Masaryk University
Faculty of Informatics

Improving out-of-bound access checking in Symbiotic

Bachelor's Thesis

Anna Řechtáčková

Brno, Spring 2020

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Anna Řechtáčková

Advisor: Mgr. Marek Chalupa

Acknowledgements

First of all, I would like to thank my advisor, Marek Chalupa, for his help when technical difficulties appeared, and for all his patience and guidance.

I would also like to thank to the members of the group preparing homework for IB005 for their tips on how to write a thesis and not go mad in the process and also to my boyfriend and my family who stood by me when I did my best to fail at not going mad.

Abstract

This thesis is focused on deciding validity of accesses of array elements done inside a loop using an index whose value is limited by the array's size. To achieve this, a value relations analysis of the code is implemented.

I describe the fundamentals of both the analysis and procedure that uses the analysis to decide array access validity.

The procedure has been integrated as a plugin into Symbiotic, a verification tool for C programs, for the purpose of improving Symbiotic's checking for memory safety violations. The impact of this augmentation is evaluated on a set of benchmarks.

Keywords

static analysis, program analysis, LLVM, memory safety, out-of-bound array access

Contents

Introduction **1**

1 LLVM **2**
1.1 *Basic Instructions* 2
1.2 *The Code Structure* 4
1.3 *An Example* 4
1.4 *Static Single Assignment Form and the Φ Function* 8

2 Data Structures **10**
2.1 *The Code Representation* 10
2.2 *The Representation of Relations* 12
2.2.1 Being equal 13
2.2.2 Being non-equal 13
2.2.3 Being lesser and being lesser or equal 13
2.2.4 Being a load 15
2.3 *Other Stored Information* 16
2.3.1 Allocated areas 16
2.3.2 Call relations 16

3 The Analysis **17**
3.1 *Building the Graph* 17
3.2 *The Structure Analysis* 18
3.2.1 Edge categorization 18
3.2.2 Finding loops 20
3.2.3 Collecting instruction sets 20
3.2.4 Collecting call relations 20
3.3 *The Value Relations Analysis* 22
3.3.1 Inferring from a single predecessor 22
3.3.2 Merging relations from multiple predecessors . 30
3.3.3 An example 37
3.4 *Computing Allocated Areas* 40

4 The Use of the Analysis in Symbiotic **42**
4.1 *Plugin's Broader Context* 42
4.2 *Proving Pointers' Validity* 43
4.2.1 Validity for a single `ValueRelations` instance . . 43

4.2.2 Considering pointer's multiple contexts 44

5 The Experiments **46**
5.1 *The Effect on Instrumentation and Slicing* 47
5.2 *The Effect on Resource Consumption* 49

Conclusion **51**

Bibliography **52**

A Archive Structure and Compilation **53**
A.1 *Compiling and Running the Analysis* 53
A.2 *Compiling and Running the Plugin as a Part of Symbiotic* . 54
A.3 *Provided Examples* 55

Introduction

The world relies on the products of the IT industry more and more every day. The pressure to deliver reliable applications on time and in good quality does not seem to diminish. IT companies are spending a significant amount of resources on software testing, and so there are incentives to make it cheaper. Therefore, various tools are being developed that can automate the process and avoid the costly and time-consuming manual testing.

The goal of my thesis is to write a plugin for such a tool, that would be able to differentiate between valid and invalid memory accesses. The plugin focuses on arrays accessed inside a loop using an index, and it seeks to prove that the indexed memory is not out-of-bound, i.e. the value of the index ranges from zero to the length of the array (with the size of array's elements taken into account as well).

To achieve that goal, value relations analysis is performed. This analysis examines different relations between variables in code, such as comparative relations (equality, being lesser, ...) or the relationship between a memory address and the value it stores.

The plugin has been integrated into Symbiotic [1], an open-source tool for analysis of code in the C programming language. The performance of this augmented Symbiotic has been evaluated on a set of benchmarks from the competition on software verification (SV-COMP) and indeed several previously undecided benchmarks were marked correct as a result of this enhancement.

In the first chapter, I outline the structure and instructions of the LLVM intermediate representation, since the analysis is performed on code compiled to this intermediate language. In the second chapter, I describe data structures used to store information throughout the analysis. In chapter 3, I elaborate on the process of the analysis itself, and in chapter 4, I explain how are the results of the analysis used to prove a pointer's validity. In chapter 5, I summarize the results of experiments performed on a set of benchmarks from the *MemSafety* category of SV-COMP. I conclude with the outline of a possible other uses of the analysis.

1 LLVM

The analysis described in my thesis is performed on code that is in the form of LLVM IR. Therefore I am dedicating this chapter to the basic description of this representation's components.

The LLVM project [2] is a collection of tools for optimization and code generation (the LLVM Core), intermediate code representation (the LLVM IR), compilation (Clang), various means of code verification (static analysis, code sanitization, symbolic execution), and many more.

The LLVM IR [3] comes in three different forms: the in-memory compiler representation, the on-disk bitcode, and the human-readable assembly language. As these three are equivalent, I will only describe the human-readable form. This representation will be from now on denoted as the LLVM language.

In the following sections, I first cover the most common instructions of the LLVM language. After that, I describe the high level structure these instruction constitute. Then I give an example of a more complex LLVM function and in the last section, I mention the requirements that are imposed on the code by the static single assignment form.

1.1 Basic Instructions

In this subsection, I briefly describe the most common LLVM instructions that are relevant to my thesis. The full list can be found in the LLVM Language Reference [3].

`alloca` allocates memory on the stack and returns its address

`load` retrieves a value from the memory address specified by its operand

`store` writes its first operand to the memory address specified by its second operand

`getelementptr` computes the memory address of an element of an aggregate type (e.g., the address of an array element)

`add`, `sub`, `mul`, `rem` perform the corresponding arithmetic operation (addition, subtraction, multiplication, modulo) on their two operands

`bitcast` changes the variable type without any change to the actual bits

`zext` extends its operand to the desired bitwidth without changing its value when interpreted as unsigned integer

`sext` extends its operand to the desired bitwidth without changing its value when interpreted as signed integer

`icmp` performs the comparison of its operands and returns one or zero to mark the result

`phi` returns a value depending on which block did the control come from, more on this in section 1.4

`call` calls the function given as the first operand, passing the rest of the operands as parameters, and possibly assigns the return value to a variable

`br` determines the following basic block; this instruction has both conditional and unconditional form

the unconditional form performs a jump to the basic block specified by the label passed as an operand

the conditional version performs a jump to one of the two specified blocks depending on a condition variable (usually the result of `icmp` or `phi`)

`switch` determines the following basic block from multiple possibilities depending on the value of the first operand

`ret` causes return from a function to its caller, it may or may not carry a value

The LLVM language is strongly typed. There are basic types like integers with given bitwidth, types for representing floating-point

numbers, and pointers to a type, but also aggregate types like vectors, arrays and structures.

If an instruction returns a value, it can be stored in a register labelled with an identifier (the language works with an infinite number of virtual registers). Later in the compilation process, these virtual registers are mapped to the actual physical registers which hold values during runtime.

There are two types of identifiers in the LLVM language: local and global. Local identifiers (register names and some types) begin with the `%` sign, global identifiers begin with `@`.

In the example in section 1.3, there are multiple local identifiers (`%iPtr`, `%array`, ...) and one global identifier (`@main`).

1.2 The Code Structure

A sequence of instructions together forms a basic block. A basic block has a single entry node and a single exit node and no branching in between; it can have multiple predecessor and successor blocks. Each well-formed basic block ends with a terminator instruction – an instruction, which causes the transfer of control away from its block.

In the example in 1.3, there are five basic blocks, each marked by a label (`%entry`, `%check`, ...).

A set of interconnected basic blocks forms a function. Functions together with global variables and symbol table entries comprises a module. Modules compose an LLVM program. Each module usually corresponds to a single translation unit (e.g., a file) and they can be linked using the LLVM linker.

