Automated bug detection

Student Name: Abdul Ghani

Supervisor Name: Stefan Dantchev

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the Department of Computer Sciences, Durham University

Abstract —

Context/Background. The field of program verification, although fraught with undecidability, has still seen much success. Many techniques have been developed in order to cope with different requirements and standards - for example, automated testing of 15K lines of video game code should differ from the formal verification of 600 lines of safety-critical code.

Aims. The aim of this project is to implement graph-search algorithms in order to detect various bugs, i.e. misuse of arrays and loops that may not terminate. In particular we will focus on detecting 'tired programmer errors' such as using unsanitized user input to index arrays and potential overflows.

Method. The user will write programs in a high level, procedural, C based language. This will be compiled into a Control Flow Graph (CFG) on which some standard optimisations will be applied. Graph-theoretic algorithms will then be used in order to find feasible paths and detect loops. Along the way we will try to find executions that lead to errors such as out of bound accesses, overflow and division by zero. We will then try to determine the monotonicity of variables and certify termination (or infinite looping). The CFG will finally be translated into machine code for a state-based virtual machine.

Results. We had success in verifying termination of various looping programs, including recursive functions where termination is not immediately obvious (e.g. the Ackermann function). We were also able to certify nontermination of ruined versions of those looping programs as well as executions leading to the previously mentioned errors.

Conclusions. In this paper we describe a bug detection method that can be built into the compilation process. This method quickly discovers a range of errors.

Keywords — Program verification, symbolic execution.

I INTRODUCTION

This paper describes the development of a compiler that attempts to verify certain desirable properties of the input during the course of the compiler. This has been identified by Hoare as a 'A grand challenge for computing research' (Hoare, 2003).

Software has become the backbone of modern society. The inevitable failures can be anything from annoying to costly to fatal - infamous examples are the failed Ariane 5 launch in 1996 (costing an estimated $500 million) and the patriot missile system (costing 28 lives).

Many diverse methods have been developed in order to detect them. The method of choice will depend on factors such as concurrency, desired level of coverage, type 1 vs. type 2 errors, the size of the program, etc. The most thorough approaches invoke constraint solvers or satisfiability modulo theories engines. These approaches, although successful, are computationally expensive and may be overkill for non-safety-critical applications.

This project instead utilises techniques from the field of symbolic execution in order to over-estimate the set of feasible paths through a program. The idea is to try to maintain an upper

and lower bound on the possible values of each variable, as well as basic relationships between variables. These bounds are determined by conditions and arithmetic expressions (see Listing 1). We will then try to find paths through loops where progress towards breaking a loop condition can be guaranteed by tracking monotonicity of variables.

Listing 1: Example of ranges.

`double x;`	$x = 0$
`input x;`	$min \leq x \leq max$
`double y = x % 8;`	$min \leq x \leq max, \quad 0 \leq y < 8$
`if (y < 4)`	
`{`	
`  ...`	$min \leq x \leq max, \quad 0 \leq y < 4$
`}`	
`else if (x >= z)`	
`{`	
`  ...`	$min(z) \leq x \leq max, \quad 4 \leq y < 8, \quad x \geq z$
`}`	

In giving up constraint solvers we trade off accuracy for speed and flexibility. We gain flexibility from the ability to symbolically simulate stacks and arrays, and we gain speed by determining path feasibility using depth first search.

The solution was implemented successfully. We managed to verify the termination of many small programs, some involving array usage and recursion.

II RELATED WORK

A Ranking functions

A well founded partial order is a partial order where there are no infinitely decending chains. If we can find a *ranking function* that maps successive loop iterations into a strictly decending chain we know that the loop must always halt. This method was popularised by Floyd (Floyd, 1967).

This method, although powerful and simple, is hard to automate. One attempt (Colón and Sipma, 2002) begins by generalising the idea of a ranking function to strongly connected components (SCS) in a control flow graph - where now the ranking function is defined on the edges of an SCS, never increases on any edge, and strictly decreases on at least one edge. Every potentially loop must live in an SCS, as we can reach any node in the loop by following it around. We start by finding a ranking function (actually, a cone of such ranking functions, as if one exists there are infinitely many) for all the SCSs in a graph. We remove the edges where the ranking function decreases, as a nonterminating loop cannot use any of these edges. If the remainder of the component is not acyclic, we repeat the procedure.

This method is 'sound' in the sense that if it succeeds in finding a ranking function for every SCS the program always halts. It may fail to find ranking functions for programs that do infact halt.

B Term rewriting systems

Briefly, a *term rewriting system* which transforms an 'object' into another 'object' by applying transformation rules to 'sub-objects'. This framework is very general and touches many ideas

of fundamental importance in computer science (i.e., the lambda calculus, semi-Thue systems, grammars). The termination properties of such systems has therefore been very well studied (Terese, 2003, p. 181).

Functional languages such as LISP and Haskell are amenable to interpretation as a rewrite system. Imperative languages, however, need more work. A method given in (Falke and Kapur, 2009) takes the possible states of the program (the values of variables, including the program coutner) to be the objects and transforms each basic block into a rewrite rule, possibly enriched with a condition given in some logic. This method performs well on imperative programs without arrays and function calls, however it does not provide immediate diagnostics on the cause of nontermination. The high level of abstraction also leads to something akin to the path divergence problem discussed in the next subsection.

