Verification by Gambling on Program Slices

Murad Akhundov[1], Federico Mora[2], Nick Feng[1], Vincent Hui[1], Marsha Chechik[1]

ATVA 2021

1. University of Toronto, Canada
2. University of California, Berkeley, USA

Problem

Problem: Automated verification is expensive

It's desirable to verify properties in large programs.

Many state-of-the-art verifiers struggle with complex code.

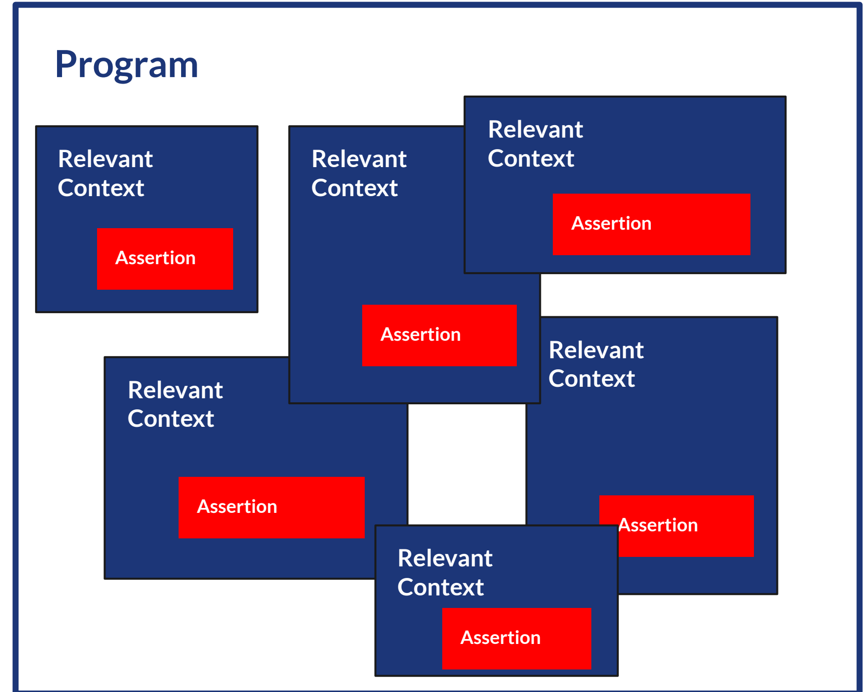
Problem: Irrelevant context increases cost

Can we quickly verify desired properties in an entire program by considering it in smaller pieces?

Problem: Finding relevant context is hard

1. Manually partitioning a program into parts small enough for a verifier to handle is time consuming

2. It's not trivial to determine how much context is required to verify a property, leading to potential false-positive violations



Reducing irrelevant context

Regular slicing techniques can help reduce irrelevant context, but are often either:

- **Too cautious**, and leave too much irrelevant complexity for the solver
- **Too aggressive**, sacrificing soundness [1]

Our tool Qicc, takes a gamble by trying to verify assertions with **less context, separately, and concurrently**.

1. Cook, B., Döbel, B., Kroening, D., Manthey, N., Pohlack, M., Polgreen, E., Tautschnig, M., Wieczorkiewicz, P.: Using Model Checking Tools to Triage the Severity of Security Bugs in the Xen Hypervisor. In: Proc. of FMCAD'20

Motivating Example

Motivating Example: context and bound checks

```
1 ... REST OF THE PROGRAM
2
3 for (i = 0; i < F25519_SIZE; i++) {
4     int j;
5     c >>= 8;
6     for (j = 0; j <= i; j++){
7         c += ((word32)a[j]) * ((word32)b[i - j]);
8     }
9
10    ... REST OF THE LOOP
11
12 }
13
14 ... REST OF THE PROGRAM
```

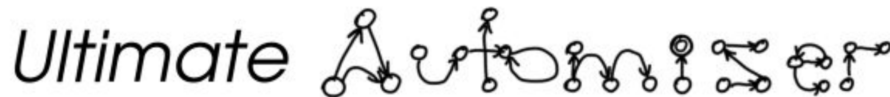
Taken from curve25519 implementation in busybox. ~450 lines of code.

To verify array upper bounds highlighted in blue, code highlighted in pink can be ignored, including the rest of the function and the program.

State-of-the-art program checkers struggle when presented with the entire program.

Array bounds check in curve25519

Both CBMC (a bounded model checker for C) and Ultimate Automizer (an automate-based verifier) are unable to terminate in two hours when presented with curve25519 function.



Why these checkers don't work?

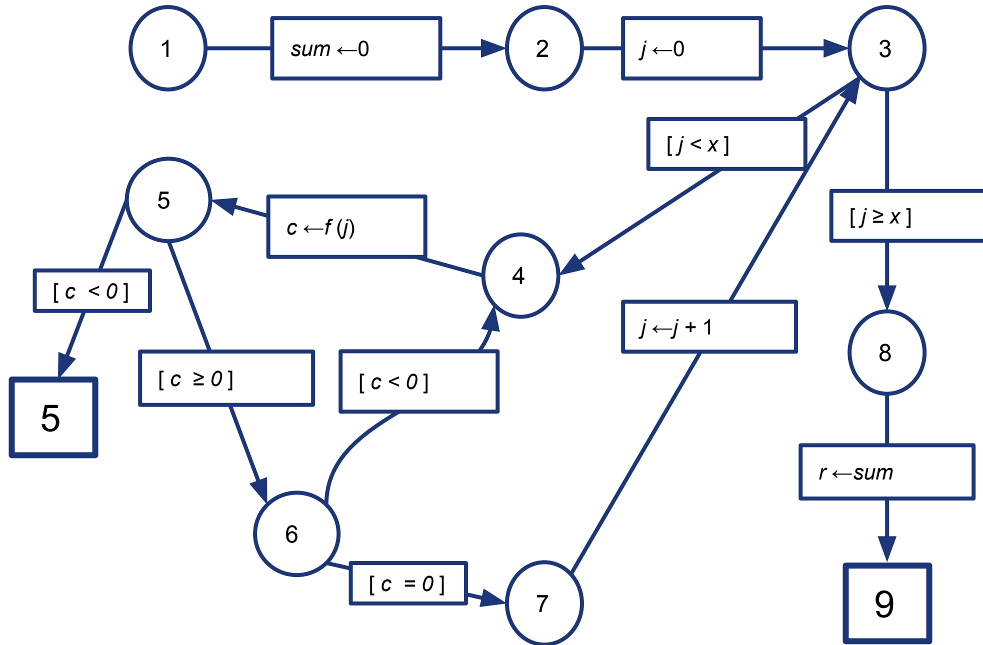
- The checkers produce large, complex models

- Irrelevant context makes complexity much worse

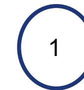
Our tool, Qicc, enables two existing checkers to solve this motivating example in **under a minute**, when they would be otherwise **unable to terminate** in two hours.