Also, basic blocks form the control flow graph (CFG) of a function, as described in [4]. In short, a CFG is a directed graph in which nodes are basic blocks, which are connected by edges representing the control flow between them. An example of a CFG can also be found in section 1.3.

1.3 An Example

The purpose of this example is to show the most basic case of code in which the array access can be validated using the analysis described

in this thesis. An example of the analysis process can be found in subsection 3.3.3.

```
define i32 @main() {
entry:
  %iPtr = alloca i64
  %array = alloca i8, i32 10
  store i64 0, i64* %iPtr
  br label %check

check:
  %iVal = load i64, i64* %iPtr
  %cond = icmp slt i64 %iVal, 10
  br i1 %cond, label %body, label %end

body:
  %elemPtr = getelementptr i8, i8* %array, i64 %iVal
  store i8 42, i8* %elemPtr
  br label %increment

increment:
  %iInc = add i64 %iVal, 1
  store i64 %iInc, i64* %iPtr
  br label %check

end:
  ret i32 0
}
```

Figure 1.1: An LLVM function

The example 1.1 defines a function `main` which consists of five basic blocks.

As for its semantics, in the `%entry` block there is an array of ten bytes allocated. The array is then iterated using an index and every byte is set with the value 42.

The `%check` block makes the decision whether to execute another iteration or end the loop. The `%body` block is the body of the loop: it computes the next element's address and stores the value 42 in it. The

`%increment` block performs the increment of the index. The `%end` block is executed after the loop has ended and it terminates the function.

```
int main() {
    int8_t* array = alloca(10);
    for (int64_t i = 0; i < 10; i++)
        array[i] = 42;
    return 0;
}
```

Figure 1.2: The C equivalent to the LLVM code in 1.1

The function in figure 1.1 roughly corresponds to the C code in example 1.2. Nevertheless it is not the exact code that would be obtained upon compiling the C code from 1.2 to the LLVM. It was manually simplified and the numeric identifiers of registers and blocks were replaced with more expressive names.

To complete this example, in the figure 1.3 there is the CFG corresponding to the function in figure 1.1.

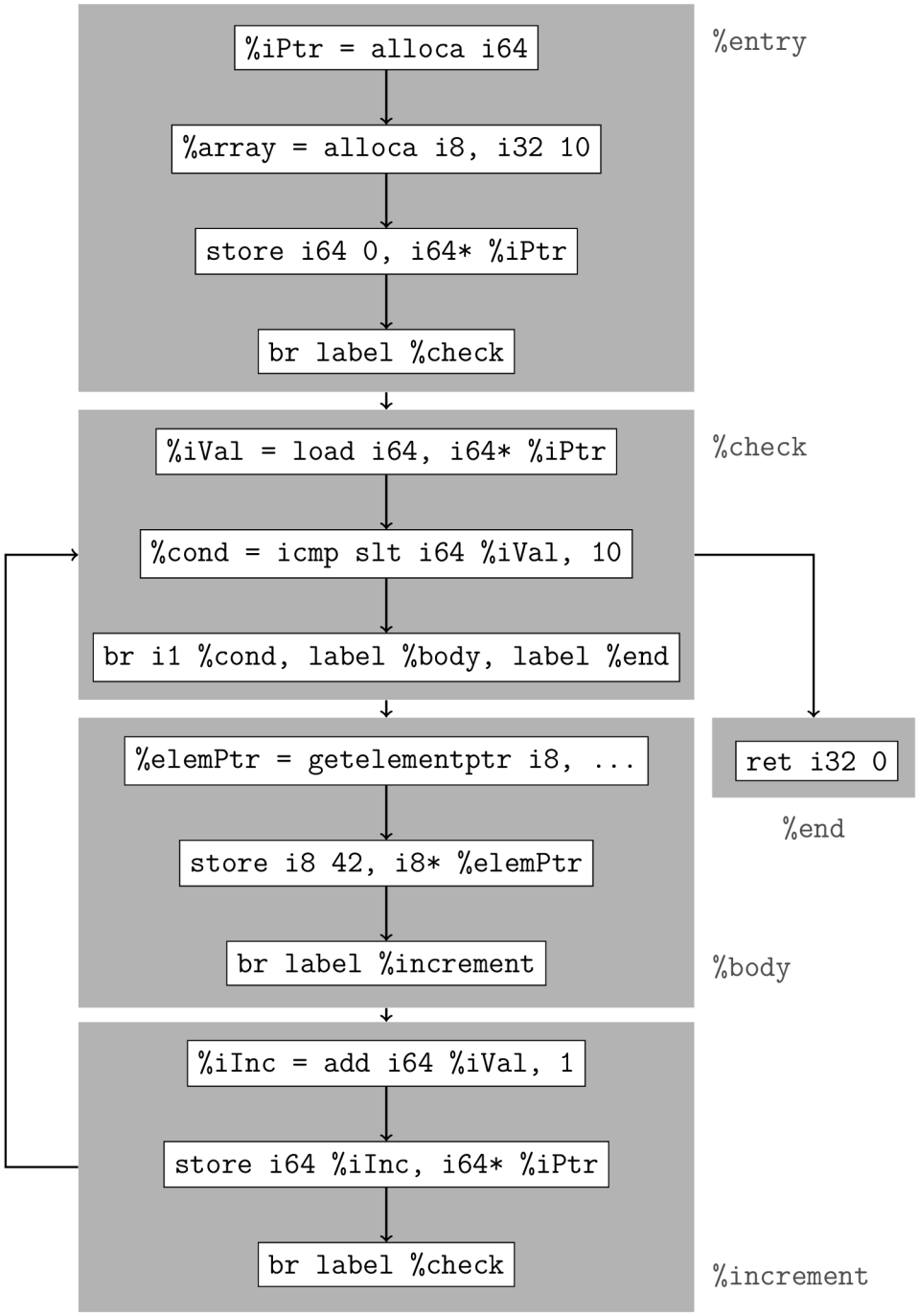

Figure 1.3: The control flow graph of the code in 1.1

1.4 Static Single Assignment Form and the Φ Function

All LLVM programs are in the static single assignment (SSA) form [5], in the sense that each of LLVM's virtual registers can only be assigned once in the code. The restriction on the number of assignments does not however extend to memory objects; the number of times a memory address can be written to is not limited.

The restrictions imposed by the SSA form complicate the situation when at a branch join, a variable may have different values depending on which branch was taken. Consider the example in 1.4. In the code, the variable a is assigned two times, so this C code cannot be transformed to the SSA-requiring LLVM language straightforwardly.

```
a = random_int();
if (a == 42) a++;
return a;
```

Figure 1.4: A C code in which a variable is assigned multiple times

For these case, the authors of [5] introduce the Φ function. This function can return a value depending on which basic block did the control come form. The code in 1.4 could compile approximately to the form in figure 1.5.

```
before:
  %a = call i32 @random_int()
  %cond = icmp eq i32 %a, 42
  br i1 %cond, label %then, label %after

then:
  %incA = add i32 %a, 1
  br label %after

after:
  %result = phi i32 [%a, %before], [%incA, %then]
  ret i32 %result
```

Figure 1.5: An LLVM code in the SSA form

In the block `%after`, the value in register `%result` is computed using the `phi` instruction which simulates the Φ function. The `%result` is set to the value of `%a` if control came directly from the `%before` block, or it is set to the value of `%incA` if control came from the `%then` block.

The understanding of the `phi` instruction will be of use in section 3.3.

2 Data Structures

In this chapter, I describe the most prominent data structures that appear in the analysis. Firstly, I describe the graph representation of the code itself. Secondly, I focus on describing the insides of the `ValueRelations` class, the fundamental structure for storing information about various relations between values. In the last section, I mention auxiliary structures used to keep track of valid allocated memory, and the structure used to store information on function calls.

This chapter focuses purely on the structures themselves. The process of their construction and use is outlined in the following two chapters.

2.1 The Code Representation

During the analysis, the code is represented as a directed graph similar to a CFG. There are some significant differences, though.

