Verification by Gambling on Program Slices

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

[1] University of Toronto, Canada
[2] University of California, Berkeley, USA

Abstract. Automated software verification is a computationally hard problem that is often exasperated by irrelevant context. Existing verification engines address this problem with slicing techniques that are either too cautious, producing large verification condition queries, or too aggressive, sacrificing soundness. In this paper, we present a novel technique, called Qicc, that is aggressive, sound, and "a little risky." Specifically, we use procedure extraction to generate a small set of verification queries that we check with existing verification engines. If any query in the set passes verification, then the original program will pass verification. However, there is no guarantee that such a query will exist, so Qicc may waste time searching. We study the effectiveness of Qicc when it is combined with two different verification engines, finding that Qicc's extra cost is small while the rewards it brings to the analysis are significant. We evaluated Qicc on a case study—the verification of a cryptographic function in BusyBox—and found that Qicc succeeds when paired with two different verifiers, while both verifiers are unsuccessful on their own.

1 Introduction

Automated software verification tools take as input an implementation annotated with specifications and aim to return a correctness proof, or a counterexample. Over the years, a wide range of automated verification techniques have been proposed, including those based on bounded model checking, k-induction, and predicate abstraction [3, 11, 9, 6]. These techniques differ substantially and succeed on different kinds of verification tasks; however, they all have one thing in common: a better encoded problem leads to better engine performance [21].

One technique for improving problem encodings is *program slicing* [4, 23]. Program slicing techniques take a program and a slicing criterion, and return a subset of the input program based on that criterion. In the realm of verification, these techniques have been used as sound pre-processing steps that eliminate irrelevant context and focus the underlying verification engine on the properties in question. For example, verification engines usually eliminate lines of code that cannot affect assertions. When verification problems are too large, recent work has suggested using slicing as an unsound pre-processing step, the idea being that an approximate answer is better than no answer at all [7]. In this paper, we seek to retain the soundness of the former use, while achieving the reductions of

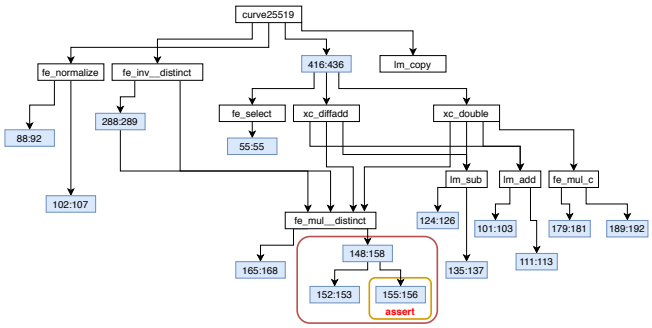


Fig. 1: Interprocedural region hierarchies of `curve25519`. Every node is a region. White nodes are functions, and blue nodes are parts of functions labeled by corresponding start and end lines of code. Arrows denote region containment. For example, the function `curve25519` calls the function `fe_select` inside a region at lines 416-436.

the latter use. We do this with a novel application of a classic program slicing technique called *procedure extraction* [16]. Procedure extraction takes a program and a set of program locations, and returns a minimal procedure that captures the behaviour of these locations. Traditionally, procedure extraction has been applied to program refactoring by automatically grouping features into functions [16]. The main challenge in applying it to verification is in deciding which program locations to extract. If we extract too many locations, the slice will remain large; if we extract too few, the slice may miss some important context.

In this paper, we propose an approach that "gambles" on a few well-thought-out slices. The cost of each gamble is small, but the reward is potentially big, often allowing us to solve previously non-terminating cases. To get an intuition for our approach, consider the example in Fig. 1 which shows the *interprocedural region hierarchies* of the `curve25519` function inside BusyBox's TLS library. We formally define regions in Sec. 2. For now, consider regions to be contiguous portions of the control-flow graph of a program with a single entry location and a single exit location. Regions can be nested, and Fig. 1 shows this nesting for `curve25519`. `curve25519` is a Diffie-Hellman function that takes a private key and returns a corresponding public key using elliptic curve cryptography, and our goal is to check whether its assertions hold. We highlight one of these assertions with a red "assert" in Fig. 1. When given the entire program as input, existing verification engines, e.g., CBMC [5] and UltimateAutomizer [14], struggle to prove this assertion (and the other assertions in the program) because they are overwhelmed by the size of the problem. However, as we discuss in more detail in Sec. 4.5, the part of the program inside the red rectangle is sufficient to prove that this assertion always holds, and focusing verifiers on this part of the program is sufficient to have them succeed. Our approach searches the regions of a program until it finds such a sufficient part of the program. When there are multiple assertions and their candidate sufficient regions do not overlap, assertions can be checked independently and in parallel.

Contributions Specifically, this paper makes the following contributions. 1. We develop a verification approach, Qicc, that searches for regions of a control-flow graph sufficient to prove assertions and checks these regions in parallel. 2. We implement a prototype of Qicc that handles a significant subset of C, and allows concurrent verification with existing verification engines as a parameter. 3. We empirically evaluate our prototype on a comprehensive case study.

Organization The rest of this paper is organized as follows. Sec. 2 gives the necessary formal background. Sec. 3 describes our approach and proves its correctness. Sec. 4.1 reports on the implementation. Sec. 4 evaluates the performance of Qicc when paired with different verification engines. Sec. 5 surveys related approaches. We conclude in Sec. 6.