Background - Program Safety and Regions

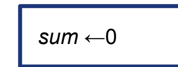
Background: Control Flow of a Program



Legend



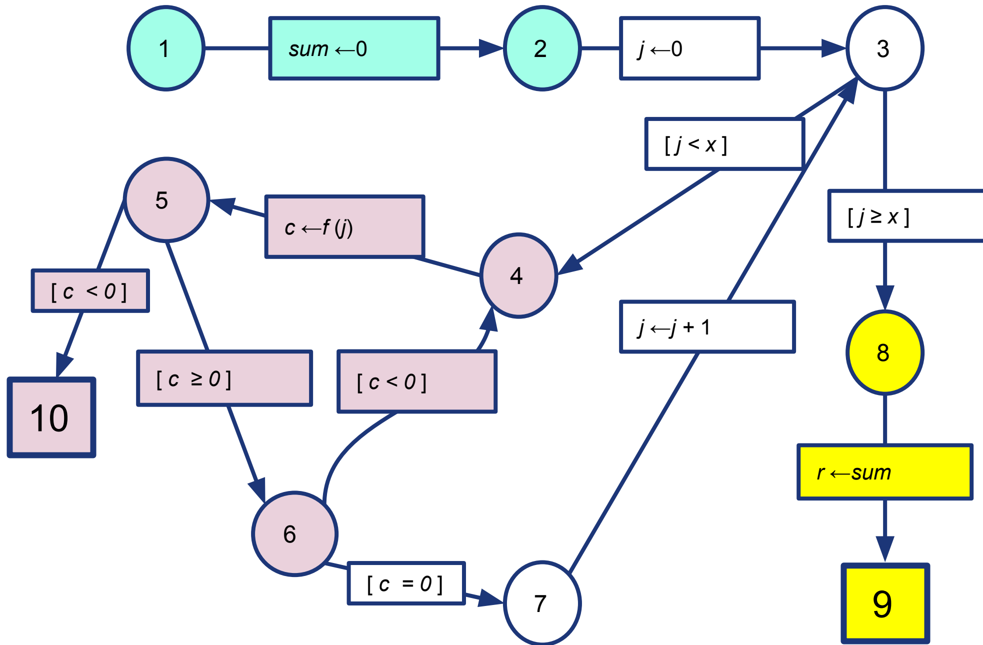
location



operations

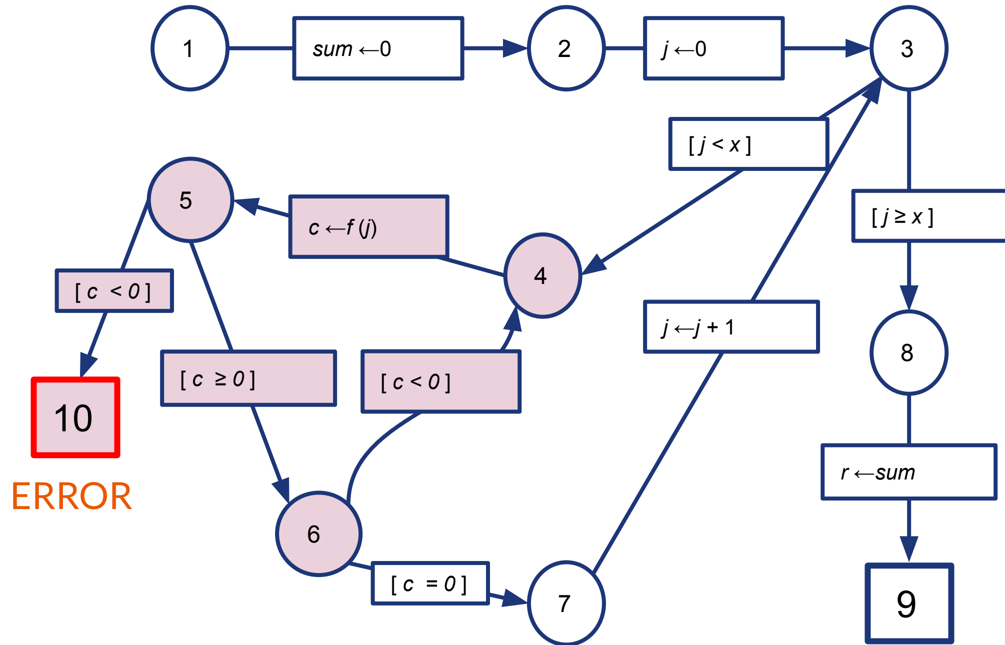
terminal location

Control flow region of a program



Regions contiguous are portions of the control-flow graph of a program with a single entry location and a single exit location