Each node in the graph, called `VRLocation`, corresponds to a location in the program and contains the information which relations hold at the point of code execution (keeps an instance of the `ValueRelations` class). Throughout the rest of the text, I mostly refer to `VRLocations` simply as locations.

Each edge represents the execution of an instruction, or an assumption about the result of some comparison. There are four types of edges which differ by the performed operation and in the thesis, I refer to them by the name in the parenthesis:

- no operation (a *noop* edge)
- an instruction (an *instruction* edge)
- an assumption a value is true or false (an *assumeBool* edge)
- an assumption a value is equal to another value (an *assumeEqual* edge)

Each operation on edge describes a change in relations between two `VRLocations`. An *instruction* edge represents the execution of the instruction, and *assume* edge represents assumption happening between the source location and the target location. In figure 2.1, there is an example of a code and its representation.

```
before:
  %ptr = alloca i32
  %a = call i32 random_int()
  %cond = icmp sle i32 %a, 0
  br i1 %cond, label %then, label %else
then:
  store i32 0, i32* %ptr
  br label %after
else:
  store i32 %a, i32* %ptr
  br label %after
after:
  %c = load i32, i32* %ptr
  ret i32 %c
```

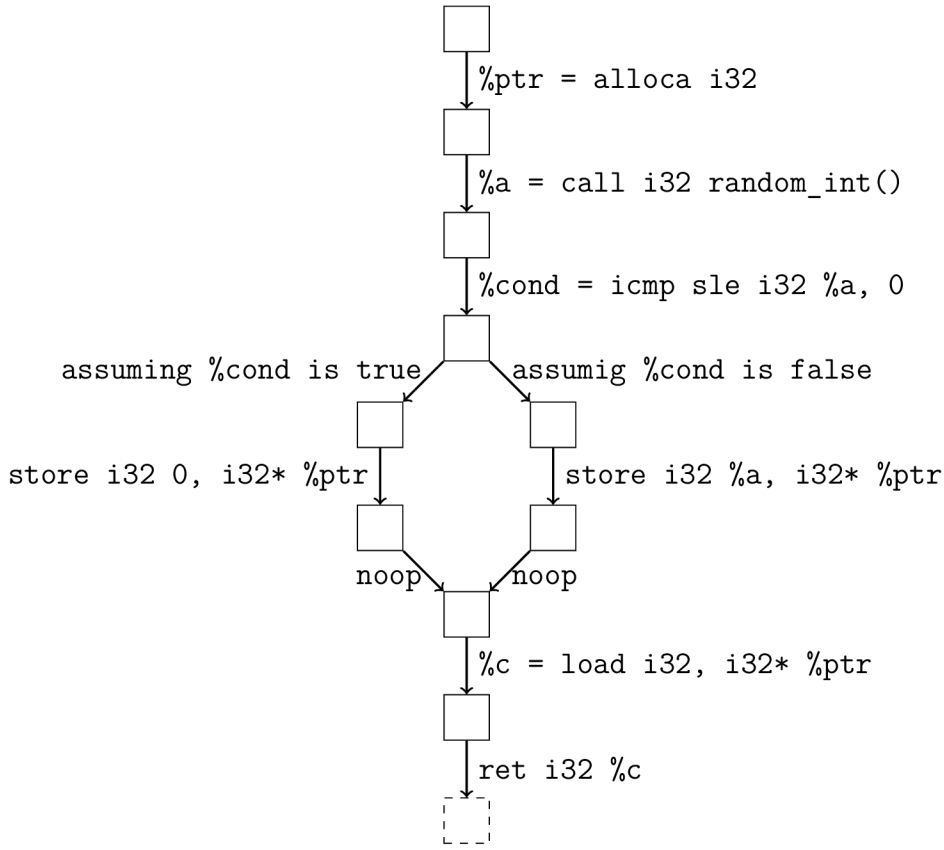

Figure 2.1: An example of a code and its representation in the analysis

2.2 The Representation of Relations

The `ValueRelation` class is a structure used to store relations between LLVM values. For the purposes of this thesis, LLVM values are constants and the values in registers. There are several different relations stored:

- being equal
- being non-equal
- being lesser
- being lesser or equal (for situations when it is not possible to decide which of those is the case)
- being a load (a relation used to keep track of values in memory; a precise definition follows in subsection 2.2.4)

I decided to store comparison relations, except for non-equality, in a graph structure.

Instances of the `EqualityBucket` class represent the nodes in the graph. Each bucket contains the values that are equal amongst themselves and the remaining relations regard these buckets, not for the actual values.

There are three types of oriented edges to express different relations between the buckets:

- lesser edges: signify lesser relation between two buckets
- lesserEqual edges: signify lesser or equal relation
- parent edges: edges in the opposite direction to every lesser and lesserEqual edge

This structure is to be called a *lesserEqual* graph. An example of such a graph can be found in subsection 2.2.3.

This graph view of the comparison relations allows for their effective representation and an easy way to check for relations, that are implied transitively.

The relations of being non-equal and being a load are stored separately from this graph, except that they also use the `EqualityBuckets`.

In the following subsections, I describe the structures in more detail.

2.2.1 Being equal

Equal values are stored together in one instance of the `EqualityBucket` class. The problem of determining equality of two values is thus reduced to deciding whether they both belong to the same bucket.

If two values are set equal, then their buckets merge into one. The new bucket participates in all relations in which at least one of the merged buckets participated.

It remains to mention that not all equality buckets are necessarily non-empty. I make use of empty buckets (which I call placeholder buckets) when merging relations from multiple source locations. I describe this in greater detail in subsection 3.3.2.

2.2.2 Being non-equal

At the time of writing this thesis, the non-equality relation is collected even though it is not used to decide the validity of pointers. It will, however, be necessary for further extensions of the current analysis.

For each equality bucket, the `ValueRelations` class stores a set of buckets that are non-equal. This way is not particularly space-efficient because each non-equality information is stored twice, once for each bucket involved.

2.2.3 Being lesser and being lesser or equal

These relations are represented by the edges between `EqualityBuckets` in the *lesserEqual* graph, as described at the beginning of this section. An example of this structure can be seen in figure 2.2.

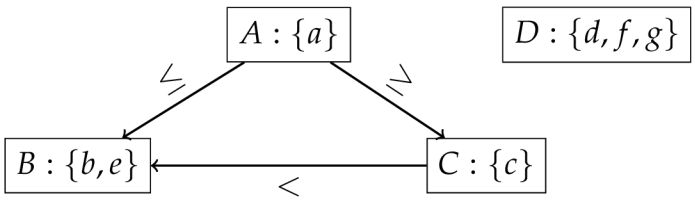

Figure 2.2: An example of a *lesserEqual* graph (parent edges omitted)

A value a is known to be related to value b if and only if a's `EqualityBucket` is reachable from b's bucket in the *lesserEqual* graph,

with the type of edges taken into account. A bucket X is strictly lesser than bucket Y if there is a path from Y to X that contains at least one lesser edge. X is lesser or equal to Y if X is Y or X is reachable from Y via both lesser and lesserEqual edges.

In figure 2.2, it holds for A that $B < A$, $C \leq A$ and D is not known to be either lesser nor lesser or equal to A.

When the structure is queried whether B is related to A, then two sets are computed for bucket A: one contains all buckets that are strictly lesser, other contains buckets that are lesser or equal, but not strictly lesser. These buckets are then iterated and checked whether some of them is B. If the structure is queried for the strict relation between B and A, then only strictly related buckets are taken into account.

To determine the two sets, the set of strictly and non-strictly related buckets is computed for each reachable bucket in depth-first search post order.

It would suffice to perform a simple breadth/depth-first search when checking whether a bucket is lesser or equal to another. Unfortunately, neither of the two searches can guarantee a correct answer when determining the strict relation. A counterexample for both is in the very graph in 2.2.

When setting the relation of being lesser or being lesser or equal, an respective edge is added between the involved `EqualityBuckets`, except when there is already a more precise relation stored (e.g., lesserEqual edge is not added if the buckets are already known to be lesser).