C Symbolic Execution

The last two methods are powerful but are not well suited to deal with arrays and function calls/stack usage. They also require 'extra work' - transformations to, and manipulations of, structures and representations that are not in themselves useful to the compilation process. Many approaches search for bad execution paths by simulating program execution, substituting symbolic representations of variables for concrete values. Each 'probe' of the search will represent a range of possible program executions. As each probe proceeds through the program it builds a path condition - the conjunction of conditionals it has taken. This method is named *symbolic execution.*

Anand (Anand, 2012, pp. 2–3) identifies three issues with symbolic execution:

1. **Path explosion.** At every branch we may double the number of paths we must trace. While nothing can be done about this combinatorial increase, We hope to lessen the damage by keeping the overhead for each path as low as possible, by using simple constraints and appropriate data structures (see Section III.B below).
   Some other possible solutions are discussed. An interesting one is first summarising each basic block by how it affects certain variables.

2. **Complex constraints.** The path conditions generated may be intractable or even undecidable. Some more thorough program verifiers (Zhang, 2001; Gupta et al., 2008) invoke theorem provers and/or constraint solvers to try and determine the feasability of a path condition. This, while improving accuracy, will increase complexity and running time.
   In an attempt to deal with this, the intermediate language used to represent the control-flow graph will only use conditionals with inequalities between a variable and another variable or constant value. More complicated expressions will be compiled down into these simple conditions. We will then only maintain possible ranges for each variable. If a range of a variable becomes empty, the path is not feasable.

3. **Path divergence.** Innacuracies will be inevitably introduced by modelling complex software with a simpler representation. Path divergence refers to the difference between the

path followed by a symbolic execution of said model and the path that should be followed in the code a programmer wrote. In our case, variables will have different names (as the state machine has no scope, variables will be renamed) and also it is likely that compilation and optimisation will change the structure of the program to a significant degree. That being said, the representation itself can be directly executed and so detected bugs will apply to the running program.

D More similar work

A similar approach to the solution presented here is the 'lasoo' approach discussed in (Gupta et al., 2008). It begins by symbolically executing the program and finding paths that revisit some state. It then uses a constraint solver to find states that are recurrent.

Another approach is the method discussed in (Xie et al., 2017). Like us, it tries to determine the monotonicity of variables in a loop execution. For each loop it builds a path-automata, where the nodes are paths and the edges possible transitions. Feasible cycles in this automaton are then executions of the loop that could run forever. However, they don't use a symbolic engine, and cannot work with constructs such as call stacks and arrays.

III SOLUTION

A The approach

A.1 Compilation

Most programs are developed using some high level language. We will be compiling and verifying programs written in a language based on PL/0, with the addition of functions and function calls.

Listing 2: Fizzbuzz.

```
while (current < limit)
{
 if (current % 15 == 0) print("FizzBuzz");
 else if (current % 3 == 0) print("Fizz");
 else if (current % 5 == 0)  print("Buzz");
 else print(current);
 print("\n");
 current = current + 1;
}
```

Programs in this form are easy for humans to read and write but harder for computers to optimise, test for bugs, and run. For this reason we will implement a small compiler that will translate the program into machine code for a virtual machine.
Programs will be parsed in a top down predictive manner. The translation will be carried out during the parsing of the program. For this reason the parsing and translation will not be strictly context free - for example, the code generated for a `return` statement will depend on the type of the function we are currently parsing.

After translation the program will be held in memory as a Control Flow Graph (CFG):

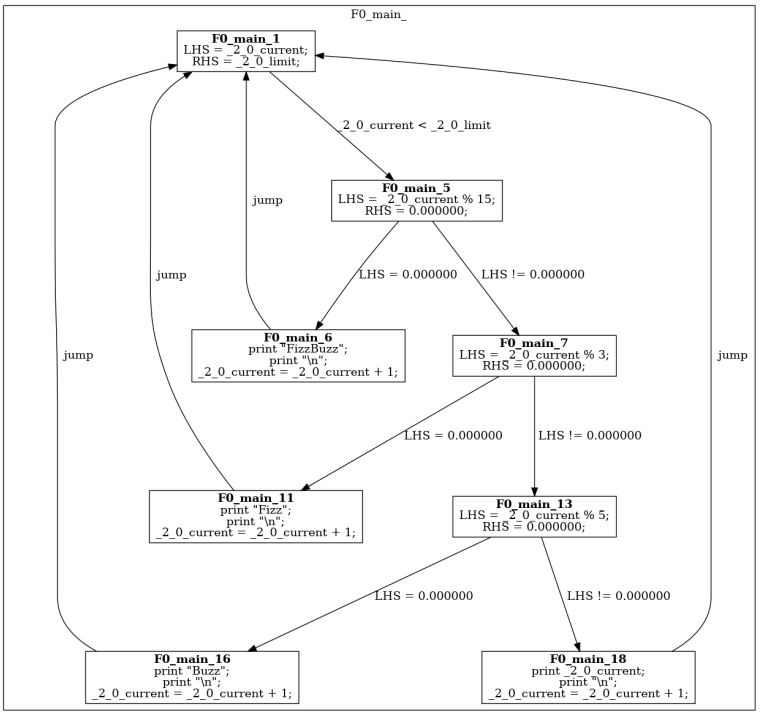

Figure 1: CFG for Fizzbuzz.