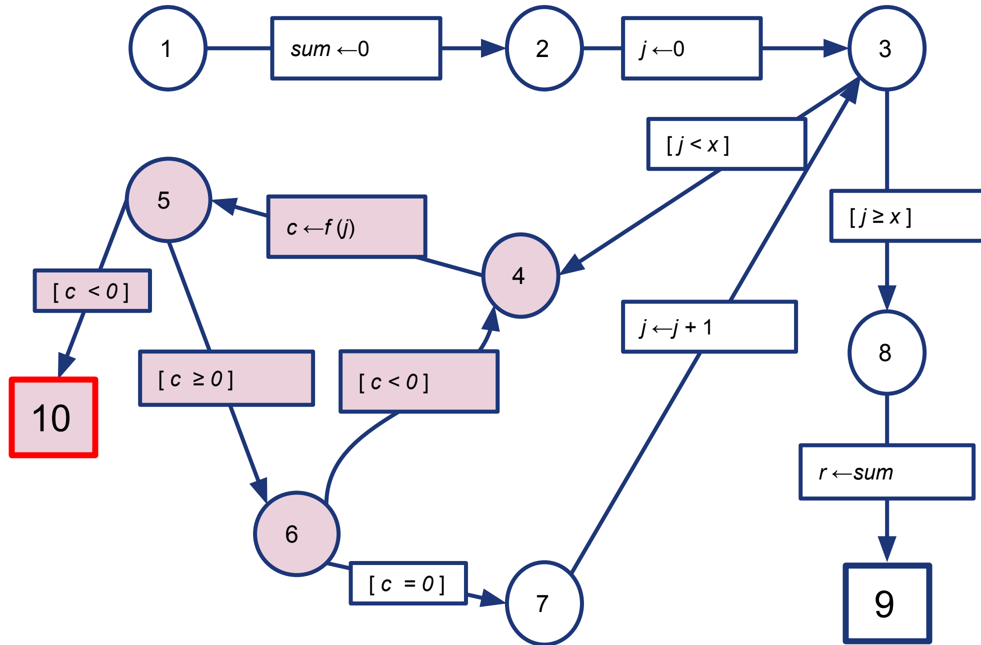
Program safety



If no error location is reachable, then the program is safe.

If no error location is reachable in the region then the region is safe.

Error reachability

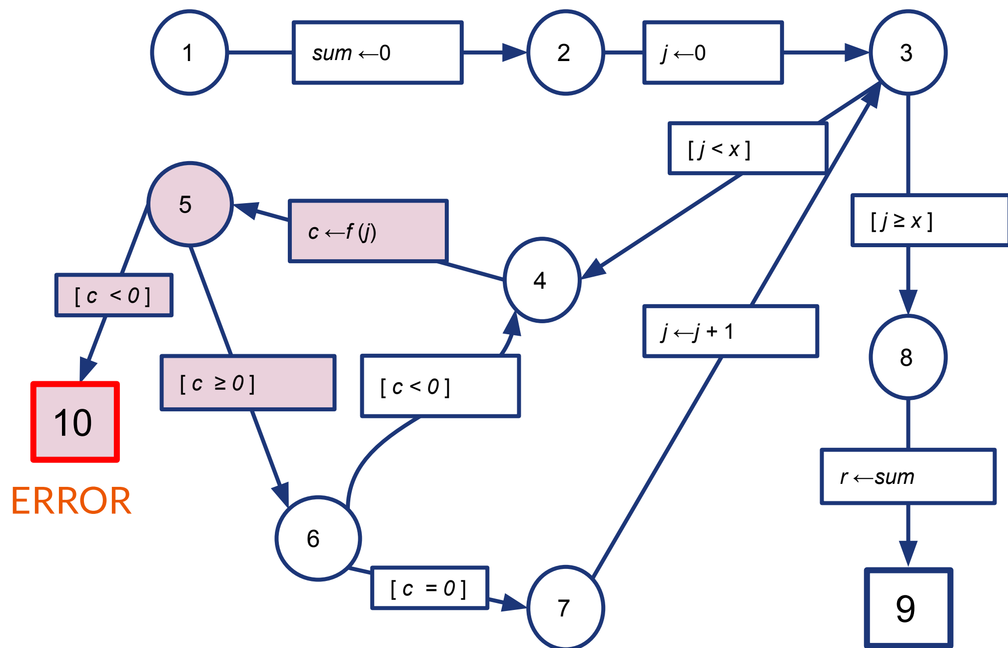


If an error location is unreachable in a region that contains it, then the error location is unreachable in the whole program.

Theorem: If every assertion is contained in a safe region, the program is safe.

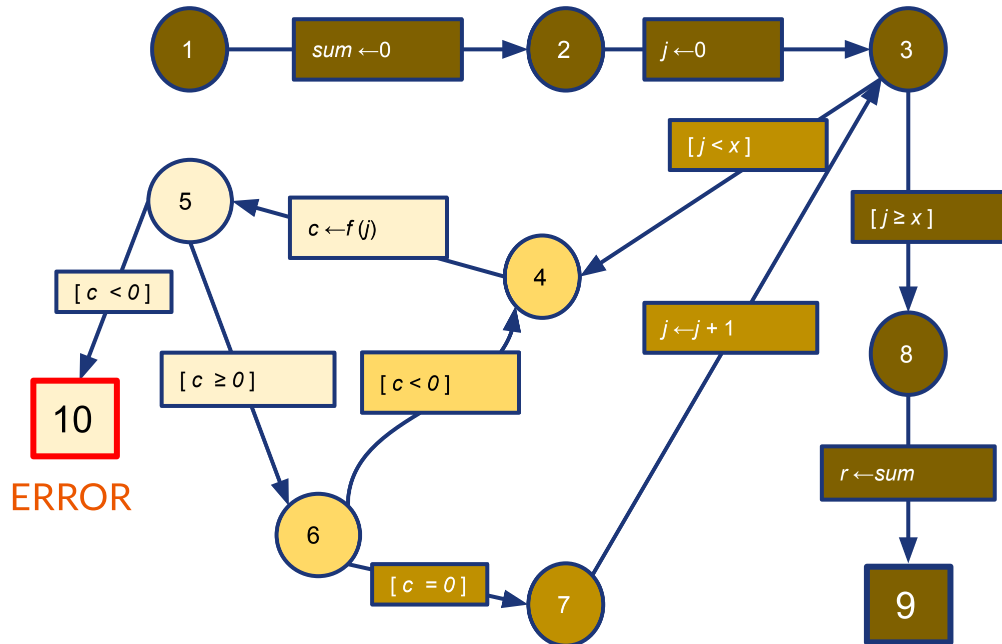
Our Approach

Our Approach: minimizing context



The goal of our approach is to expedite verification by **minimizing irrelevant context**.

Gambling on small regions



Our technique "takes a gamble" by trying to verify with as **little context** as possible, and **gradually adds context** back until it's sufficient.