When setting the lesser relation and the buckets are already lesser or equal, the lesserEqual edge is removed only if it leads directly between the two buckets. Otherwise, there are multiple lesserEqual edges between the buckets and it is therefore impossible to determine which of them should be removed. This is one of the cases when multiple paths between two buckets may appear. Leaving the edge however does no harm, since the lesser edges are prioritized in querying.

Sometimes when there is already a known relation between two values, a more precise relation can be inferred. For example, when setting value a to be lesser or equal to value b, while b is already known to be lesser or equal to a, it is inferred that a is equal to b. It is even

deduced that all buckets on the path between a's bucket and b's bucket equal them as well.

2.2.4 Being a load

First, it is necessary to define the relation of being a load: if a is a value and b is a memory address, then a is a load of b / b loads a if the following statement holds: if a value is loaded from b, then it is equal to a.

This relation is essential for keeping comparison relations for values, that are not stored just in registers but also in memory.

At every given point of code execution, each memory address identifies the value that it loads (but of course, one value may be loaded from several different, independent addresses). Therefore if a memory address loads multiple LLVM values in one location, then they must be all equal.

To mirror this, each `ValueRelations` instance stores a map pairing `EqualityBuckets` so that any value from the *key* bucket loads any value from the *value* bucket. The keys must be unique in the map, their values do not have to.

When setting a load relation, it is possible to infer equality between two buckets. If a is set to be a load of b, but the graph already remembers b loading c, then a must equal c. This is illustrated in the example in 2.3.

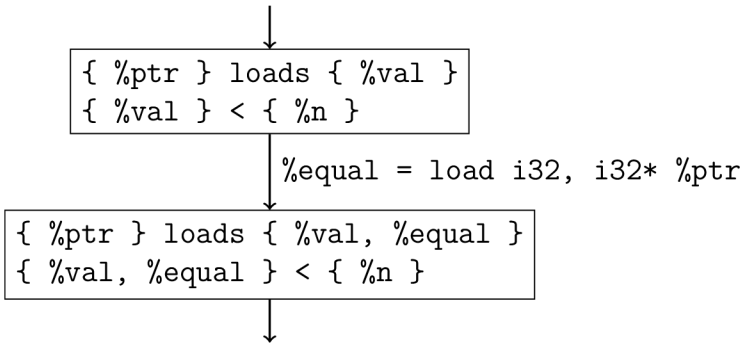

Figure 2.3: An example of inferring equality from a new load relation

2.3 Other Stored Information

It now remains to describe allocated areas and call relations representation. A collection of each of these is a part of the `StructureAnalyzer` class, since they are more of a structural aspect of the code, then related to value relations and in a way, they are independent of specific locations.

I describe the process by which the collections are computed in sections 3.2 and 3.4, but I outline the structures themselves in the following two subsections.

2.3.1 Allocated areas

For each execution of the `alloca` instruction or a call to a heap memory allocation function, information about the obtained pointer and allocated size is stored. For the correct handling of memory obtained by a call to `realloc`, each structure also keeps a pointer to the original memory.

The areas are stored in a fixed order and each location contains a bitvector in which `1` signifies the area is valid and may be accessed, `0` marks the opposite. It is necessary to keep track of this because otherwise, a pointer could be marked valid even though it addresses freed memory or memory allocated in a scope which has ended.

2.3.2 Call relations

For each function, there is a set of `CallRelation` structures. The structure contains information about one call location of a function: the pairing between real and formal parameters and a pointer to the `VRLocation` just before the `call` instruction, from which the call site's relations can be accessed.

These relations are necessary to prove the validity of pointers when it depends on values passed to the function. Their use is described in depth in section 4.2.2.

3 The Analysis

In this chapter, I describe the workings of the value relations analysis. The order of section corresponds to the order in which the phases of the analysis happen.

The actual inferring of relations between values is described in section 3.3. The remaining sections describe the auxiliary processes necessary either for the value relations analysis itself or later for the validation process.

3.1 Building the Graph

The first step is building the graph structure of code, as described in section 2.1.

The construction is done in a mostly straightforward manner. For each function, all LLVM basic blocks are traversed and mapped to `VRBasicBlocks`. These blocks are composed of `VRLocations`, and LLVM basic block's instructions are put on the edges between locations.

More complex transformations happen to the last instruction of each block (the terminator instruction, as described in section 1.2). In my analysis, I handle the following terminator instructions: `br` (both unconditional and conditional), `switch` and `ret`.

The unconditional branch instruction is simply transformed to a *noop* edge between the last `VRLocation` of instruction's basic block and the first `VRLocation` of the basic block the branch references.

The conditional branch initializes the creation of two *assumeBool* edges: one for the block to jump to if the condition is true, other for the false block.

The `switch` instruction produces multiple *assumeEqual* edges, one for each handled value of instruction's operand. At this moment, the edge to the default location is labeled a *noop* edge, even though it would be possible to collect non-equality information about the operand's value since the default location is used only if none of the possible values matches.

The `ret` instruction results in an ordinary *instruction* edge, but this edge has no target (target pointer is set to `nullptr`). I decided not to direct the edge to any `VRLocation` for some ease of memory

management and also because it does not strictly speaking make sense to describe the state of a function after the return has been performed. Nevertheless, the information about which values the function may return will be necessary when implementing the collection of relations regarding the value returned from a function.

3.2 The Structure Analysis

After the graph is built, the next phase takes place. This one includes categorizing edges which is necessary for then second step: loop detection. After that follows building sets of instructions for generation of equality relations between arithmetic operations in the value relations analysis phase. Sets of `CallRelations` (as described in section 2.3.2) are also computed.

3.2.1 Edge categorization

Firstly, the edge categorization takes place. Every edge is labelled with one of three possible values: tree, forward and back. To achieve this, a depth-first search (DFS) is run from the entry location of each function. The reasoning behind the categorization and the details of the process are explained in [6], page 609.

In short, tree edges are the edges forming the DFS tree, forward edges connect a node to its descendant in the tree, back edges connect a node to its ancestor. Authors of [6] introduce a fourth type of edge, a cross edge, which is used in all other cases, but I decided to omit cross edges and label them forward instead.

This labelling of edges allows categorizing join locations (those are `VRLocations` with multiple incoming edges) and subsequently collecting a set of instructions executed inside the body of a given loop.

Tree and forward edges

If a location has only a tree edge and forward edges coming in, then it is most probably a product of a conditional branch which used to be branching also in the original C code. Such join is to be called an if-join.

In this context, a forward edge (by the original categorization from [6]) represents the skip to the location after an if branch which has no else branch. But were the successors searched in a different order, or if there was also an else branch, a cross edge would be generated. Both these cases can be seen in figure 3.1.

Rarely, cross edge may be generated if there is a direct jump from the body of one function into another. However, the manner if-joins are handled is valid for that case also.

Back edges

Similarly, if a location has only a tree edge and back edges incoming, then it must be the joining point of a loop. Such a join is to be called a loop-join. Again this must hold, since an edge is labelled back if and only if it connects a node to its ancestor in a depth-first tree, in the way a loop comes back to the location where it started. An example can also be seen in figure 3.1.

The ability to make this distinction between various join locations in the code representation allows for their more specific treatment.

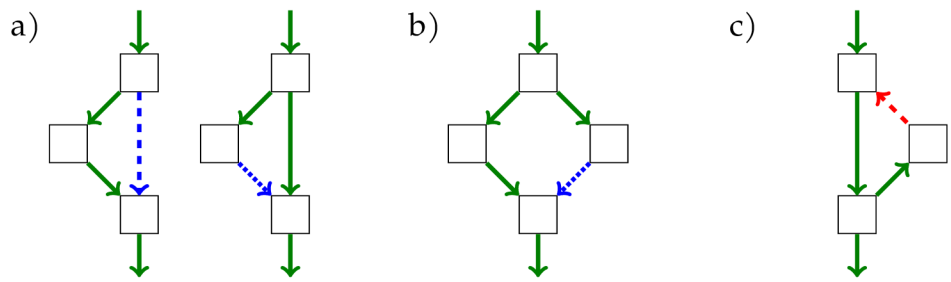

Figure 3.1:
a) Two ways to categorize edges in a graph depending on the order in which the successors are explored
b) A graph in which a cross edge would always occur
c) A graph with a loop and a back edge

Green continuous lines represent tree edges, a blue dashed line is a forward edge, a blue dotted line would be a cross edge, a red dashed line marks a back edge.