2 Background

This section provides a brief overview of *control-flow automata* (CFA) which we use to model programs and specifications; *regions* which are the isolated components of CFAs that can be verified in isolation; and *cyclic region bodies* which are a special case of regions that Qicc identifies and attempts to verify.

Control Flow Automata We represent programs using *control flow automata (CFA)* borrowed from Beyer et al. [2]. Formally, a CFA (L, l_i, L_f, V, G) has a finite set of program locations L, an initial location l_i, a set of final locations L_f, a finite set of program variables V, and a finite set of control-flow edges $G \in L \times O \times L$. The set O of program operations contains assignment and assumption operations. *Assignments* are denoted by $v \leftarrow t$, where v is a program variable in V and t is a term of the same type as v. *Assumptions* are denoted by $[b]$, where b is a boolean term. *Terms* are defined inductively: constants and variables are terms, and a function application $f(t_1, t_2 ... t_n)$ of function $f : D_1, D_2 ... D_n \rightarrow D_r$ over input terms $t_1, t_2 ... t_n$ of type $D_1, D_2 ... D_n$ yields a term of type D_r. A state of a CFA is a valuation for all variables in V together with a location.

A control-flow edge $l \xrightarrow{o} l'$ represents the transfer of control from location l to l' after successfully executing an operation o. An assignment $v \leftarrow t$ is successfully executed on edge $l \xrightarrow{v \leftarrow t} l'$ if the value of v at state $s' = \{\sigma', l'\}$ is the same as the value of t at state $s = \{\sigma, l\}$. An assumption is successfully executed on edge $l \xrightarrow{[b]} l'$ if b evaluates to $\top$ at state $s = \{\sigma, l\}$. A *program path* $l_1 \xrightarrow{o_1} l_2 \xrightarrow{o_2} ... \xrightarrow{o_n} l_n$ is a sequence of edges representing a transition from the source location l_1 to the target location l_n. The path is *feasible* if every operation on the path can be successfully executed in sequence. The path is *complete* if the source location is l_i and the target location is some $l_f \in L_f$.

Program Safety and Assertions We express safety properties with assertions in the program. Intuitively, an assertion takes a predicate p as input, and checks whether p evaluates to $\top$ while executing the program. We capture this intuition formally in CFAs, by representing assertions as control-flow edges $l \xrightarrow{[\neg p]} l_{err}$,

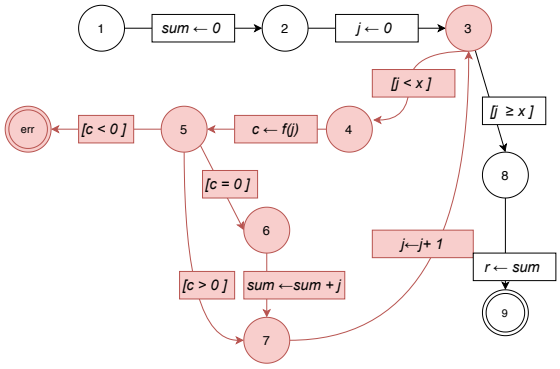


Fig. 2: An illustration of a region r (highlighted in red) in a CFA (based on the motivating example of Feng et al .[10]). The region has 3 as the initial location, and 3 and l_{err} as its final locations. A path from 1 to any location in the region must have a local suffix in r, e.g., the path 12345 has a local suffix 345.

where the target location l_{err} is a special final location representing assertion violation, and p is the asserted predicate. We require l_{err} to be reachable only through assertion edges. We say a path is *an error path* if it ends in l_{err}. An error path is *complete* if it starts from l_i, and *feasible* if every operation on the path is successfully executed.

Definition 1 (Program Safety). *A program is safe with respect to program assertions if and only if its CFA has no complete and feasible error path. In this case, we say that the program satisfies the assertion or property. If a program is not safe, then we say that the program violates the assertion or property.*

Sub-CFA and Region. $f' = (L', l'_i, L'_f, V, G')$ is a sub-CFA of $f = (L, l_i, L_f, V, G)$, where $L' \subseteq L, l_{err} \in L'_f$ and $G' \subseteq G$. A *region* r is a special sub-CFA that further requires that every path that starts at l_i and ends at some location $l' \in L' \setminus l_{err}$ must contain a local suffix which starts at l'_i and contains locations and edges exclusively from L' and G', respectively. This requirement allows r to be treated as a standalone CFA. Fig. 2 illustrates an example of r (highlighted in red) in a CFA. Since l_{err} appears in both f and r, we have the following property:

Theorem 1 (Error Suffix). *Suppose $r = (L', l'_i, L'_f, V, G')$ is a region of CFA $f = (L, l_i, L_f, V, G)$, and r contains an edge $l \xrightarrow{o} l_{err}$. If f has a complete error path p that ends with $l \xrightarrow{o} l_{err}$, then there exists a complete error path p' in r, and $|p| \geq |p'|$. Moreover, if path p is feasible, then p' is also feasible.*

Proof. Since the edge $l \xrightarrow{o} l_{err}$ is in G', and $l \in L'$, by the requirement of region, every error path p that reaches l must have a local suffix that starts at l'_i and contains locations and edges exclusively from L' and G', respectively. Therefore, the suffix is the complete error path p' in r, and $|p| \geq |p'|$. If the complete error path p is feasible, then the suffix p' is also feasible. □

Thm. 1 shows that if an assertion cannot be violated in a region r, then it also cannot be violated in the original CFA. In the example in Fig. 2, if there are no complete and feasible error paths in the highlighted region that reaches l_{err} through edge 5 $\xrightarrow{[c<0]}$ l_{err}, then there is no complete and feasible error path in the original CFA reaching l_{err} through the same edge. In other words, if every assertion is contained in some safe region, then the original CFA is also safe. Therefore, a region is a sound program slice for assertion verification.