Minimum Context

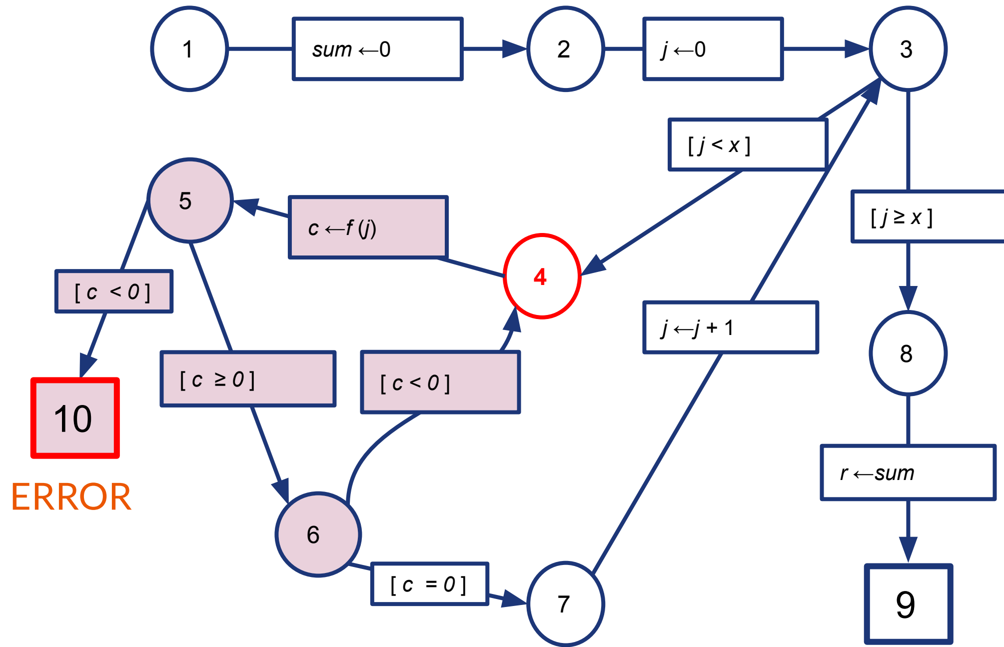
Maximum Context

Qicc hits and misses

- We say that Qicc **hits** when an identified region can be verified successfully in isolation. **(The gamble pays off)**

- We say that Qicc **misses** when more context is needed to prove the assertion in a region. Qicc will need to expand context and try again. **(The gamble was lost)**

Choosing regions to gamble on



Since it would be too expensive to attempt every possible region, our tool, Qicc, prioritizes **bodies of cyclic regions**, as they are often much quicker to verify than the parent region containing the cycle.

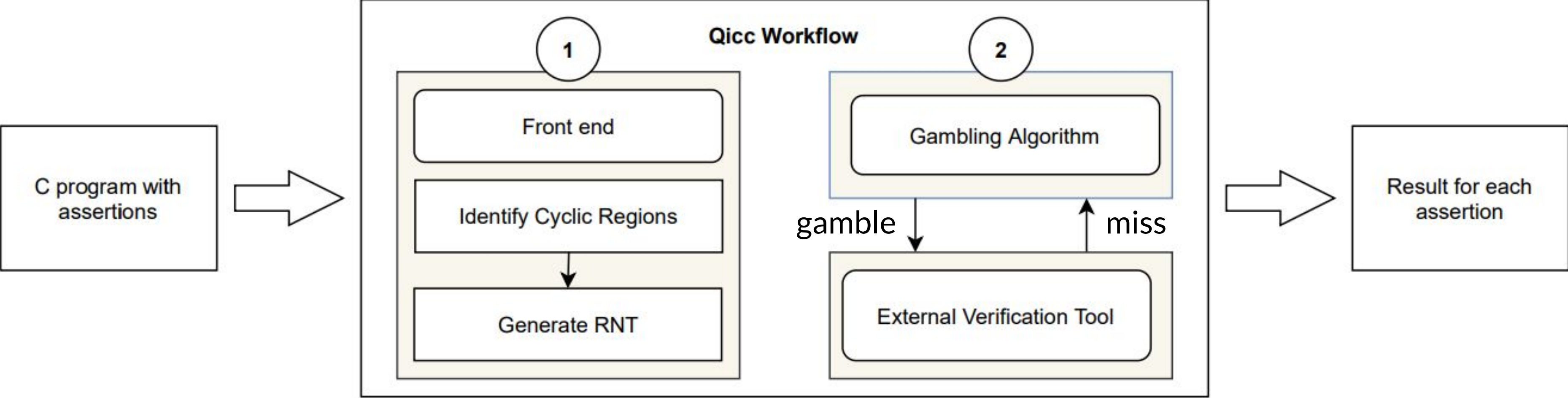
Hypothesis: misses will be cheap

When Qicc misses, it means selected region was **too small** to check the assertion.

A small region is usually **faster** to check than the entire program.

Therefore, checking a few small regions should be relatively cheap.

Qicc Workflow



Cyclic Region Identification

Algorithm 1: RNT Gen

Data: Reducible CFA, f
Result: Region Nesting Tree T

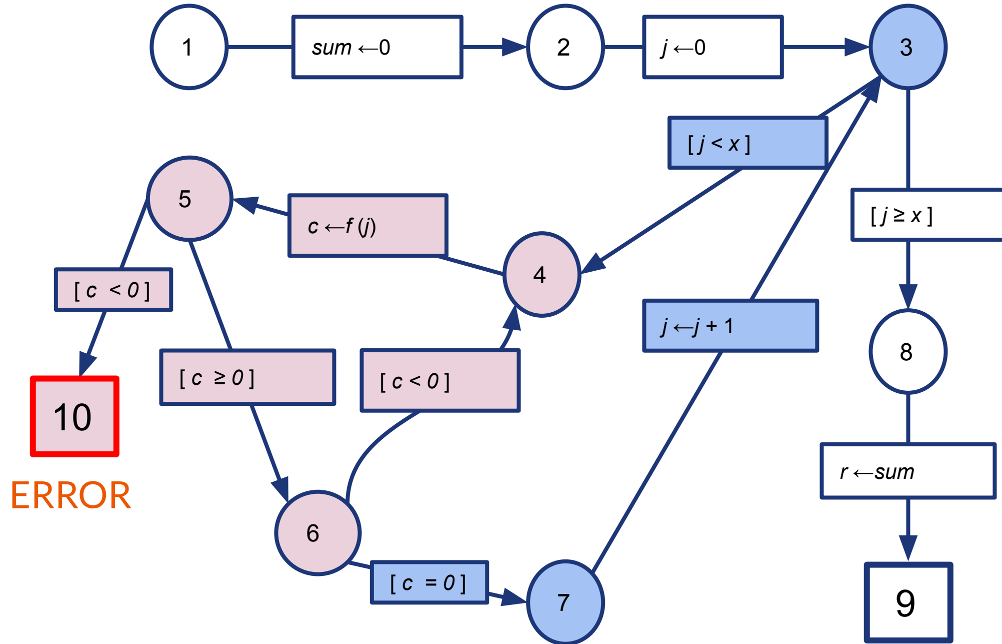
```
1  σ ← new Stack()
   /* T is an association map */
2  T.addChild(ROOT, f)
3  σ ← push(σ, f)
4  while ¬isEmpty(σ) do
5  |   γ ← pop(σ)
6  |   IRs← InnerFinder(γ, T)
7  |   σ ← pushAll(σ, IRs)
8  return T
```