The nodes in the CFG are the so-called *basic blocks* of the program - the sections of code that are always executed in sequence, containing no jumps. These basic blocks can be seen as 'states' in a pushdown-automata-like machine equipped with a stack (used for function calls), scopeless variables, and various (macros for) three address commands. Once we have the CFG, it's easy to translate into the description language of such a machine:

```
...
LHS = _2_0_current % 5;
jumpif LHS = 0.000000 F0_main_16;
jump F0_main_18;
end

F0_main_16
print "Buzz\n";
jump F0_main_1;
end
...
```

The numbers in front of the translated variable names are to uniquely address variables in a scope free manner. See Section B for details of the implementation of the symbol table and the tagging of variables.

The CFG representation of a program is much more workable than the stream of characters initially received from the user. For example, it allows the compiler to perform certain optimisations.

A.2 Dataflow and optimisation

The flow of control, represented by the directed edges in the CFG, is a crucial element of most optimisation techniques. *Data flow analysis* involves solving a system of equalities over some lattice, usually a power set ordered by inclusion. In the sections that follow, $pred(n)$ denotes the incoming neighbours of a node n in the CFG and $succ(n)$ its successors. $start$ will denote the entry of the CFG and $exit$ the exit.

Live variables Say that a variable is *live* at some point of the program if the value it is holding at that moment could be read later on (during any possible execution that passes through that point) and that it is *dead* otherwise. We can remove statements using only dead variables, as they can have no effect on the program. For each basic block n we can simply iterate through its statements and calculate the sets of variables $used(n)$ (the variables used in that block) and $kill(n)$ (the variables assigned and not read before the end of the block). The set of live and dead variables at each block then becomes

$$live(n) = \left(\bigcup_{w \in succ(n)} live(w) \right) \setminus kill(n)$$
$$dead(n) = used(n) \setminus live(n)$$

and $live(exit) = \{\}$.

Assignment propagation It's useful to propogate assignments as far along as possible in a program. Doing so may let us simplify expressions and possibly remove conditionals.
Let $assignBasic(n)$ be the set of assignments leaving n - for example, tuples of variable names and the values assigned to them. We can find variables safe to replace with the following equations:

$$assign(n) = \left(\bigcap_{w \in pred(n)} assign(w) \right) \cup assignBasic(n)$$
$$assign(start) = assignBasic(start)$$

General formulation and solution The two examples given are meant to demonstrate the similarities and differences found in data-flow problems. In the general framework introduced by Kildall (Kildall, 1973) we find a least fixed point of some monotone function over a lattice. Generally, each node monotonically transforms some input set $in(n)$ into an output set $out(n)$. They generate a set $gen(n)$ and kill a set $kill(n)$. The whole ordeal is then parameterized by the following variables:

1. The actual 'type' of the set elements - i.e., variables, expressions etc.
2. *Forward* vs *backward* analysis. This refers to whether the set assigned to each nodes depends on its predecessors or its successors, respectively.
3. *Any* vs *all* paths. This essentially dictates whether we work with unions or intersections when calculating $in(n)$.

4. The initial output set $initOut(n)$ assigned to each node.

5. The functions $gen(n)$ and $kill(n)$.

6. The *transfer function* T_n at for each node n, which relates $in(n)$ and $out(n)$ (that is, $out(n) = T_n(in(n))$).

This insight makes it possible to implement a general algorithm to solve a range of different dataflow problems.

A simple solution to this general formulation is the *worklist algorithm*. We maintain a stack of nodes waiting to be processed. We pop nodes off this stack and run their transfer function - if there is a change, we push the successors of that node onto the stack ('successors' here might be predecessors in the case of backwards analysis).

A.3 Symbolic execution

Symbolic execution is an umbrella term encompassing many different techniques. The most careful maintain sets of symbolic equations of relationships between variables and employ solvers to see if the set is feasable. To reduce the running time of our program, and to ease implementation, we will instead maintain only simple relationships between variables (equalities and inequalities) as well as upper/lower bounds. Statements will then be allowed to manipulate this data. By a *symbolic variable* we mean one of these indeterminate variables.

A symbolic execution of the program will be a search through the CFG. During the search we will maintain a *state* which is just a set of symbolic variables and a symbolic stack. As we progress through a basic block we will affect the state as appropriate. When we reach a branch - which are the edges leaving a node - we may have to clone the state. This is because we cannot allow searches down independent paths to influence each other. Cloning the entire state on every branch would be wasteful and instead we adopt a 'copy-on-write' approach discussed in Section B below.

The verification procedure will consist of two searches. The first ignores how variables change and simply tracks the possible ranges of variables at each basic block of the CFG. A non-idempotent operation will affect the state as if it could be executed an arbitrary number of times. Similarly to data-flow analysis, we will carry on a search to the successors of some node n only if we learn something new whilst searching n. Each node is tagged with the union of the ranges of variables in each visit. (For example, if a variable always happens to be nonnegative in a basic block this will be known.) Nodes that are never visited are then removed. After this stage, we carry out a more detailed search through detected loops in order to produce certificates of termination/non-termination.

Improving variable-variable comparisons Say we encounter the condition `if` $(x < y)$. If we only restict the upper bound of x so that it is less than the upper bound of y, we lose the potentially useful information that $x < y$. We maintain this information by linking together symbolic variables with edges. We then retain those simple relationships between variables, and we also gain the ability to infer new relationships via transitivity. As an example, if we hit the condition `if` $(x < y)$, but we have previously seen the conditions `if` $(x \geq z)$ and `if` $(z > y)$ we know we cannot enter. If later we increase x by a strictly positive amount, we forget all the less-than, less-than-or-equal-to, and equality edges leaving x but keep the remaining.