Domination, backedge, reducibility, and strongly connected components Let f be a CFA (L, l_i, L_f, V, G). A location $i \in L$ *dominates* a location $j \in L$ if every path from l_i to j passes through i. An edge $j \xrightarrow{op} i$ is *a backedge* if i dominates j. Graph G is *reducible* if it becomes acyclic after removing all of its backedges. A *strongly connected component* (SCC) S of f is the maximal sub-graph of G with the property that there is a path from every location in S to every other location in S. A node i is an *entry point* of S if $i \in S$ and there exists a location $n \notin S$ and an edge $n \to i$. Our definition of *reducible* is equivalent to the following: a CFA is reducible if every strongly connected sub-CFA has a single entry [13].

Cyclic regions and cyclic region bodies Let f be a CFA (L, l_i, L_f, V, G), and e be a backedge $j \xrightarrow{op} i \in G$. The *cyclic region* of e is defined to be the smallest set of locations L' and edges G' such that: (1) $i, j \in L'$, $e \in G'$, (2) if some location $a \neq i$ is in L' then its predecessors location b ($\exists$ edge $b \xrightarrow{op} a \in G$) is also in L', and (3) G' is a strongly connected component. We call i the *head* of the cyclic region. A cyclic region can be also seen as a region $r = (L', i, L'_f, V, G')$, where L'_f is the set of final locations in L_f or locations with external edges $e'_{ext} \in G \setminus G'$. The inclusion of predecessors up to i in condition (2) ensures that every complete path to some location $l \in L'$ must have a local suffix that starts with i. In the example in Fig. 2, the highlighted region is a cyclic region of backedge 7 $\xrightarrow{j \leftarrow j+1}$ 3 with region head 3.

The *body* of a cyclic region r is $b = (L', i, L'_f, V, G'')$, where the graph G'' is constructed from G' by removing all the backedges to the region head i. The body of a cyclic region r is also a region: for every local suffix p in r there exists a local suffix of p which does not contain the removed backedges of G'', by taking the suffix from the last appearance of the region head in p. For the example in Fig. 2, we obtain the highlighted region r by excluding the backedge 7 $\xrightarrow{j \leftarrow j+1}$ 3 from r's graph. The path 34567345 in r has a local suffix 345 in the cyclic region.

3 Qicc

The goal of Qicc is to expedite verification of assertions by trying to remove irrelevant context. Qicc is aggressive, in that it attempts to solve the problem with the least amount of context possible and then gradually adds context until it is sufficient. It would be too expensive to attempt every possible region, so Qicc prioritizes those that are easy to identify, quick to check, and likely to work. Specifically, Qicc prioritizes cyclic region bodies, as defined in Sec. 2. This

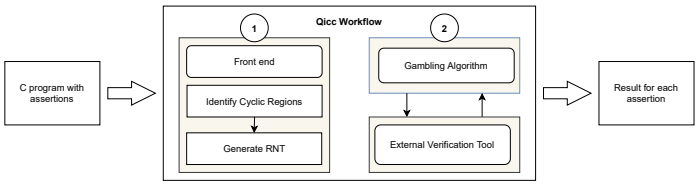


Fig. 3: Qicc Architecture

heuristic is good because cycles can be very expensive to handle—especially with techniques like Bounded Model Checking [3]. For the example in Fig. 4a, Qicc starts by trying to verify the green region in isolation, and if that context is insufficient, it attempts a larger region (green and red). If that context is still insufficient, Qicc will attempt the entire CFA for the complete context.

In this section, we describe how Qicc works. We begin with cyclic region identification (front-end step in Fig. 3) and show why checking region bodies is sound. We then describe the verification step that gambles on program slices and calls an external verifier X. We refer to the combination as Qicc+X.

3.1 Cyclic Region Identification

Algorithm 1: RNT Gen

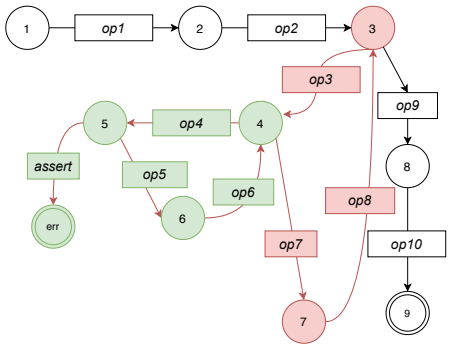
```
   Data: Reducible CFA, f
   Result: Region Nesting Tree T
1  σ ← new Stack()
   /* T is an association map */
2  T.addChild(ROOT, f)
3  σ ← push(σ, f)
4  while ¬isEmpty(σ) do
5  |  γ ← pop(σ)
6  |  IRs← InnerFinder(γ, T)
7  |  σ ← pushAll(σ, IRs)
8  return T
```