Algorithm 2: InnerFinder

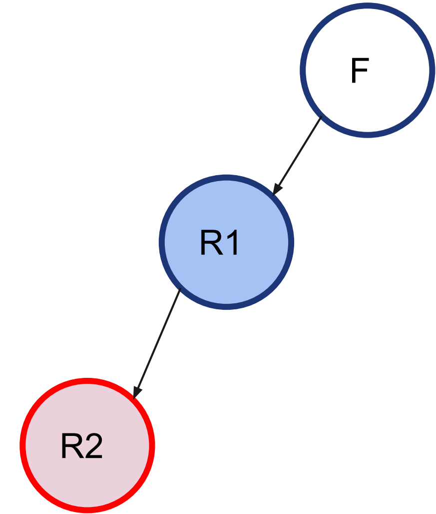
Data: Region γ
Mutate: Region Nesting Tree T
Result: Set of regions inside γ

```
1  components ← Tarjan(γ)
2  InnerRegions ← set()
3  for S ← components do
4  |   if size(S) > 1 then
5  |   |   S_body ← RegionBody(S)
6  |   |   T.addChild(γ, S_body)
7  |   |   InnerRegions.add(S_body)
8  return InnerRegions
```

Region Nesting Tree generation

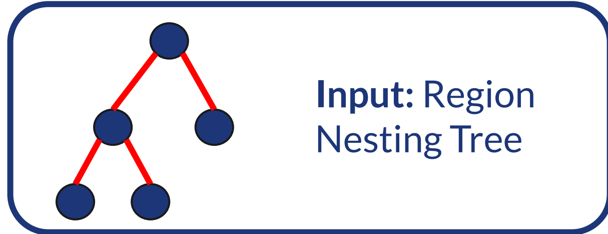


Control flow of a program with cyclic regions highlighted

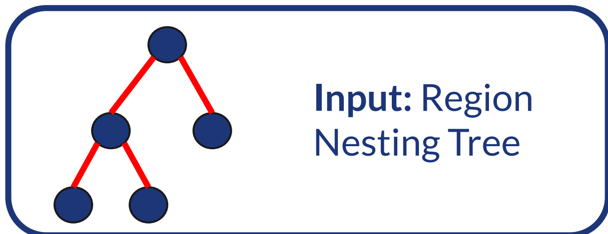


Region Nesting Tree

Gambling on regions

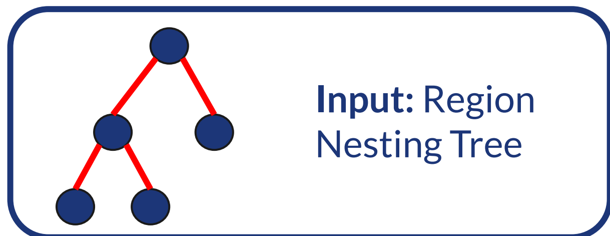


Gambling on regions

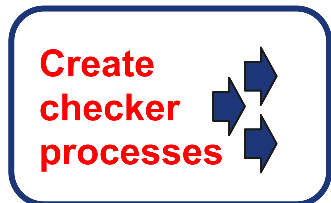


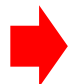
Identify deepest regions with assertions

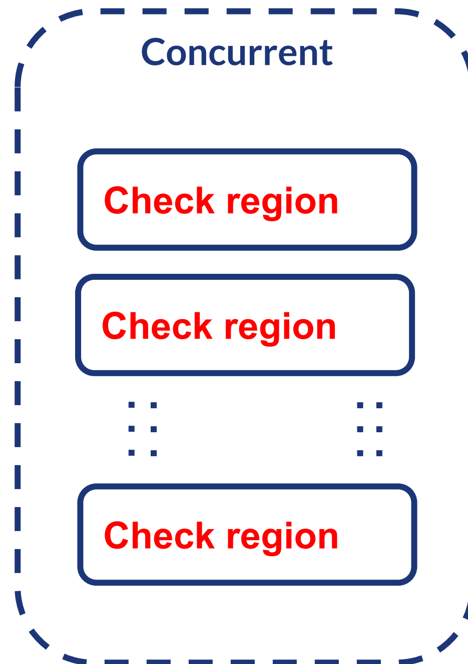
Gambling on regions



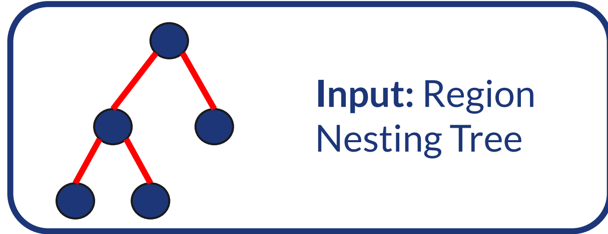
Identify deepest regions with assertions