Such treatment permits inferring more relations crucial for deciding pointers' validity. This process is described further in the subsection 3.3.2.

As of yet, I have not come across a location being targeted both by a forward and a back edge, nevertheless were such a location to appear, it would not be treated in a way specific for a loop-join or an if-join, just for safety's sake.

3.2.2 Finding loops

Thanks to the ability to determine a join is a loop-join, it is possible to obtain instructions that are executed in the body of a loop. Having access to this set is essential for preserving and inferring relations at loop-joins. Again, for details see subsection 3.3.2.

For a given loop-join, the loop instructions can be computed straightforwardly by collecting two sets of instructions. The first one is obtained when going in the direction of the edges (to instructions executed later in the code) and the second is collected going opposed their direction (to instructions executed before the join location) but ignoring the tree edge coming to the join. The loop instructions are the ones in the intersection of the two.

Omitting the incoming tree edge causes that for nested loops, only inner loop's instructions are collected for the inner loop-join, while all executed instructions (including the inner loop's) are collected for the outer loop.

3.2.3 Collecting instruction sets

In this part of the structure analysis, a set is collected of all instructions of a given type, that are executed in the code (currently the `add`, `sub` and `mul` instructions are searched for). These sets are used in the value relations analysis to determine more equality relations. The process is described in subsection 3.3.1.

3.2.4 Collecting call relations

This part is again very straightforward. For each function defined in the module, all its calls are iterated. For each such call, the map

between formal and real parameters is computed and it is stored with the call location to be used later in pointer validation, as described in 4.2.

3.3 The Value Relations Analysis

This section describes the way value relations are collected from the code representation. At the end of this section, there is an example of the analysis run when processing a function with non-trivial behavior.

High level view

To determine the value relations, the code representation is traversed location by location. This analysis pass through the code is repeated multiple times until no new relations are added, although the total number of passes can be limited.

The necessary number of passes to reach the fixpoint depends on the number of consecutive loops in each graph (with a constant multiplier). The majority of programs I came across needed no more than four passes.

Each function is processed separately from the others, because relations from calls are included later when deciding the validity of a pointer.

When inferring relations for a location, only the relations in its immediate predecessors and the information from incoming edges is taken into account. The gathering of relations proceeds differently depending on how many predecessors the location has – locations with zero, one, or more incoming edges are processed in completely distinct ways.

If a location has no predecessors, no relations can be added and so there are no relations in such location. I describe both remaining cases in the following subsections.

3.3.1 Inferring from a single predecessor

The gathering of relations over a single edge happens in three steps:

- firstly, the relations from the source location are copied to the target location
- secondly, some load relations may be invalidated and removed; this step happens only for *instruction* edges
- lastly, new relations are added based on the information the edge carries

Copying Relations from the Source Location

The first step works on the assumption, that if some relations hold in the location's predecessor, then most of them are valid at the location also.

In the implementation, all relations are copied, even those that are not necessarily valid at the target location. An *instruction* edge can invalidate a load relation, if its instruction may write to memory; the removal of the outdated load happens in the second step. An *assume* edge can invalidate a comparative relation if it were to add a conflicting relation; the removal of this conflict-causing relation happens in the third step. It is safe to copy all relations over a *noop* edge, for obvious reasons.

The Invalidation of Loads

The second step happens only when passing over an *instruction* edge. It serves to remove load relations that may be invalidated by the instruction.

Assuming value A is a load of the memory address B at the source location, then that load relation is invalidated if the edge's instruction might overwrite the value A.

Fortunately, under certain circumstances, it can be concluded that a given instruction surely does not write to some memory address.

The first control is a call to several methods of the LLVM Instruction class. These methods mark instructions that certainly do not access memory. These methods allow ignoring instructions like `add` or `bitcast`, which only regard their arguments.

Also, some debug instructions are set to be ignored even if they may write to memory.

Handling the `store` instruction

The handling of the `store` instruction is the most specialized, since the instruction does access memory, but in the most controlled manner.

In all cases, the `store` invalidates the load relation for the memory it writes to. But some other loads may be invalidated still.

If the `store` writes to an address specified by a constant value, then all loads are invalidated, because it cannot be said which of them would be affected.

Then there are the cases, when the `store` writes to an address specified by a variable.

Properties of memory addresses

Before elaborating on details, two properties of a memory address must be introduced.

The first property is that a memory address may have an alias. An alias is a different name for the same memory location (a different register holding the same address). A memory address may have an alias, if it has been stored, because then the value may be loaded later under a different name. If the address can't have been stored, then it surely has no alias.

The second property is whether the memory address has a known origin. Currently, the origin of a memory address is deemed known, if the address is a result of the `alloca` instruction or equal to such a result. All memory whose origin is unknown is also considered aliased.

A memory address is considered clean if it cannot have an alias and it is of a known origin.

Invalidating loads by the `store`'s properties

If it is possible that the memory the `store` writes to might be an alias of a load relation's pointer, then the load must be invalidated. That can happen in two cases: If the `store`'s memory address does not have a known origin, then a load from any address which may have an alias must be unset, because the store's address might be that very alias. If the origin of the `store`'s memory address is known but it still may have an alias, then only loads with unknown origin must be invalidated, aliased loads with a known origin can be safely kept.

If the `store`'s address is clean, then no load has to be unset, because there is no way the `store` might affect them.

In the example in figure 3.2, there is a code and the load relations for its last basic block.

The `load` instruction at the beginning of the `%after` basic block generates the load relation between `%ptr` and the loaded value `%z`.

```
%ptr = alloca i8*
%a = alloca i8
%b = alloca i8
store i8* %a, i8** %ptr
%c = call i1 @rand_bool()
br i1 %c, label %then,
   label %after

then:
store i8* %b, i8** %ptr
br label %else

after:
%z = load i8*, i8** %ptr
store i8 0, %b
store i8 42, %z
br label %somewhereNext
```

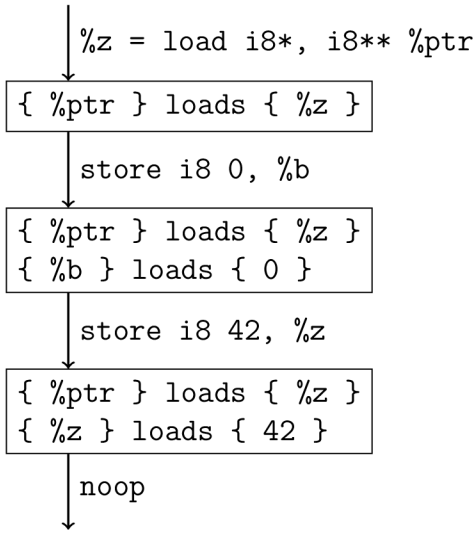

Figure 3.2: A code with aliased values %a and %b and a value with unknown origin %z, and the load relations in its last block

The `store i8 0, %b` does not invalidate it, because the even though `%b` may have an alias, the origin of `%ptr` is known. The load relation between `%b` and `0` is added.

The line `store i8 42, %z` however immediately invalidates this newly added relation, because the origin of `%z` is unknown and since `%b` may have an alias, its load relation must be invalidated. Nevertheless, the load relation for `%ptr` is kept, because `%ptr` cannot have an alias.

Handling other instructions that write to memory

Having handled instructions that do not write to memory and the `store` instruction, it remains to consider the cases when the instruction does write to memory but is not store, nor one of the ignored debug instructions.

Usually, but not always, this suggests a `call` instruction. In general, it is not possible to assume anything about the way the called function uses memory, especially where it writes to. Consequently, all loads must be invalidated when such an instruction is encountered.