Certifying termination of unnested while loops By a path through a loop we mean a simple path beginning with the loop header and ending with the exit. The path is essentially described by which conditions in the loop body it passes and fails. For example, in Listing 2 there are 4 paths, one for each `if/else` condition. We only inspect the first iteration, so there are only finitely many such paths.

We break each loop up into all possible paths from the entry to the exit. We will call a path *good* if it makes progress towards leaving the loop, possibly depending on user input, and *bad* if it can repeatedly execute without making progress towards leaving the loop, independent of user input. Essentially, we want to make sure no bad paths are ever executed, and we want to inform the user if we find this not to be the case. This will involve producing a certificate showing the conditions taken at each branch.

Characterisation and extrapolation of the motion of variables As we search down one execution path of the loop we will try to find the most positive and most negative possible change of each variable. If both of these values are positive (negative) we will guess that the variable increases (decreases) with every execution of the loop. After reaching the end of a path, we examine the variable involved in each loop condition encountered on that path and try to check that it moves towards that condition. (By 'a loop condition' I mean any condition that leaves a node in the loop.) The path is called 'good' if it moves towards any such loop condition.

It may be the case that for some number of loop iterations all relevant variables may be moving *away* from the condition but eventually meet some `if` condition that kicks us out of the loop:

```
double n = 9;
while (n < 10)
{
  n = n - 1;
  if (n ≤ 0) n = 11;
}
```

This will be combatted by attempting to extrapolate the current motion of variables into the future. During a search we may encounter some condition which the variable ranges forbid us from taking. So we first visit the second branch. If any execution of this branch always moves us towards meeting this condition, we will go ahead and take it.

Nested loops It may be that one loop is nested inside another. We form an 'inclusion tree', where the nodes are loops and an edge represents proper nesting. When we come to verify a loop in this tree we first recursively verify its children. If we decide that they're bug-free we then verify the outer loop. When encountering the header of a nested loop, we update our symbolic variables with the overestimation formed during the symbolic execution - this simulates an arbitrary number of iterations of the inner loop.

The method used to find the inclusion tree is adapted from a bioinformatical algorithm usually used to find perfect phylogenies (Gusfield, 1997):

1. Assign each loop a bitvector, where the ith bit is 1 iff the ith node appears in this loop.

2. Sort these bitvectors using radix sort.

3. For each loop, find the next highest loop that is is a superset and (if it exists) add this loop as a child.

Bug detection for arrays Arrays could be 'symbolically simulated' by creating a symbolic variable for each cell - however this is obviously very wasteful. We instead maintain two lists - the first a list of indicies pointing to contiguous ranges which (as far as we can tell) have the same value, and the second a list of pointers to symbolic variables corresponding to those ranges. For example, say the user creates an array of $10,000$ doubles. This array is represented by a single symbolic double $init$ set to 0. Now the user adds 10 to the cell with index x, $20 \leq x \leq 100$. We create a new determined symbolic double nd with value 10. The index list becomes $(20, 100, 10000)$ and the value list $(init, nd, init)$.

Given a undetermined index i this symbolic array can then build a symbolic variable which represents our belief about the range of values present in the indicies spanned by i. Continuing with the example, if we ask for the variable with index $i \in [15, 25]$ we receive a constructed variable $c \in [0, 10]$. We emit a warning in the cases where i could be negative or exceed the size of the array.

B Symbol tables and spaghetti stacks

B.1 The symbol table

An easy way of handling scoped variables is to use a so-called *spaghetti stack* (Aho et al., 2006). A spaghetti stack is usually a set of linked lists of hashtables (scopes), but during parsing we discard old scopes and only need a single linked list.
When a new scope is entered - which is marked by a { - we create a new hash table and set it up to point to the previous one. New variable declarations are put in this table. If we declare a variable with the same name multiple times in the same scope, we will find that variable in the current scope and raise an error.
When we search for a variable, we first check the present scope. If it is not found there, we recursively search for it in the parent scope. If we end up running into a `nullptr`, we are trying to find a variable that has not been declared. Otherwise we will find and return the 'most recent' declaration of that variable. Variables can be tagged with various data - some necessary for finding errors (i.e. type) and some useful for warnings (i.e. whether or not that variable has been defined).
When leaving a scope we delete the head of the spaghetti stack and forget every declaration that was inside. The symbol table keeps track of its current depth and the number of scopes it has processed in that depth in order to uniquely name each variable.

B.2 Adapting the spaghetti stack for symbolic execution

A small modification adapts the spaghetti stack into a structure useful for reducing memory used during symbolic execution.
During symbolic execution we will encounter conditions which may be true or false according to the information we currently hold. For example, we might be sure that $0 \leq i \leq 5$ and encounter the command `jumpif i >= 2 state`. We search the two different paths, one

Listing 3: A loop.

```
function main() void
{call loopheader();}

function loopheader() void
{if (x < 10) call loopbod();}

function loopbod() void {
  print(x, "\n");
  x = x + 1;
  call loopheader();
}
```

Listing 4: Not a loop.

```
while (1==1)
{
  return -1;
}
```

with $0 \leq i < 2$ and the other $2 \leq i \leq 5$. These two different branches must be independent in the sense that they cannot affect each other or future searches. One way of ensuring this would be to clone the entire state - duplicating every variable met so far. This is very wasteful, as the only difference at the moment is the single variable i.