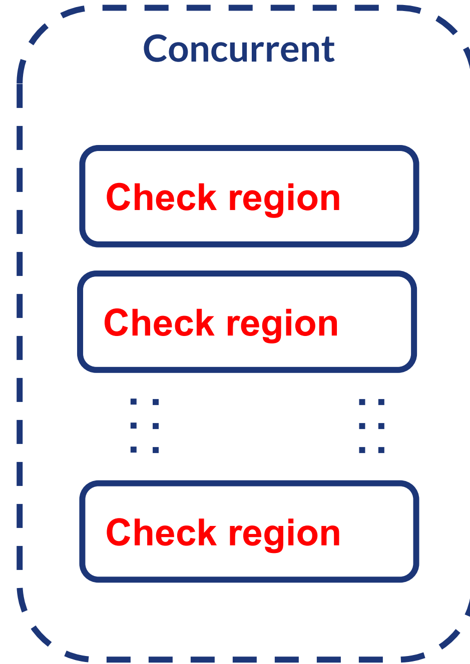


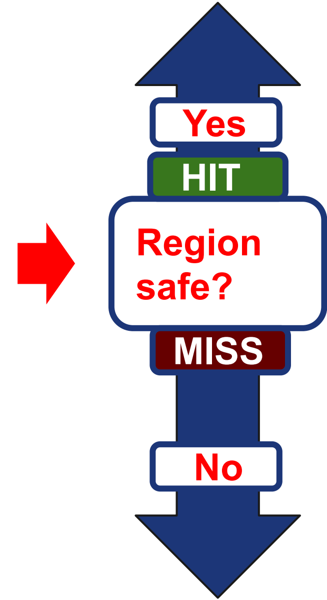
Gambling on regions



Identify deepest regions with assertions

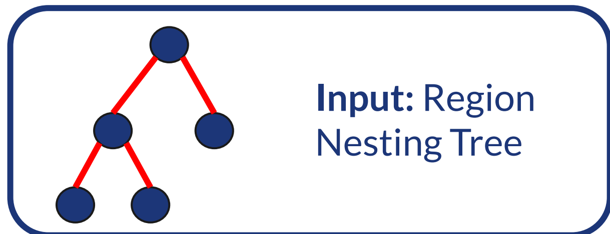
Create checker processes





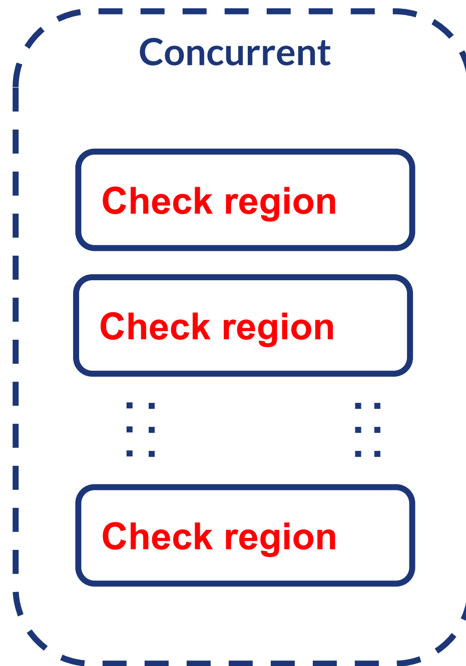
Mark Assertions in the region as verified

Gambling on regions



Identify deepest regions with assertions

Create checker processes



Mark Assertions in the region as verified

Fetch parent regions

Root region

Terminate

Gambling Algorithm Optimisations

1. Batch Verification: If two assertions are located within the same region, they will be passed to the verifier as a single batch. If a region is found to be safe, all assertions are marked as verified.

2. Concurrent Verification: Regions in the nesting tree can be verified in parallel, as long as one isn't a parent of another.

Evaluation

Implementation

Qicc Frontend (identification & tree generation): OCaml, as plugins for C Intermediate Language (CIL)

Gambling Algorithm: TypeScript, this includes RNT traversal and interfaces with verification engines

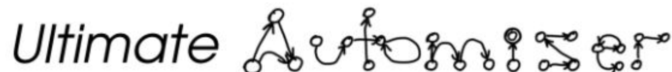
Implementation Limitations
No support for: regions with multiple entries, recursion.

Evaluation with existing verifiers

We integrated Qicc with two existing state-of-the-art program checkers.

CBMC: A bounded model checker for C. We expected Qicc to perform very well with CBMC as bounded model checkers struggle with cycles.

Ultimate Automizer (UA): An automata based model checker. Chosen for variety and being top performer in SV-COMP.



Research Questions

RQ1: What is the benefit of Qicc+CBMC when the gamble succeeds? What is the cost of Qicc+CBMC when the gamble fails?

RQ2: Does the performance benefit of Qicc extend to other verifiers? (Ultimate Automizer)

RQ3: Can Qicc scale to large, real-world, programs?

RQ1&2: Systematic analysis - Experimentation

A thorough analysis with synthetically generated programs for CBMC and Ultimate Automizer.

15 SVCOMP programs used as base benchmarks, with irrelevant context systematically introduced, simulating both hits and misses.

10 minute time out for all verification runs.

Systematic Analysis: Varying program structure

(a) Baseline/Linear

```
1 [fact]
2
3 [assert]
```

(b) Single 1

```
1 loop:
2     [fact]
3     [assert]
```

(c) Single 2

```
1 [fact]
2 loop:
3     [assert]
```

(d) Double 1

```
1 loop:
2     loop:
3         [fact]
4         [assert]
```

(e) Double 2

```
1 loop:
2     [fact]
3     loop:
4         [assert]
```

(f) Double 3

```
1 [fact]
2 loop:
3     loop:
4         [assert]
```

Systematic Analysis: Loop Bounds

We varied used 3 different preset loop bounds and varied them for each program structure: small static bound (10), large static bound (200), and arbitrary bound.

This variable was only used for CBMC, as UA models programs using automata and the runtime is not directly affected by static loop bounds.

Systematic analysis: generating programs

- 6 preset program structures, used to generate synthetic cases
  - 2 guaranteeing only Qicc misses and 2 guaranteeing only Qicc hits
  - 1 baseline with no sub regions, 1 with both a hit and a miss
- 3 loop bound presets varied in every case

```
void main(){
    int a = 0;
    for(int i = 1; i < 2; i++){
        a = i;
    }

    assert(a == 1);
}
```