The mock functions that are used in the benchmarks of SV-COMP (`__VERIFIER_nondet_int`, `__VERIFIER_nondet_char`) are manually set to be considered pure (not writing to memory) because otherwise, they would cause invalidating relations that are essential for proving the validity of some memory accesses. This is a valid assumption, as these functions are to be treated as having no side effects by the rules of the competition, as stated in [7].

The next step would be to mark safe also pure functions from the C standard library or even the C POSIX library, but for now I decided against it. Before taking this next step I plan to look into the possibility of deciding whether the call is to the actual library implementation, or to a user-provided implementation, which may be flawed.

Of course, there could be a user-defined function with side effects named `__VERIFIER_*`, but I suppose there is much smaller probability of a user naming a function that way.

The Inference of New Relations

After copying the relations from predecessor and removing the outdated loads, new relations are inferred depending the type of the edge and the information it carries.

For a *noop* edge, no new relations are generated. For an *assumeEqual* edge, an equality relation is generated between the values the edge carries.

Inferring relations from an *assumeBool* edge

For an *assumeBool* edge, a comparison relation is generated if the condition is a product of the `icmp` instruction. The reasoning becomes more complicated when the edge carries a relation which is in conflict with another relation already stored. This can happen when the code contains nested branching with mutually exclusive conditions. In such case, all relations regarding the problematic variables are unset, but since the location is unreachable, it does not matter that relations are lost.

Handling the `phi` instruction

If the condition on the *assumeBool* edge is the result of a `phi` instruction, the inference becomes a bit more complicated. The basic idea is that if the value returned by `phi` could have been obtained only by coming from one specific block, then relations from the last location in this block can all be merged to the edge's target location.

Consider the example in 3.3.

In the example, the location `B` is reachable only from `A`. It is because if the control came from the last instruction in `%lt` basic block, then the value of `%c` is unconditionally set to `false` and therefore the control is always directed to the location `C`.

The value of `%c` can only be set to `true` if control came from the `%rt` basic block (and `%v` has been set to `true`). If this situation can be identified in the analysis, it is safe to merge all relations from `A` to `B`, as if there was a direct *noop* edge between them.

This situation may occur when there is a condition composed of several simpler expressions connected via a lazily evaluated logical operator (`&&` or `||`). The example shown in figure 3.3 could represent the evaluation of `val && rand_bool()`.

Inferring relations from an *instruction* edge

Instruction edges are processed in the most complicated way, because different instruction types are processed individually.

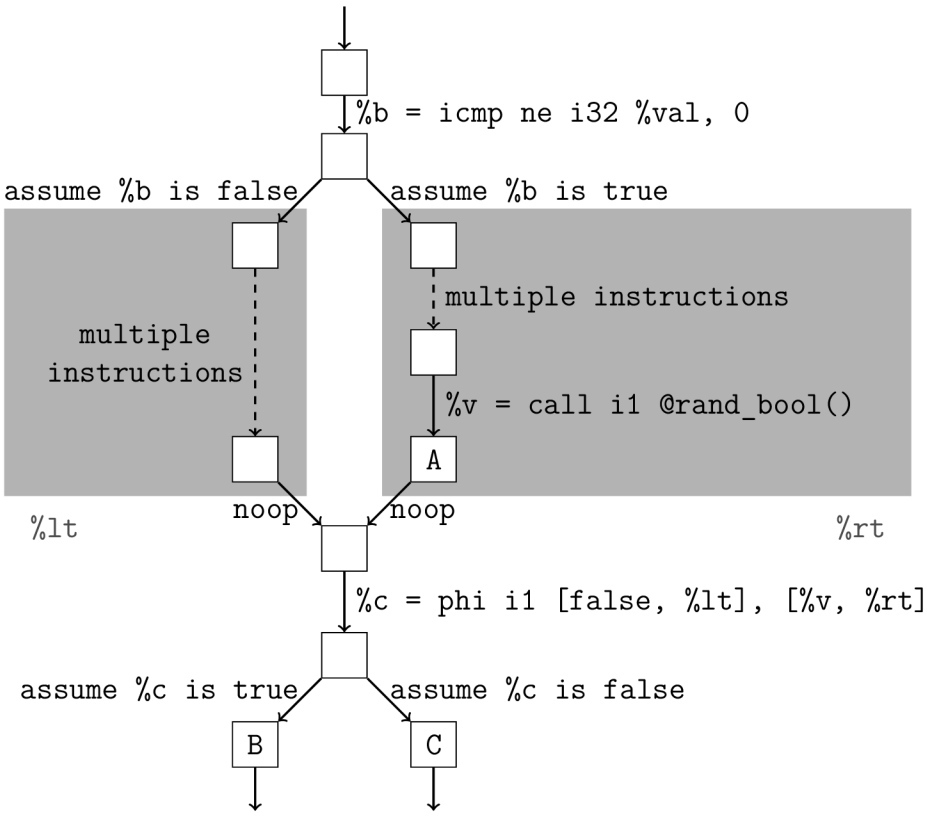

Figure 3.3: A graph representation of a code using the `phi` instruction

In some cases, the inference of a relation depends on another relation already being stored. Then the relations from the immediately preceding location minus the invalidated loads are considered.

I briefly describe the handling of several instructions.

- `load` and `store` instructions generate a load relation
- the `getelementptr` instruction generates an equality relation
  - between its result and its pointer operand if all indices are zero
  - between its and other `getelementptr`'s result if all pairs of indices on corresponding positions are equal
- the `z/sext` and lossless cast instructions generate the equality relation between their parameter and the returned value
- the `add`, `sub` and `mul` instructions generate multiple comparison relations depending on whether their operands are constants or variables and whether those are greater or equal to zero and greater or equal to one; I mention only those relations I see as the most important.
  - if both operands are equal to both operands of other instruction of the same type in the same order, then the results are set equal (to check all instructions of the same type, the instruction sets precomputed in the structure analysis are used)
  - (for `add` and `mul` only) if both operands are equal to both operands of other instruction of the same type in reversed order then the results are set equal (because of operations' commutativity; again uses instruction sets)
  - (described for `add`, for `sub` or a negative value likewise) if a positive value is added to a variable, then the result is greater than the variable
  - (described for `add`, for `sub` or minus one likewise) if one of the operands is a constant equal to one, then all values that are directly greater than the other parameter are set to be greater or equal to the result; this implication describes the relationship:

$$\forall\, p, q \in \mathbb{Z} : p < q \implies p + 1 \leq q$$

  - (described for `add`, for `sub` likewise) if one of the operands is constant and the other has a lower bound, then the result is set to relate to the sum of the constant operand and the bound in the same way the non-constant operand relates to its bound

    $$\forall\, b, c, x \in \mathbb{Z} : b\, R\, x \implies (b + c)\, R\, (x + c)$$

    where R is either $=$, $<$ or $\leq$

- the `rem` instruction generates comparison relations if the dividend is greater than zero; in that case the result is set to be greater or equal to zero and lesser than the divisor

3.3.2 Merging relations from multiple predecessors

In this subsection, I describe how relations from multiple sources are combined. The merge again happens in three phases:

- firstly, only comparative relations are regarded
- secondly, only load relations are taken into account
- lastly, more comparative relations are collected using the load relations

I describe each phase in turn.

When merging values from multiple predecessors, the inferring is dependent purely on the relations in predecessors, the information from edges is ignored.

For loop-joins, the incoming edges are mostly *noop* edges, but in the case of if-joins, this causes disregarding some *assume* edges. I decided not to consider them since I did not find a way of working with them that would not be wasteful in terms of memory and the information from the edges was not needed to achieve my current goal.

Merging Comparative Relations

In this phase, relations from all predecessors are intersected. This means that if A is in a comparative relation to B in all predecessors, then A is set to be in the same comparative relation to B at the join location also.

Even more merging happens if the join is labelled as a loop-join. In this case, all comparative relations from the source location of the incoming tree edge can be added. These are the relations that hold before the start of the loop.