A better way is to take advantage of the 'most recent' property of searching through a spaghetti stack. Suppose we are using hash tables to map variable names to a `SymbolicVariable` object. Instead of duplicating that hash table for independent branches, we just set up two new hash tables with pointers to their parent. If we read a variable, we search up the spaghetti stack looking for the closest occurrence of that variable. If we change that variable in some way, we first copy it into the present table so that we don't affect other branches. This method has been used before (for example in KLEE - see (Cadar et al., 2008), Section 3.2).

C Finding loops in a CFG

C.1 What is a loop?

We don't keep information about high level loop constructs when we move to the CFG representation, as not all 'loops' are while loops and not all 'while loops' are loops (see Listings 3 and 4). A tempting definition of a loop would just be a cycle, however this would be confused by branches within loops.
So how do we define a loop? We start by declaring that every valid execution that passes through a basic block in a loop must have passed through a 'header' and will eventually pass out of some exit block.

Definition 1 *A node a* dominates *a node b if every execution that passes through b must have first passed through a.*

Definition 2 *A node a* immediately dominates *a node b if a dominates b and every node n dominaiting b also dominates a. Clearly it is unique, and we denote the unique immediate dominator of a as $idom(a)$.*

As the domination relation is antisymmetric and transitive it can be used to form the *dominator tree* in which a node has as its children the nodes it immediately dominates. The node then dominates everything in its rooted subtree.

Definition 3 *An edge in a CFG is a* backedge *if its head dominates its tail.*

This is the normal definition of 'back edge' in the dominator tree.

Definition 4 *The* natural loop *of a backedge $n \to d$ is the set of all nodes that can reach n without going through d. d is the* entry point *and n the* exit point.

Once we find dominators, we can easilly find natural loops by searching backwards from the exit point.

C.2 Finding dominators

We could compute the nodes that a node n dominates by removing it and seeing which nodes become unreachable. We could also use an iterative approach, as the dominators of a node are just the intersection of the dominators of its predecessors. However these algorithms are fairly slow and requires further work to find the immediate dominators. The following algorithm, by Lengauer and Tarjan (Lengauer and Tarjan, 1979) is a better approach.
The algorithm begins by constructing a DFS tree of the CFG. As it does so, it labels the verticies in the order of arrival (i.e. preorder). Let $pre(n)$ denote the label given to node n, and $parent(n)$ its parent in the DFS tree.
After the nodes are labled, it computes 'semidominators':

Definition 5 *The* semidominator *of a is*

$$\arg\min\{pre(v) | v \to n_0 \to n_1 \to ... \to a, pre(n_i) > pre(a)\}$$

That is, the semidominator is the least numbered node with a path to a consisting of only higher labled nodes. Such a path is called a semidominator path.

We denote the semidominator of a node n as S_n. Semidominator paths can be used as counterexamples to one node dominating another. Intuitively, they 'sneak around the right' of nodes in the DFS tree and prevent them from dominating nodes below. The semidominator can be thought of as the best possible counterexample.

We will need the following fact:

Fact 1 *If $semiNum(n)$ (the* semidominator number *of n) is the label of the semidominator of n, then*

$$\begin{aligned} semiNum(n) = \min(&\{pre(v) | V \textit{ is a predecessor of } n \textit{ with } pre(v) < pre(n)\} \\ &\cup \{semiNum(u) | u \textit{ is a predecessor of } n \textit{ with } pre(u) > pre(n)\}) \end{aligned}$$

Roughly, all of the immediate predecessors of n are potential semidominators, with the higher labled ones being potential ends of semidominator paths. The semidominators of higher labled predecessors are also semidominators of the node itself. As the calculation of the semidominator depends only on the semidominators of higher labled nodes, we can start with the highest labled node and work downwards.
After calculating semidominators, we calculate the immediate dominators using the following fact:

Fact 2 (Lengauer and Tarjan (1979), Corollary 1) *Let $n \neq start$ and let u be the node which minimises $pre(S_u)$ among all nodes x satisfying $S_x \xrightarrow{+} x \xrightarrow{*} n$. Then:*

$$idom(n) = \begin{cases} S_n & \textit{if } S_n = S_u \\ S_u & \textit{otherwise} \end{cases}$$

C.3 Implementation details

As mentioned above we first process the nodes in decreasing DF order in order to calculate semidominators. We maintain for each node n the set $bucket(n)$ of nodes than n semidominates. The algorithm calculates semidominators using a union-find-like data structure. The trees in this structure will be fragments of the DFS tree formed earlier. It comes equipped with two functions:

$link(v)$: Connect $parent(v)$ with v in the forest.
$eval(v)$: If v is the root of it's connected component, return v.
Otherwise return a node n with minimum semidominator number,
such that $n \xrightarrow{*} v$, not including the root.

Now S_w is $\arg\min\{semiNum(eval(n))|n \to w\}$.
After finding S_w we place w in $bucket(S_w)$. We then call $link(w)$ and link w with its parent $parent(w)$ in the forest. Now we can implicitly find the immediate dominators of unprocessed nodes semidominated by $parent(w)$ (the nodes currently in its bucket) using Fact 2. For each such node v we calculate $u = eval(v)$. If $S_u = S_v$ then $idom(v) = parent(w)$. Otherwise, $idom(v) = idom(u)$. Note that in the second case we have only calculated $idom(v)$ *implicitly* as we have not yet found $idom(u)$. The bucket is then emptied.
Finally, after all the semidominators have been computed and the immediate dominators implicitly computed, we process the nodes in increacing DF order to fill in the blanks.