Original program

```
[fact]
loop:
    loop:
        [assert]
```

Program Structure

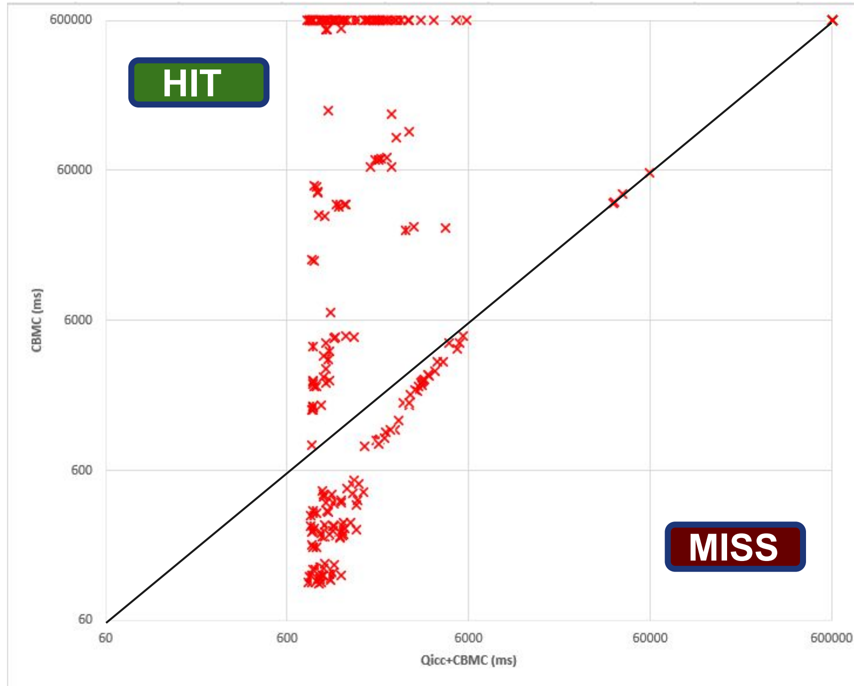
```
void main(){
    void main(){
    int a = 0;
    for(int i = 1; i < 2; i++){
        a = i;
    }

    for(int i = 1; i < LOOP_BOUND; i++){
        for(int i = 1; i < LOOP_BOUND; i++){
            assert(a == 1);
        }
    }
}
```

Generated Scenario

RQ1: Systematic Analysis Results: CBMC

Bounds/Structures	Solved instances, CBMC/Qicc+CBMC				
	Single 1	Single 2	Double 1	Double 2	Double 3
small	13/13	13/13	8/13	13/13	13/13
large	7/13	13/13	4/13	7/13	13/13
arbitrary	0/13	0/0	0/13	0/0	0/0
small/large			6/13	13/13	13/13
small/arbitrary			0/13	0/0	0/0
large/small			6/13	7/13	13/13
large/arbitrary			0/13	0/0	0/0
arbitrary/small			0/13	0/13	0/0
arbitrary/large			0/13	0/13	0/0

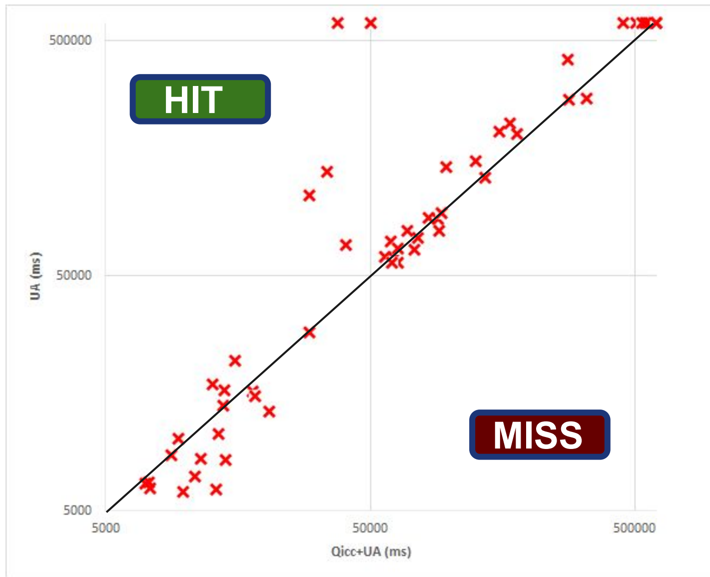
RQ1: Systematic Analysis Results: CBMC



Instances solved CBMC: 175 CBMC+Qicc: 325

- Performance gain when Qicc hits is very significant, cost of a miss is low.
- The cost of the miss is at worst proportional to the depth of the program.
- The misses are proportionally more impactful on easy cases.

RQ2: Systematic Analysis Results: UA



Instances solved

UA: 45 UA+Qicc: 53

- Cost of a miss remains low
- Qicc+UA is able to solve more instances than UA alone
- Performance gained by using Qicc not as large as with CBMC, but may be improved by using different slicing heuristic

Systematic analysis - Threats to validity

1. Irrelevant context introduced was limited to cycles, which affects CBMC a lot more directly than UA.

2. All examples were synthetically generated, but they were varied systematically.

   Synthetic examples enabled control of irrelevant context, which is usually absent in benchmarks.

RQ3 Case study - Experimentation

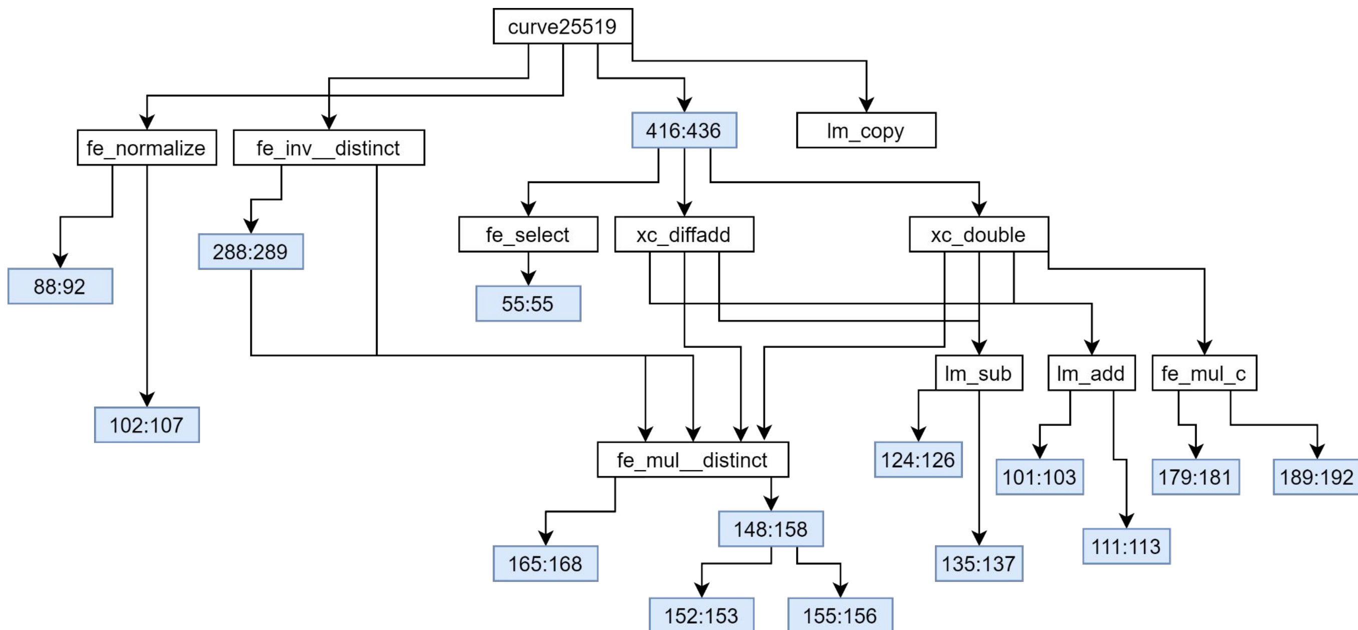
Curve25519 implementation taken from busybox/wolfssl.