Should a conflicting relation be inferred later in the loop, the join location will not be reachable from the body of the loop, because no code after the conflicting branch will be reachable.

Merging Load Relations

Same as for the comparative relations merge, the first step is to intersect loads that hold in all predecessors.

After that, again, comes more relations collecting from the tree predecessor in loop-joins. From the set of instructions executed in the body of the loop (precomputed in the structure analysis), a set of all addresses whose content is overwritten through the course of the loop is obtained. Using this set, it can be determined which load relations from the tree predecessor remain valid in the loop's body and these loads are added to the join location. This part allows remembering loads for memory addresses, that are only read from in the loop but not modified (like the length of an array)

When computing the overwritten addresses, it may happen that a load from an address that is not yet defined at the join location is invalidated. An example of code where this could be a problem is in figure 3.4.

In the code representation, the join location is just before the execution of the call to `rand_bool`, and it is targeted both from the block `%before` and the `%body`.

In the join's tree predecessor, the fact that `%intPtr` loads `0` is stored. Nevertheless, the load that is directly invalidated (by the rules described in subsection 3.3.1) is the load from `%otherPtr` which is not yet defined at the join location. If this was not accounted for, then the load from `%intPtr` would be copied to the join location even thought the load does not hold if control came from the `%body` block.

To avoid inferring relations that do not hold, the information which loads are invalidated indirectly is added too, when collecting the set of overwritten addresses (e.g., value at the address `%otherPtr` is overwritten, and any pointer loaded from `%ptrPtr` is overwritten).

```
before:
%ptrPtr = alloca i32*
%intPtr = alloca i32
store i32* %intPtr, i32** %ptrPtr
store i32 0, %intPtr
br label %check

check:
%cond = call i1 @rand_bool()
br i1 %cond, label %body, label %after

body:
%otherPtr = load i32*, i32** %ptrPtr
store i32 1, %otherPtr
br label %check
```

Figure 3.4: A code in which a load is invalidated indirectly

Merging Relations by Loads

In this last phase, there are more comparative relations inferred based on the information from the load relations. The phase makes use of the fact that a value loaded from a particular address can take part in the same relations in different predecessors, even though these loaded values are not equal amongst themselves. The considered situation is show in figure 3.5.

Placeholder buckets are used to represent these relations. These are `EqualityBuckets` that hold no value; they purely enable the representation of comparative relations. But most importantly, each placeholder bucket participates in a load relation – the bucket is set to be the load from the address which initiated its creation.

Relations regarding placeholder buckets are copied over edges together with the rest of the relations. When an instruction that loads from/stores into the bucket's memory address, then the loaded/stored value is added to the placeholder bucket, which thus becomes an ordinary value-filled bucket.

The relations are again inferred in two steps and in both steps, the inference happens only for loop-joins. Any new relations can be added

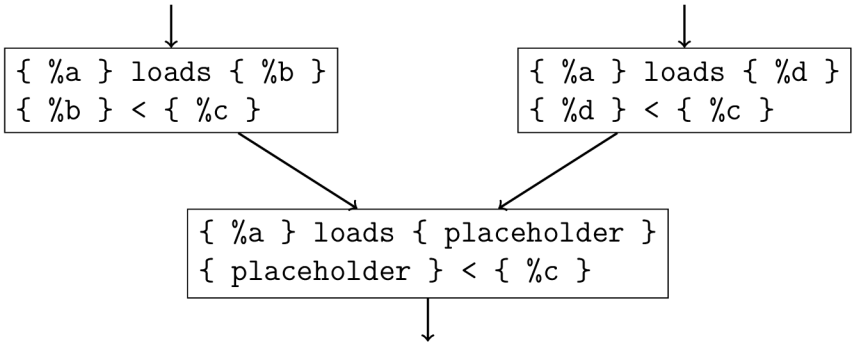

Figure 3.5: Intersecting relations by loads

only if the whole code has been passed by the analysis at least once, since both steps look for a relation present in all the predecessors, and the second one checks for relations in the loop locations also.

Also, only the memory addresses which do not have a load set from the previous phase are accounted for.

Inferring relations about an initial value

This procedure happens only for the loop-joins with exactly two predecessors. The majority of joins has only two predecessors, and a more general solution would be unduly complicated.

This step serves to infer relations in the situation when a memory is initialized with a value and that value only increments/decrements in the loop's body and is then stored back to the same address. It can then be inferred that throughout the loop, the addressed value is greater or equal/lesser or equal to the value it has been initialized with. This situation is illustrated in figure 3.6.

There, in the block before the loop, an initial value `0` is stored to the memory address `%iPtr` and so in the last location before the loop-join, there is a load relation between the two.

In the last location of the `%body` basic block, the value `%iInc` is a load of `%iPtr` and at the same time, `%iInc` is greater than the value of `%iVal`. And what more, `%iVal` is set to be the load of `%iPtr` before the value that would load from `%iPtr` is overwritten (in the example, that is immediately after the join location). So it is possible to conclude, that the value at `%iPtr` has incremented in the loop's body.

```
before:
  %iPtr = alloca i32
  store i32 0, i32* iPtr
  br label %check
check:
  %c = call i1 @rand_bool()
  br i1 %c, label %body, label %after
body:
  %iVal = load i32, i32* %iPtr
  %iInc = add i32 %iVal, 1
  store i32 %iInc, i32* %iPtr
  br label %check
```

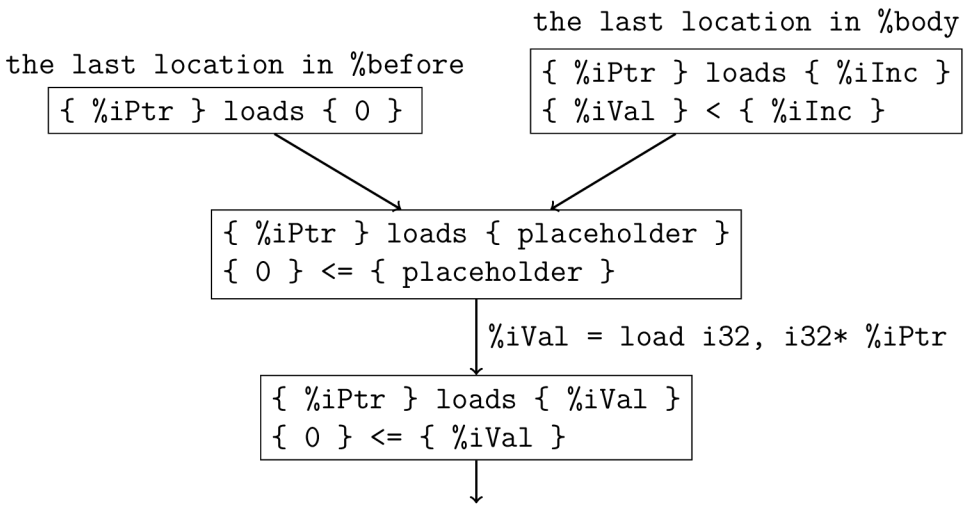

Figure 3.6: A code in which a value in memory is only incremented in a loop

In such a case, it can be concluded that inside the body of the loop, the value loaded from `%iPtr` is surely greater than the original value or equal to it.

A loaded value can be analogously set lesser or equal, if the loaded value is decremented in the loop's body.

This process is essential for inferring that an index used to address an array element is greater or equal to zero in all loop's iterations, if zero-initialized.

Inferring relations about invalidated loads

This procedure enables keeping some information about a value loaded from a memory address even though the value it loads is overwritten inside the loop.

Firstly, a set of `VRLocations` is computed (for the remainder, each `VRLocation` stores the relations valid at a given point of the program). This set contains the join's tree predecessor and also all `VRLocations` which are targeted by an *instruction* edge, whose instructions invalidates the considered memory address. Note that these are exactly the locations that contain the relations for any value which the memory address might hold during the loop.

An example of this can be found in figure 3.7.

If it holds for all elements of the computed set that the value loaded from the given memory address A is in a comparative relation to B, then the same relation is set between the value loaded from A and the value B in the join location also (in the example, it is the relation of being lesser set between the value loaded from `%iPtr` and `%n`).