D Testing

The purpose of writing a virtual machine for the produced machine code is twofold. The first is to combat the path divergence problem discussed in Section II.C. It also allows us to observe the execution of the produced code and check that is runs as expected. The compiler was tested 'by inspection' - the produced CFG and the outputs of the compiled programs were checked manually for correctness.

Most of the testing of the verification software happened during collection of the results given in the next section. The examples were chosen so as to demonstrate all features of the solution, and in doing so these tests achieve very high code coverage. In addition, the same examples were executed under the Valgrind debugging tool. This tool helps to detect memory leaks. More importantly, it can detect bugs that lead to undefined behaviour (these bugs might not have immediately obvious effects, but lead to problems later down the line).

Finally, unit tests were carried out on sufficiently moduar parts of the code - i.e., finding loops, radix sort, building symbolic variables that represent our beliefs about some array indicies, etc.

IV RESULTS

The solution was exercised on various examples from the literature. Some of the examples have been reformatted (in superficial ways only) for brevity and readability. Execution time was recorded using the UNIX `time` command. Maximum Resident Set Size (the peak amount of active RAM allocated to the process) was also recorded using the same command. The computer used to gather these results was a dual core Intel Pentium 4405U with a clock speed of 2.1 GHz equipped with 8GB of RAM.

A No functions

Most of the following examples are taken from the appendix of (Falke and Kapur, 2009), itself a collection of examples from other papers. All the examples given in this paper terminate, so we also introduced mistakes in order to test detection of nontermination.

A.1 Bubblesort

Time: 8ms.
Maximum Resident Set Size: 4548KB.
This bubblesort implementation was correctly determined as halting. The programs obtained by removing any of the update lines, or moving either of the variables the wrong way, or inverting any of the equalities on lines 1 and 3, were correctly detected as nonterminating with an explanation given. Changing the comparitor in line 3 from $<$ to $\leq$ results in an out of bounds access, which was successfully picked up.

```
1 while (x > 0) {
2  double y = 0;
3  while (y < x) {
4   if (a[y] > a[y+1])
5     swap (a[y], a[y+1])
6   y = y + 1;
7  }
8  x = x - 1;
9 }
```

A.2 Two variables (Falke and Kapur, 2009, A.6)

Time: 6ms.
Maximum Resident Set Size: 4620KB.
This example was correctly detected as terminating. It also picked up the potential overflow in all three additions. The comparison in line 3 is compiled into the three-address-code `RHS = y + z, jumpif x > RHS ...` We then need to use two transitivity properties to classify this sample as halting - the inequalities `x > RHS = y + z`, and the fact that if `y` and `z` are increasing then so is `RHS`. Changing any of the increments in lines $2-3$ to decrements, and inverting the inequality in line 1, are detected as infinite loops with good reasons given.

```
1 while (x > y + z) {
2  y = y + 1;
3  z = z + 1;
4 }
```

A.3 Relative change (Falke and Kapur, 2009, A.8)

Time: 6ms.
Maximum Resident Set Size: 4488KB.

```
1 while (x > y) {
2   x = x + 1;
3   y = y + 2;
4 }
```

Both variables in this example are increasing - the loop only halts as y is increasing faster than x. This is noted by the program and it is classified as termiating. Inverting the condition, or swapping the constants in the addition, results in a nonterminating program that is detected.

A.4 Multiplication (Falke and Kapur, 2009, C.1)

Time: 8ms.
Maximum Resident Set Size: 4472KB.

```
1 while (x >= 0) {
2   double y = 1;
3   while (x > y) y = 2 * y;
4   x = x - 1;
5 }
```

In this example, the inner loop is seen to terminate (as y is increasing) and then the outer loop follows (as x is decreasing). Replacing the 2 in line 3 with a positive constant less than 1, or inverting any of the comparisons, results in a detected nonterminating program. The multiplication in line 3 always gives an overflow warning, so long as the constant is greater than 1.

A.5 Extrapolation

Time: 6ms.
Maximum Resident Set Size: 4448KB.

```
1 while (x > 0) {
2   x = x + 1;
3   if (x > 50) x = -1;
4 }
```

This is an example that demonstrates the extrapolation of variable motion. x is increasing away from the loop header condition, however we know it is increasing towards the condition in line 3 and therefore will leave the loop. This is accepted as halting. Flipping the condition in line 1, or turning line 2 into a decrement, result in programs that are still correctly classified as halting. Flipping the condition in line 3 results in a nonterminating loop that is detected with the correct annotated trace given.

The method does make some mistakes. An invalid overflow warning is emitted for line 2. Also, changing the `x = -1` on line 3 into `x = 1` results in a nonterminating loop that is not detected - the reason for this failure is given in the next section.

A.6 Arrays (Beyer, 2012)

Time: 8ms.
Maximum Resident Set Size: 4412KB.

```
1 if (i > 0) {
2   while (i < 100 && a[i] >= 0) {
3     i = i + a[i];
4     a[i] = a[i] - i;
5   }
6 }
```

In this example a is an array of size 100 with fully undetermined entries. This example is accepted as halting, however the user is warned that i could overflow or exceed the size of the array (as each $a[i]$ can be any positive number). Swapping lines 3 and 4 leads to similar warnings but is not accepted as halting - a 'potentially bad' path is flagged, where no variables make guaranteed progress towards relevant conditions.