Assertions manually inserted to check for safety of all array accesses.

Two hour timeout, 8 GB memory limit.

RQ3: Case Study

Curve25519 algorithm implementation taken from busybox/wolfssl, region hierarchy.



Legend

fe_normalize — Function

102:107 — Qicc-identified region startline:endline

→ Region containment or function call

14 assertions checking for array access bounds

RQ3: Case Study Results

Checker/Mode	Baseline (no Qicc)	Qicc Sequential	Qicc Concurrent
CBMC	Out of memory (>8GB)	13s	11s
UA	Did not terminate (>2h)	55s	35s

Qicc enables existing verifiers to terminate in **under a minute**, for a **real** example they were previously **unable to handle**.

Case study - Threats to validity

1. Limited to one type of property - array bounds, and one example.

   Systematic analysis had examples with lots of different properties.

2. Frequency of hits and misses in real world programs is unknown.

   The cost of a miss is very small compared to the benefit of a hit.

Recap

Qicc is useful when verifying properties in large programs where limited context is sufficient.

Without Qicc, finding sufficiently small regions is tedious.

Cost of **misses** is small and scales very well with input. **(gambling is cheap)**

When Qicc **hits**, it can terminate much faster than the underlying checker. **(benefit of winning a gamble is large)**

Related work

Program Slicing for Verification

Finding relevant subset of a program for an assertion, making verification easier.

Qicc acts as a slicer that **exploits locality** of properties.

1. Weiser, M.: Program Slicing. In: Proc. of ICSE'81. pp. 439–449. IEEE Press (1981)

2. DeMillo, R.A., Pan, H., Spafford, E.H.: Critical Slicing for Software Fault Localization 21(3), 121–134 (1996)

Related work

Differential Program Verifiers

2Clever: **differential program verifier** - perform similar cycle extraction/simplification to Qicc.

Do not target program safety, do not reason about regions in isolation.

---

1. Feng, N., Hui, V., Mora, F., Chechik, M.: Scaling Client-Specific Equivalence Checking via Impact Boundary Search. In: Proc. of ASE’20. ACM (2020)

Related work

Program Transformations for Verification: Other Examples

- Lifting assertions out of inner regions. [1, 2]
- Inlining cycles with arbitrary variables. [3]

Can be combined with Qicc, by applying on expanded context after misses

1. Lai, A., Qadeer, S.: A Program Transformation for Faster Goal-Directed Search.In: Proc. of FMCAD'14. pp. 147–154. IEEE (2014)

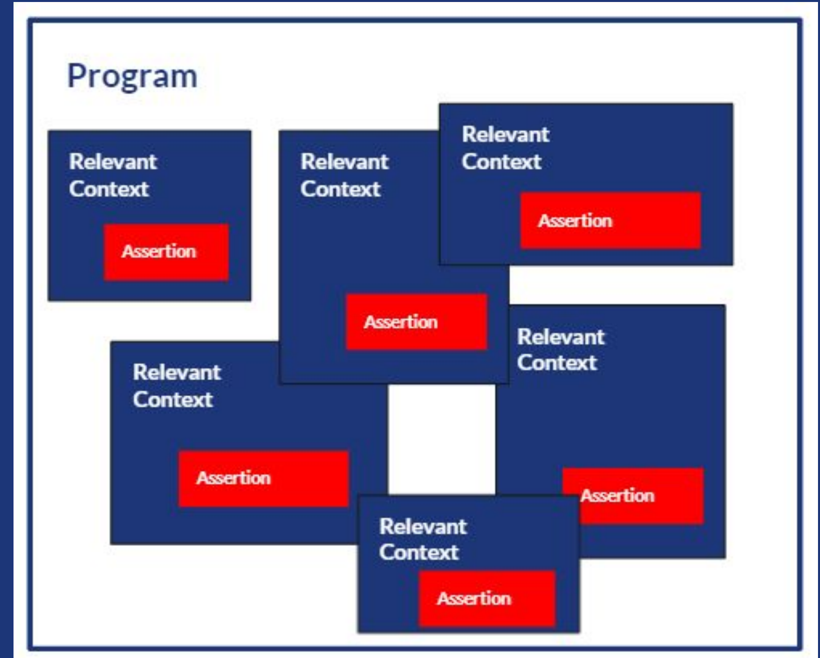
2. Gurfinkel, A., Wei, O., Chechik, M.: Model Checking Recursive Programs with Exact Predicate Abstraction. In: Proc. of ATVA'08. pp. 95–110. Springer (2008)

3. Jana, A., Khedker, U.P., Datar, A., Venkatesh, R., Niyas, C.: Scaling Bounded Model Checking by Transforming Programs with Arrays. In: Proc. of Int. Symposium on Logic-Based Program Synthesis and Transformation. pp. 275–292.Springer (2016)

Future work

- Use a combination of checkers and dynamically choose different checkers for different regions.
- Insert additional facts into extracted regions using static analysis.
- Experiment with different heuristics for regions, such as isolating expensive function calls or other operations.

Thank you!



murad@cs.toronto.edu
github.com/MuradAkh/Qicc