Algorithm 2: InnerFinder

```
   Data: Region γ
   Mutate: Region Nesting Tree T
   Result: Set of regions inside γ
1  components ← Tarjan(γ)
     InnerRegions ← set()
2  for S ← components do
3  |  if size(S) > 1 then
4  |  |  S_body ← RegionBody(S)
5  |  |  T.addChild(γ, S_body)
6  |  |  InnerRegions.add(S_body)
7  return InnerRegions
```

The first step of our approach is to identify all cyclic regions in the input program's CFA. Suppose $r_1 = (L_1, i_1, L_{f1}, V, G_1)$ and $r_2 = (L_2, i_2, L_{f2}, V, G_2)$ are two cyclic regions in a CFA f. We say that r_1 is an inner region of r_2 if $L_1 \subseteq L_2$, $G_1 \subseteq G_2$, and r_2's head i_2 dominates r_1's head i_1. A *Region Nesting Tree* (RNT) T for the CFA f is a tree of *cyclic region bodies* based on the nesting relationship. The root of T is f and its children are the bodies of cyclic regions that are not nested in other cyclic regions. If r_1 is nested in r_2, then r_1's body b_1 is *a descendent* of r_2's body b_2. b_1 is *a direct child* of b_2 if b_2 is the unique immediate ancestor of b_1. Fig. 4b shows the RNT T of the CFA f in Fig. 4a. The root of T is f. r_2's body b_2 is a direct child f. r_1's body b_1 is a direct child of b_2, which is also the leaf of T.

The identification algorithm is described in Alg. 1. It takes as input a *reducible* CFA f and returns all cyclic regions, organized in an RNT T. Alg. 1 first makes



(a) CFA f with nested cyclic regions r_1 and r_2 **(b)** RNT of f

Fig. 4: (a) An illustration of a CFA f with nested cyclic regions. Region r_1 (green) of backedge 6 $\xrightarrow{op6}$ 4 has locations 4,5,6 and l_{err}. Region r_2 (green and red) of backedge 7 $\xrightarrow{op8}$ 3 has locations 3,4,5,6,7 and l_{err}. r_1 is an inner region of r_2 since r_2's head 3 dominates r_1's head 4, and r_1 is a sub-CFA of r_2. (b) The *region nesting tree* (RNT) of f where b_1 and b_2 are the body of regions of r_1 and r_2, respectively.

f to be the root of T (line 2), and pushes f into a stack σ (line 3), which is the set of sub-CFAs that may have inner cyclic regions. Alg. 1 pops a sub-CFA γ from σ, and identifies the inner regions in γ by calling `InnerFinder` (Alg. 2). Alg. 2 first identifies all *Strongly Connected Components (SCCs)* in γ using Tarjan's Algorithm [22] (line 1). Since f is assumed to be reducible, every *SCC* S in γ is a cyclic region. Therefore, S's body S_{body} is added as a child of γ in T (line 5). Notice that S_{body} is not an SCC because the backedge to the *entry* is removed. Therefore, we push every identified S_{body} onto σ (line 7 in Alg. 1) to find inner cyclic regions. The algorithm continues popping sub-CFAs from σ until it becomes empty, and finally returns T.

In the example in Fig. 4a, Alg. 1 first identified r_2 as the SCC of f, and added r_2's body b_2 (backedge 7 $\xrightarrow{op8}$ 3 is removed) as a child of f in T. Then the algorithm searched on b_2 for SCC, and identified r_1. r_1's body b_1 (backedge 6 $\xrightarrow{op6}$ 4 is removed) is added as b_2's child. Finally, the algorithm failed to find SCC in b_2, and returned T.

Theorem 2 (RNT Gen Correctness). *Every node in the RNT T returned by Alg. 1 is either f or the body of some cyclic region in f.*

Proof. Every node is added to T by finding an SCC of a sub-CFA of f, treating S as a cyclic region, and adding the body of S to T (line 5 of `InnerFinder`). Therefore, it is sufficient to show that for every SCC S explored by `InnerFinder`, S is indeed a cyclic region of f.

In the first iteration, `InnerFinder` explores f. By the definition of SCCs, the `components` at line 5 of `InnerFinder` are all sub-CFAs of f, and are all cyclic (every node can reach every other node). Since f is reducible, every SCC including the members of `components` will have a single entry point and will

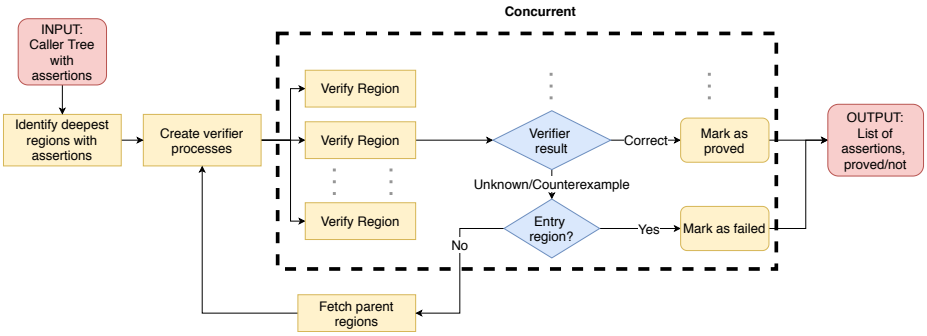


Fig. 5: The Concurrent Gambling Algorithm.

be reducible (since sub-CFAs cannot introduce new edges into existing cycles). Let S_i be the i^{th} SCC of `components` and let i be its single entry point. There must be a backedge e to i or else i would not be in the SCC. Therefore, by the definition of cyclic regions, the cyclic region of e is exactly S_i, as desired. In subsequent iterations the same argument is repeated, but with a new reducible CFA given to `InnerFinder`. □

Thm. 2 together with Thm. 1 ensure that every node in the RNT is a sound program slice to be verified against assertions. This is important for establishing the correctness of Qicc's verification process.