B Calls and recursion

B.1 The Ackermann function

Time: 19ms.
Maximum Resident Set Size: 4808KB.
This function halts, as it is always called on strictly decreasing pairs (in the lexicographic ordering). Changing the decrement in lines 5 and 7 into an an addition of a nonnegative constant is correctly seen to be nonterminating, however the method fails to detect the same when line 8 is modified. This function is normally defined on the natural numbers, and the inequalities are given as equalities - doing so here results in a potentially nonterminating input, as m and n might both be negative.

```
1  function ack(double m, double
       n) double
2  {
3   if (m <= 0) return n + 1;
4   else if (n <= 0)
5    return ack(m-1, 1);
6   else {
7    double a = ack(m, n-1);
8    return ack(m-1, a);
9   }
10 }
```

B.2 McCarthy's 91 function

Time: 12ms.
Maximum Resident Set Size: 4680KB.
This function is a standard exercise for automated termination analysis methods. Changing the expression in line 5 into subtraction of a negative constant, or inverting the condition in line 3, result in nonterminating programs that are successfully detected by the method.

```
1  function m(double n) double
2  {
3      if (n > 100) return n-10;
4      else{
5        n = n + 11;
6        return m(m(n));
7      }
8  }
```

B.3 Greatest common divisor

Time: 11ms.
Maximum Resident Set Size: 4624KB.
This is correctly determined as halting, as each parameter moves in a direction that takes us out of that control path (i.e., if $m > n$ and m is decreasing eventually $m \leq n$). Reversing the inequality on line 9, or the order of the parameters in either of the recursive calls, is picked up as nonterminating - but not if we remove the nonnegativity condition imposed on line 2.

```
1
2  ...
   if (m <= 0 || n <= 0) return -1;
3  double result = gcd(m, n);
4  ...
5
6  function gcd(double m, double n)
       double
7  {
8   if (m == n) return m;
9   else if (m > n) return gcd(m - n, n);
10  else return gcd(m, n - m);
11 }
```

V EVALUATION

A Evaulation of the method

A.1 Pluses

As shown in the previous section the solution successfully verified a diverse range of programs, and did not struggle with array usage and/or function calls. We also managed to detect a variety of bugs - overflows, out of bounds accesses, division by zero, and potential infinite loops. The method is especially good at detecting simple (yet common) errors - such as off by one errors, wrongly assuming that variables are in a certain range (i.e. that a user input is nonnegative), equalities that are the wrong way around, and forgotten/misplaced loop counter increments.

It is also worth noting that this method is mostly independent of the source language itself, as long as it is imperative. It requires only a control flow graph of the produced target machine code (which is constructed in every compilation process that the author knows of) and the effects of each machine code statement on the state of the machine, which only need to be programmed once.

The fact that the method only needs a data structure produced anyways during compilation means that it is easily built into the compilation process. Symbolic execution itself can be used to further optimise code, for example by detecting infeasible code or useless operations.

Finally this method is very easy to use, as the only input it requires from the user is the source program and it runs automatically in the background. It does not require any modifications to the source code (i.e. annotations given in first order logic), and it does not have any learning curve (i.e. it is not a separate program with many commands and operating modes à la Valgrind/GDB).

A.2 Minuses

Path dependencies. As long as we think we can leave a path we call it terminating. Consider now two paths that can alternate forever, or one path that once entered can be left but will be revisited infinitely often - our method falsely accepts these cases as terminating. This is the case with the example given in Section A.5. If x is set to 1 we must infact leave that path in the next iteration, however we will eventually re-enter the path and repeat the cycle.

An approach discussed earlier (Xie et al., 2017) builds a graph where the nodes are loop paths and the edges possible transitions between the paths, annotated with conditions that must be met for this transition to take place. Our method ony looks for nodes with only self loops - a better approach would look for feasible cycles.

Short sightedness. Our method extrapolates as much as possible by inspecting the behaviour of the variables only on the first iteration. Complicated loops can exhibit behaviour that requires more work to uncover. For example, a variable may be decremented by another variable that is begins positive but is itself decreasing:

```
double d = 5;
while (x < y){
 x = x - d;
 d = d - 1
}
```

The solution would declare x decreasing and the loop nonterminating, although this is not the case.

Inequalities. Our method declares an inequality condition eventually broken if there is any relative change between the left and right hand sides. It's correct to say that if all there is no relative change between both sides of an equality it will hold forever, however this is not necessary for a nonterminating loop. For example, the method accepts as halting the probematic code fragment

```
double x = 1; double y = 0;
while (x != y) x = x - 2;
```

Type 1 and 2 errors. In this paper we have given an example of a nonterminating fragment that the program accepts as terminating (our modification to the Ackermann function) and a terminating fragment we deem nonterminating (Section A.2). This unfortunately means that you cannot believe either decision and must inspect any diagnostic output manually.

Division and multiplication. Our method works well with linear increments. It does not perform as well with multiplication and division, as it only extrapolates from the change that occurs first iteration. For example, in the multiplication example given above (Section A.4) it sees that y increases from 1 to 2 and assumes wrongly this will continue forever. More generally, if a variable must be in the range $[l, u]$ and it is multiplied by a nonnegative constant c it is now in the range $[cl, cu]$. If l is nonnegative the method will correctly deduce that that variable is increasing but incorrectly proceed as if that variable is increasing by the same increment each loop. This is an area where a true, high resolution symbolic representation would be useful.

B Issues during the implementation and development process. Lessons learned.

Adding unused generality. The biggest and most painful lesson I learned during the project was the harm caused by the addition of extraneous and unnecessary features (known in the SE community by various related dicta such as Feature Creep, Keep It Simple Stupid, and You Ain't Gonna Need It).

