set
Initial	-	-	$\emptyset$
Step 1	`True`	No	$(0,0)^-$
Step 2	`False`	No	$(0,0)^-, (-1,0)^+_?$
Step 3	$k \leqslant -1$	No	$(0,0)^-, (-1,0)^+_?, (-1,0) \longrightarrow (4,0)$
Step 4	$k \leqslant -1 \vee k \geqslant 4$	Yes	

What if…

The speculated positive sample $(-1, 0)$ happens to be non-terminating.

Suppose we choose $(1, 0)$ instead. After several iterations the sample set becomes

$$(0,0)^-, (1,0)^+, (1,0) \longrightarrow (0,0)$$

which is inconsistent. The learner cannot return any candidate!

	Candidate	Valid?	Sample Set
Initial	-	-	$\emptyset$
Step 1	`True`	No	$(0,0)^-$
Step 2	`False`	No	$(0,0)^-, (1,0)^+_?$
Step 3	$k > 0$	No	$(0,0)^-, (1,0)^+, (1,0) \longrightarrow (0,0)$
Step 4	$\times$		

Backtracking

When the sample set becomes inconsistent, we backtrack by labeling the positive sample as negative, select a fresh positive sample and proceed.

Backtracking

When the sample set becomes inconsistent, we backtrack by labeling the positive sample as negative, select a fresh positive sample and proceed.

Is the learning algorithm guaranteed to converge?

Backtracking

When the sample set becomes inconsistent, we backtrack by labeling the positive sample as negative, select a fresh positive sample and proceed.

Is the learning algorithm guaranteed to converge?

Yes, if we select the samples in the right order!

We make sure to select a positive sample s with respect to a bound c such that.

1. s is bounded by c. ($\| s \| \leqslant c$)
2. c is incremented only when all possible states bounded by c has been sampled.

Theorem 1

Suppose the decision tree learner has a fixed set of attributes. For any loop L, if L admits a recurrent set expressible as the Boolean combination of these attributes, then the black-box learning algorithm is guaranteed to converge if the positive sample is selected with respect to a bound c.

Experimental Results

In comparison with state-of-the-art non-termination analysis tools, our prototype implementation solves more cases of TermComp benchmarks and achieves up to 5x increase in performance.

	Our Tool	RevTerm	Ultimate	VeryMax
Solved Cases (111 total)	109	101	98	103
Speedup	-	1.9x	5x	4.4x

Our algorithm is also the only one that actually works on non-linear programs.

[Chatterjee, K.; Goharshady, E. K.; Novotný, P.; Žikelić, Đ· Proving Non-Termination by Program Reversal · PLDI 2021]

[Leike, J.; Heizmann, M· Geometric Nontermination Arguments · TACAS 2018]

[Borralleras, C et al.· Proving Termination Through Conditional Termination · TACAS 2017]

Thanks!