3.2 Gambling

The RNT T returned by Alg. 1 is a tree of possible regions that Qicc can verify to establish safety for the input CFA f. For every assertion edge $e = l \xrightarrow{assert} l_{err}$ in f, Qicc identifies the initial region $b \in T$ where no descendent of b contains the edge e. Qicc then verifies the assertion e in b by calling a verifier. If verification succeeds, then e is marked safe, and Qicc moves to the next assertion. If verification fails, then Qicc verifies the parent of b, and repeats the verification process by climbing up T until the root f is verified. Qicc returns "safe" if and only if every assertion is marked safe. Qicc returns an assertion violation if and only if some assertion is violated in T's root, f. For an input f in Fig. 4a and the RNT in Fig. 4b, Qicc identifies b_1 as the initial region that contains the assertion edge $e = 5 \xrightarrow{assert} l_{err}$, and verifies it against e. If the verification fails, then b_1's parent b_2 is checked. If the verification on b_2 still fails, then the entire CFA f is verified with the complete context. Qicc returns "safe" if e cannot be violated in one of b_1, b_2 or f, and returns "unsafe" if e is violated in f.

Theorem 3 (Partial Correctness of Qicc). *If Qicc terminates on an input CFA f, then Qicc returns "safe" if and only if f is safe.*

Proof. $\Longrightarrow$: If Qicc returns "safe", then f is indeed safe: By Thm. 2, we know that every node in the RNT T returned by Alg. 1 is a region. For each assertion,

if there exists a region $r \in T$ such that the assertion cannot be violated in r (safe region), then the assertion cannot be violated in f (Thm. 1). Since Qicc returns "safe" when it finds a safe region for every assertion, no assertion can be violated in f, and f is safe.

$\Longleftarrow$: If f is safe, then Qicc returns "safe": Suppose f is "safe", then f is a safe region where no assertion is violated. Since f is a node in the RNT T, Qicc eventually verifies f against all assertions inside, and returns "safe".

Optimizations: Batch Verification and Concurrent Verification. We highlight two key optimizations of Qicc, *batch verification* and *concurrent verification*. Instead of checking every assertion separately in a region b, *batch verification* allows all assertions to be verified in b at the same time. If verification is successful, then all assertions in b are marked safe. If verification is unsuccessful, the violated assertion is disabled in b, and b is verified again with the rest of assertions until every assertion is either safe or disabled in b. When batch verification is enabled, *concurrent verification* allows two regions r_1 and r_2 in the RNT T to be verified concurrently if they contain different assertions, Fig 5 displays the logic that can be done concurrently. If two regions have some shared assertions, then the verification result is shared and propagated from one to the other. We evaluate the impact of these optimizations in the next section.

4 Evaluation

In this section, we describe a prototype implementation of Qicc, report on the results of a systematic evaluation, and present a case-study.

4.1 Implementation

We implemented the Qicc front-end region identification and the RNT generation algorithms in OCaml as CIL plugins [20]. We used TypeScript to implement the gambling algorithm to interface with the underlying solvers. Our implementation is limited to a subset of C without recursion, and focuses on a restricted version of cyclic regions which corresponds to the intuitive notion of loops. In addition, our implementation only supports regions with a single entry and where the backedge is not a `goto` statement. Our gambling algorithm does not support mutual recursion, as it would form a cycle in the verification task tree, but lack of support for regular recursion and regions described are implementation-based. Qicc can be parametrized by different verifiers – the current implementation interfaces with CBMC and Ultimate Automizer [5, 14]. We refer to these instances as Qicc+CBMC and Qicc+UA, respectively. Qicc interacts with the verifiers by extracting each selected region as a function, and the verifier is provided with an entry function for each step in the gambling algorithm. Please refer to supplementary material for the tool source and usage instructions [3].

[3] https://github.com/MuradAkh/Qicc

4.2 Experimental Design

In this section, we evaluate Qicc's performance as a slicer for program verification techniques and report the performance advantage Qicc provides when coupled with different verifiers; we first study its effect on bounded-model-checking [3] (i.e., Qicc+CBMC) before extending the study to other techniques. Since Qicc was designed to take advantage of cases where region bodies are sufficient to prove the assertion, we say that Qicc *hits* when that is the case and *misses* otherwise. We aim, in particular, to answer the following research questions:
RQ1: What is the benefit of Qicc+CBMC when it hits? What is the cost of Qicc+CBMC when it misses?
RQ2: Does the performance benefit of Qicc extend to other verifiers?
RQ3: Can Qicc scale to large, real-world, programs?

To answer these questions, we first present a thorough systematic comparison of Qicc+CBMC with CBMC, to assess benefits of a hit and costs of a miss in different scenarios. Second, we present a smaller systematic study using Ultimate Automizer (UA) [14] (i.e., Qicc+UA vs. UA) that shows how our technique has potential beyond BMCs. Third, we show how Qicc+CBMC and Qicc+UA were able to quickly prove that array bounds are respected in a real-world program while neither CBMC nor UA were able to terminate on this example. All experiments were conducted on Ubuntu 18.04 with 8 GB of memory, and a quad-core Intel Core i7 processor at 1.8Ghz.