Initially I planned to only allow programs with the single variable type of double precision floating point. However, after writing the symbolic execution engine, I noticed that it would not take much work to extend the symbolic execution phase to work with any data type that can be ordered. This was indeed true and the implementation of the extention was made easy by the use of templates.

The issues began when I came to implement the loop validation phase. In this phase we try to determine the motion of variables - however not every ordered type (i.e. strings) are 'incremented' in the way doubles are. This lead to the use of type flags, downcasting to subclasses according those type flags, and methods in the base class which have empty implementations in derived classes. This probably could have been prevented via more thoughtful OOP design, but the best solution would have been to not implement them in the first place.

Towards the very end of the development of the project I decided to remove the functionality entirely. The relevant git commit registers 1349 insertions and 2199 deletions - such an invasive operation has left its scars on the quality of the code.

Implementing a compiler. For this project I decided to write my own compiler. This was a valuable learning experience. It also did not take much time and effort, as the compiler is small and uncomplicated. However, using an existing compilation toolkit (such as LLVM) would have taken a similar level of effort and would have provided several benefits, stemming from the ability to parse 'real' languages.

The simplicity of the compiler means that the source language is necessarily constrained, For example, my compiler doesn't support expressions in function parameters or array indicies. Verifying programs written in real, practical languages would have made collecting results easier, and more importantly could have found immediate real-world use (especially if it was built into a popular toolchain).

Also, for comparison purposes it would have been interesting to see how this method compares to other methods. The benchmark of choice is SV-COMP (Beyer, 2012), which publishes their test set as well as previous results and running times. These test sets are written in C, which I could have tried the method on if I used a toolchain.

My supervisor was sceptical about both of these things - there is a third lesson to be learned here, too.

VI CONCLUSIONS

This paper has described a method that catches a wide range of program errors. For diagnostic and correction purposes it produces annotated execution traces that lead to the error occuring. It can deal with array usages and function calls. Finally, it works only with structures produced during the compilation process and can itself be seamlessly built into a compiler, without any extra work required from the user.

A Future work

Addressing the issues discussed in the previous section. None of the issues mentioned there are fundamental to the method.

Extention to other programming paradigms. The solution as described works with imperative languages only - we affect some global state as we proceed through the program and through loops. The principles underlying the technique given here could be extended to stateless paradigms, such as functional and logic programming.

Assertions. Assert statements are statements inserted into code that give an invariant condition that is expected to hold at that line - i.e., $x \geq 0$. During the execution of the program these statements are checked for their validity - if an assertion is found to not hold the program is halted. It would not be hard to use this method, or symbolic execution in general, to automatically search for program executions where an assertion does not hold.

Constraint solvers. We avoided the use of constraint solvers for speed and flexibility. There are, however, many cases where speed can be sold in exchange for accuracy and thoroughness. In these cases it may be profitable to combine the method given here with constraint solvers - doing so would retain the flexibility afforded by symbolic execution (i.e. the simulation of stacks

and memory) but give us much more power to detect feasible/infeasible paths and monotonically changing variables. The assertions just discussed could be enriched with the constraint language chosen, and we would be able to check additional, custom properties (such as 'Bubblesort actually sorts').

Memory. The method could be easily extended to search for null pointer dereferences and memory leaks. Some more work, possibly using similar techniques to the ones used to simulate arrays, could be used to simulate the contents of memory and allow for pointers to be used just like other variables.

Decomposition, summarisation, and re-compilation. At present we run the method on the entire CFG of the input program. This would take a very long time for large, real world programs. If we change a single line in the program we are forced to re-verify its entirety.

A better approach might split up the program into constituent functions, loops, or otherwise dense subgraphs of the CFG. It might then verify each independently, finding some way to approximate the sparse interactions between the segments. An approach like this is 'loop summarisation' as discussed in (Tsitovich et al., 2011), where nested loops are replaced by summaries that over-approximate their effect on the state.

References

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

Saswat Anand. Techniques to facilitate symbolic execution of real-world programs. 2012.

Dirk Beyer. Competition on software verification (sv-comp), 2012.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. 2008.

Michael A Colón and Henny B Sipma. Practical methods for proving program termination. In *International Conference on Computer Aided Verification*, pages 442–454. Springer, 2002.

Stephan Falke and Deepak Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *International Conference on Automated Deduction*, pages 277–293. Springer, 2009.

Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

Ashutosh Gupta, Thomas A Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. *ACM Sigplan Notices*, 43(1):147–158, 2008.

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997. ISBN 0-521-58519-8.

Tony Hoare. The verifying compiler: A grand challenge for computing research. In *International Conference on Compiler Construction*, pages 262–272. Springer, 2003.

Gary A. Kildall. A unified approach to global program optimization. POPL '73, pages 194–206. ACM, 1973. doi: 10.1145/512927.512945.

Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979. ISSN 0164-0925. doi: 10.1145/357062.357071.

Terese. *Term Rewriting Systems (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, 2003. ISBN 0521391156.

Aliaksei Tsitovich, Natasha Sharygina, Christoph M Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 81–95. Springer, 2011.

Xiaofei Xie, Bihuan Chen, Liang Zou, Shang-Wei Lin, Yang Liu, and Xiaohong Li. Loopster: Static loop termination analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

Jian Zhang. A path-based approach to the detection of infinite looping. *Proceedings Second Asia-Pacific Conference on Quality Software*, Dec 2001. doi: 10.1109/apaqs.2001.990006.