4.3 RQ1: Bounded Model Checking Systematic Analysis

Since Qicc's main goal is to reduce the performance penalty caused by cycles, we compared Qicc+CBMC and CBMC on 33 different configurations of synthetic loops and the benchmark as-is (baseline case).

Loop bounds We first categorize loops by their bound type: *small static bounds vs. large static bounds*, and *arbitrary / unbounded*. The bound type estimates the difficulty of verification for a bounded model checker (BMC): loops with small static bounds are cheap for a BMC to unroll, and only a small number of unrollings is necessary to convert these into equivalent loop-free programs. Loops with large static bounds are usually expensive to unroll and solve: the complexity of the loop body and degree of internal nesting both increase its difficulty. Lastly, loops with arbitrary or non-deterministic bounds cannot be verified successfully with a BMC because BMC cannot statically determine the number of necessary loop unrollings. For the purpose of our evaluation, we used 10 and 200 as small and large loop bounds, respectively.

```
1 int main(){
2   int n, x = 1;
3   while (n < 1000){
4     assert(x == 1);
5     n += x;}}
```

Fig. 6: An program with an assertion inside a loop

(a) Baseline/Linear

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

(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]
```

Fig. 7: Loop structures used to generate synthetic cases for the systematic study

Loop structure scenarios We create six general categories of relationships between loops and assertions to perform our analysis. We illustrate these relationships in Fig. 7, where *assert* denotes the location of the assertion, and *fact* denotes the location of *the furthest fact* required to guarantee the assertion always holds. For example, in Listing 6, to prove that `x == 1` on line 5, we need to know that x was initialised to 1 on line 2. In scenarios Single 2 and Double 3, Qicc is guaranteed to miss, as a required fact is outside of the loop. In the baseline scenario, Qicc is expected to perform as well as the underlying tool (plus a small baseline overhead). In Double 1 and Single 1, Qicc is guaranteed to hit, as all the necessary information lies within the inner loop. Finally, in Double 2, Qicc is guaranteed to miss once and then hit, as the furthest fact is between two loops. Fig 6 falls into Single 2 scenario. In each of these scenarios, we will vary the loop bounds to create cases.

Experiments Every valid combination of loop structure and bound type yields 34 scenarios - one baseline scenario, 3 loop bounds for both of the single loop scenarios, and 9 permutations of loop bounds for all three of the double loop scenarios. We used 14 existing benchmarks adapted from SV-COMP [1] to generate 476 (14 * 34) synthetic scenarios. This process involved adding synthetic loops; loops that were already present in the SV-COMP benchmark were left as-is. For all runs, we enabled the `unwinding-assertions` command-line option and set the unrolling limit to 200. When the unwinding assertions option is disabled, unbounded loops are considered to have the the unrolling limit n as the bound. We ran all experiments with a 600 second timeout.

Results and Analysis. Table 1 displays the number of solved instances. The columns represent different loop structures introduced earlier, and the rows represent possible bounds for loops in those structures. Qicc+CBMC is always able to solve instances that CBMC solves; in addition, Qicc is able to handle other instances where it hits, specifically in Double 1, Double 2, and Single 1 categories.

Fig. 8 plots the runtime of CBMC vs. Qicc+CBMC on all generated cases. The figure shows that in scenarios where Qicc was guaranteed to hit (Single 1

Table 1: Instances solved by CBMC and Qicc+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

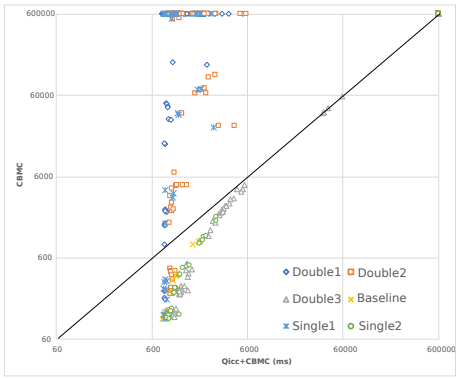


Fig. 8: Systematic study runtimes, CBMC vs Qicc+CBMC

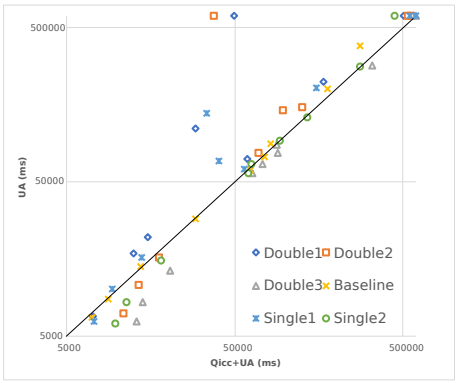


Fig. 9: Systematic study runtimes, UA vs Qicc+UA

and Double 1), the performance benefit of Qicc+CBMC was very substantial, and Qicc+CBMC was able to solve 19 (out of 39 in this category) and 93 (out of of 126) more cases than CBMC, respectively. For cases where Qicc was guaranteed to miss (Double 3 and Single 2), the overhead was very manageable, and Qicc+CBMC was able to solve the same number of instances as CBMC within the time bound. In the case of Double 2, where Qicc missed once and hit once, the timing benefit was still significant, and Qicc+CBMC was able to solve 89 (out of 126) more cases than CBMC. We can also see that our technique is most helpful with large loops, although in Double 1 scenario Qicc is able to solve more cases even with only small loops. In the baseline case (with no synthetic loops and no opportunity for Qicc to hit), both configurations solve 13/14 instances.

Answer to RQ1: When Qicc hits, its potential performance benefit is significant, and can make the difference between CBMC terminating and not terminating. The cost of the miss is at worst proportional to the depth of the program, and is generally low. The misses are proportionally more impactful on simpler examples, where the verification cost is small anyway.

Table 2: The number of solved instances in the systematic study, by loop structure

Bounds/Structures	Number of solved instances					
	Baseline	Single 1	Single 2	Double 1	Double 2	Double 3
UA	9	8	8	6	6	8
Qicc+UA	9	9	9	9	9	8

4.4 RQ2: Evaluation with Automata Verifier

We perform a similar systematic analysis to the one described in Sec. 4.3 but replacing CBMC with Ultimate Automizer (UA) [14] as the underlying verifier. We chose the state-of-the-art automata-based model checker UA to show that Qicc generalizes beyond bounded-model-checking techniques, and because of UA's top-tier performance at SV-COMP [1]. We used the same loop structure scenarios as in the BMC evaluation. Unlike in the BMC evaluation, the loop bound was not varied as we found that it had no direct impact on UA's performance. Table 2 shows the number of instances solved by UA and Qicc+UA. Qicc+UA was able to solve 8 more cases than UA, particularly when loops were doubly nested (Double 1), and even in scenarios where Qicc missed once and hit once (Double 2). Fig. 9 shows runtime for all the runs. As in case of CBMC, the overhead of a miss is minor, as no points are significantly below the diagonal apart from cheap cases. We can see hits can yield substantial performance benefit as a number of cases are significantly above the diagonal. Because of UA's inherent randomness, Qicc+UA sometimes outperformed UA in cases where this was not expected, and vice-versa; the linear cases above the diagonal in Fig. 9 are clear examples of the former. In order to show that the number of successful solves was not affected by randomness, we reran cases where Qicc+UA or UA did not terminate, but no further cases terminated after the re-run.

Answer to RQ2: Qicc shows potential when combined with an automata-based model checking technique. Just like with CBMC, the cost of a miss is low. The benefit of a hit is not as consistent as with CBMC, but Qicc+UA outperforms UA in some cases.

4.5 RQ3: Case Study

We now aim to show that Qicc can be effectively applied to real programs. To do this, we verify static array bounds in a larger (400-line) file from busybox (see `networking/tls_fe.c` in the busybox repository). The only modification made to the file was the encoding of arrays with pointers, as our implementation does not fully support C array syntax. As part of the study, we added assertions to check bounds of all instances of references to array elements.

Results and Analysis. Neither CBMC nor UA were able to prove the assertions without Qicc. CBMC was unable to complete the unrolling process without running out of memory. The unrolling limit was set to 253, same as the largest loop bound in the program. UA did not run out of memory but did not terminate within 2 hours. Qicc+UA terminated in 55s in sequential mode and 35s

in concurrent, meanwhile Qicc+CBMC terminated in 13s and 11s for sequential and concurrent modes respectively.

Qicc+UA saw a 20-second performance boost when proving the assertions concurrently (one thread per assertion). The performance benefit was more significant for Qicc+UA as UA has a larger baseline runtime overhead. Our case study contained 13 assertions and our machine had 4 cores, so not all threads were able to run in parallel. Because of this, we expect a significant performance boost on machines with more cores.

Answer to RQ3: Qicc improves the performance of both UA and CBMC on a real example. Furthermore, verifying assertions concurrently yields a substantial performance boost.

4.6 Threats to validity

We have identified two threats to validity of our evaluation. Our case study was limited to verifying array bounds on a single program, and it may not scale well on other verification tasks, where the fact is further away from an assertion. However, our systematic analysis used a variety of verification tasks with different types of loops, showing that the cost of a miss is often negligible.

Second, we have not investigated the frequency of hits or misses in real-world programs. However, our results show that the cost of a miss is likely far smaller than the benefit of a hit. The cost of a miss can be further reduced using multiple processor cores, allowing a child and a parent region to be executed concurrently.

5 Related Work

In this section, we describe tools and techniques most relevant to our approach. We include problem reduction techniques that either have similar goals, or make similar simplifications to Qicc's region identification and isolation.

Program Slicing. Program slicing, proposed by Weiser et. al [23], is a family of strategies that look for the minimal section of a program relevant to preserving a specific behavior. Slicing complements safety verification techniques in that it scopes down the potential region [8] impacting an assertion; a more specifically scoped region makes verification more tractable. For example, Cook et al. prototyped a CBMC slicer based on approximating the *cone-of-influence* of program variables [7]. Qicc is a program slicer based on assuming *locality* of assertions—that proving assertions is made easier by using and isolating nearby context. Qicc uses *regions* as the program slice for speeding up verification.

Program Transformations for Verification. The problem of exploiting relevant context for program verification was previously explored by Lai et. al [19] and Wei et. al [12] in the context of *mixed semantics*. They outlined a program transformation that lifts out assertions from procedure calls using the fact that execution returns to the caller only when contained assertions are safe. As a side-effect, their transformation also ensures that neighboring context is prioritized to prove an assertion. Their alternative approach starts with information from

the outermost context (i.e., the region where an assertion is moved out to) first in contrast to Qicc, which starts from the innermost inlined context.

Differential Program Verifiers. SymDiff [17] and 2Clever [10] are *differential program verifiers* that reason about a program's semantic differences after a change. Both techniques exploit a similar loop transformation to Qicc. SymDiff encodes each loop iteration as an inlined tail-recursive procedure [18]. 2Clever, which specifically targets changes made relative to an unchanging context, applies a variation of our *procedure extraction*-based transformation. SymDiff and 2Clever differ from Qicc as they target differential verification (and, in particular, do not target program safety (error reachability)), and they do not reason about segments in isolation.

Checking Array Properties. To check properties in large arrays, Jana et. al. [15] removed loop head and in-lined the body, assigning non-deterministic values to loop variables. Unlike Qicc, their approach still performs verification on the entire program rather than on an isolated segment. However, this approach may be effective in cases where Qicc fails to achieve convergence.

6 Conclusion

Large, complicated programs challenge verification engines which need to discover relevant context in a large space. We implemented a prototype of Qicc, instantiated it with two existing verifiers, CBMC and UA, and evaluated both instantiations, Qicc+CBMC and Qicc+UA, on 476 and 84 systematically generated test cases, respectively. We found that Qicc+CBMC solved 312 cases compared to 162 cases solved by CBMC alone; and Qicc+UA solved 53 cases compared to 45 cases solved by UA alone. We then evaluated both pairs on a case study, verification of a cryptographic function in BusyBox. Qicc+CBMC and Qicc+UA both succeeded in the verification task whereby both CBMC and UA failed. Qicc's overhead is reasonable, while its benefits are large.

To further improve performance, in the future we plan to add a mechanism to reuse information from previously attempted proofs and insert additional facts into loop bodies as assumptions. We also intend to insert additional facts such as constant variables that can be identified using static analysis. We expect that these improvements will greatly expand the number of cases where Qicc is able to improve performance of the underlying verification tool.

Acknowledgments This work was supported in part by NSF grants CNS-1739816 and CCF-1837132, by the DARPA LOGiCS project under contract FA8750-20-C-0156, by the iCyPhy center, by gifts from Intel, Amazon, and Microsoft, and by NSERC and General Motors.

References

1. Beyer, D.: Advances in Automatic Software Verification: SV-COMP 2020. In: Proc. of TACAS'20. pp. 347–367. Springer (2020)

2. Beyer, D., Gulwani, S., Schmidt, D.A.: Combining Model Checking and Data-Flow Analysis, pp. 493–540. Springer (2018)
3. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y., et al.: Bounded Model Checking. Advances in Computers **58**(11), 117–148 (2003)
4. Chalupa, M., Strejcek, J.: Evaluation of Program Slicing in Software Verification. In: Proc. of IFM'19 (2019)
5. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Proc. of TACAS'04. pp. 168–176. Springer (2004)
6. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate Abstraction of ANSI-C Programs Using SAT. Formal Methods in System Design **25**(2-3), 105–127 (2004)
7. 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
8. DeMillo, R.A., Pan, H., Spafford, E.H.: Critical Slicing for Software Fault Localization **21**(3), 121–134 (1996)
9. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software Verification Using k-Induction. In: Proc. of SAS'11. pp. 351–368. Springer (2011)
10. Feng, N., Hui, V., Mora, F., Chechik, M.: Scaling Client-Specific Equivalence Checking via Impact Boundary Search. In: Proc. of ASE'20. ACM (2020)
11. Gadelha, M.Y., Ismail, H.I., Cordeiro, L.C.: Handling Loops in Bounded Model Checking of C Programs via k-Induction. STTT **19**(1), 97–114 (2017)
12. Gurfinkel, A., Wei, O., Chechik, M.: Model Checking Recursive Programs with Exact Predicate Abstraction. In: Proc. of ATVA'08. pp. 95–110. Springer (2008)
13. Hecht, M.S., Ullman, J.D.: Characterizations of reducible flow graphs. J. ACM **21**(3), 367–375 (1974). https://doi.org/10.1145/321832.321835, https://doi.org/10.1145/321832.321835
14. Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., Podelski, A.: Ultimate Automizer with SMTInterpol. In: Proc. of TACAS'13. pp. 641–643. Springer (2013)
15. 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)
16. Komondoor, R., Horwitz, S.: Semantics-Preserving Procedure Extraction. In: Proc. of POPL'00. pp. 155–169 (2000)
17. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: Symdiff: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In: Proc. of CAV'12. pp. 712–717. Springer (2012)
18. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential Assertion Checking. In: Proc. of ESEC/FSE'13. pp. 345–355. ACM (2013)
19. Lai, A., Qadeer, S.: A Program Transformation for Faster Goal-Directed Search. In: Proc. of FMCAD'14. pp. 147–154. IEEE (2014)
20. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Proc. of TACAS'02. pp. 213–228. Springer (2002)
21. Prasad, M.R., Biere, A., Gupta, A.: A Survey of Recent Advances in SAT-Based Formal Verification. International Journal on Software Tools for Technology Transfer **7**(2), 156–173 (2005)
22. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing **1**(2), 146–160 (1972)
23. Weiser, M.: Program Slicing. In: Proc. of ICSE'81. pp. 439–449. IEEE Press (1981)