This part is necessary to prove pointer validity in the cases when the index variable is assigned a completely different value, which is also a valid index.

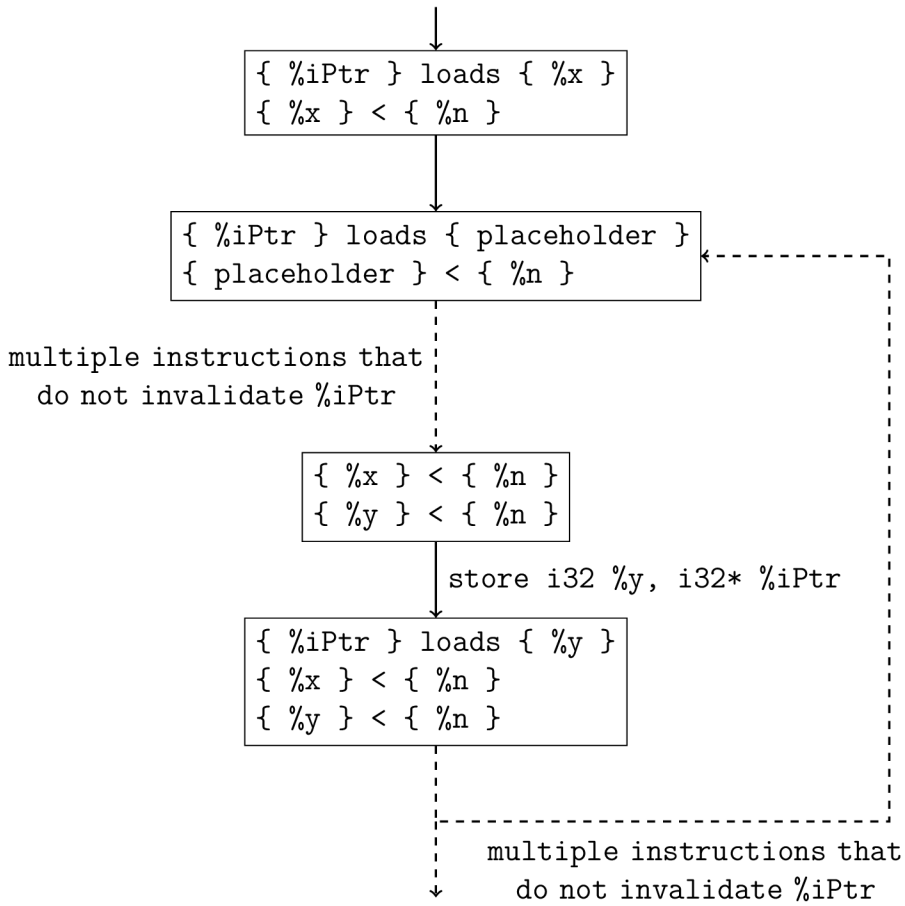

Figure 3.7: Intersecting relations from locations after load invalidation

3.3.3 An example

This section shows an example evaluation of the code in figure 1.1.

Figure 3.8 shows the state of value relations after the first analysis pass has been performed. There are several comparison relations generated by the *assumeBool* edges and the `add` of a constant. The `store` and `load` instructions generate multiple load relations. Keep in mind that at the time of processing the loop-join, there are no relations in its in-loop predecessor yet. At the same time, its tree predecessor contains no comparison relations and the only load relation it holds is invalidated in the loop's body. Therefore, there are no relations inferred for the join in the first pass.

There can be some relations inferred for the loop-join in the second pass. Since it can now be determined that the value at `%iPtr` is only incremented in the loop's body, its value can be set greater or equal to the value the address it is initialized with, as described in subsection 3.3.2. This new relations then propagates to all successors.

Since the stored relations have changed in the second pass, the code is processed one more time. Nevertheless, this last pass adds no new relations and so the analysis terminates. The final state of the value relations can be seen in figure 3.9.

Mark that at the location the array element pointer is computed, the index is known to be both greater or equal to zero and lesser to ten (and ten is the number of elements in the indexed array).

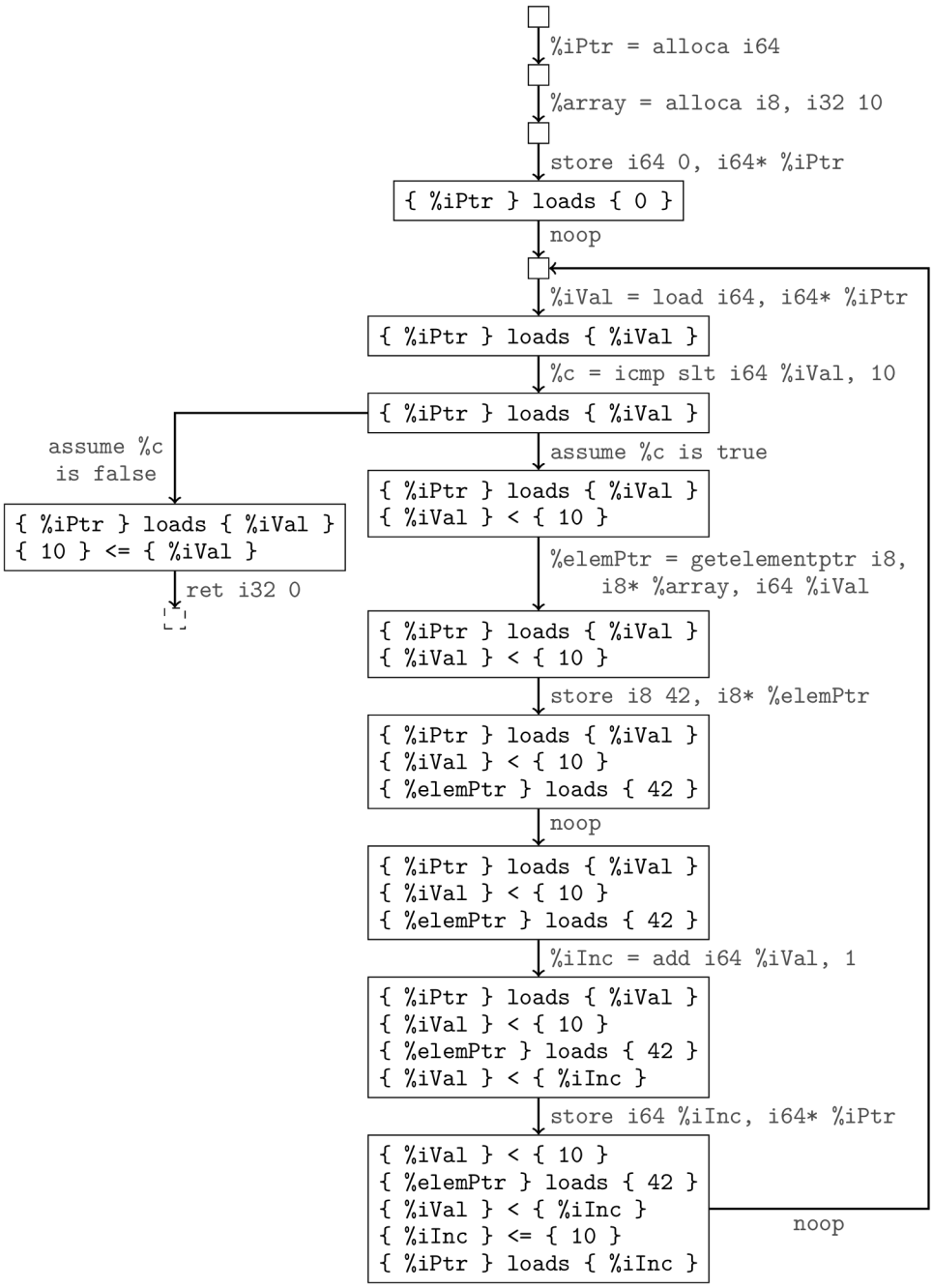

Figure 3.8: The relations after the first analysis pass

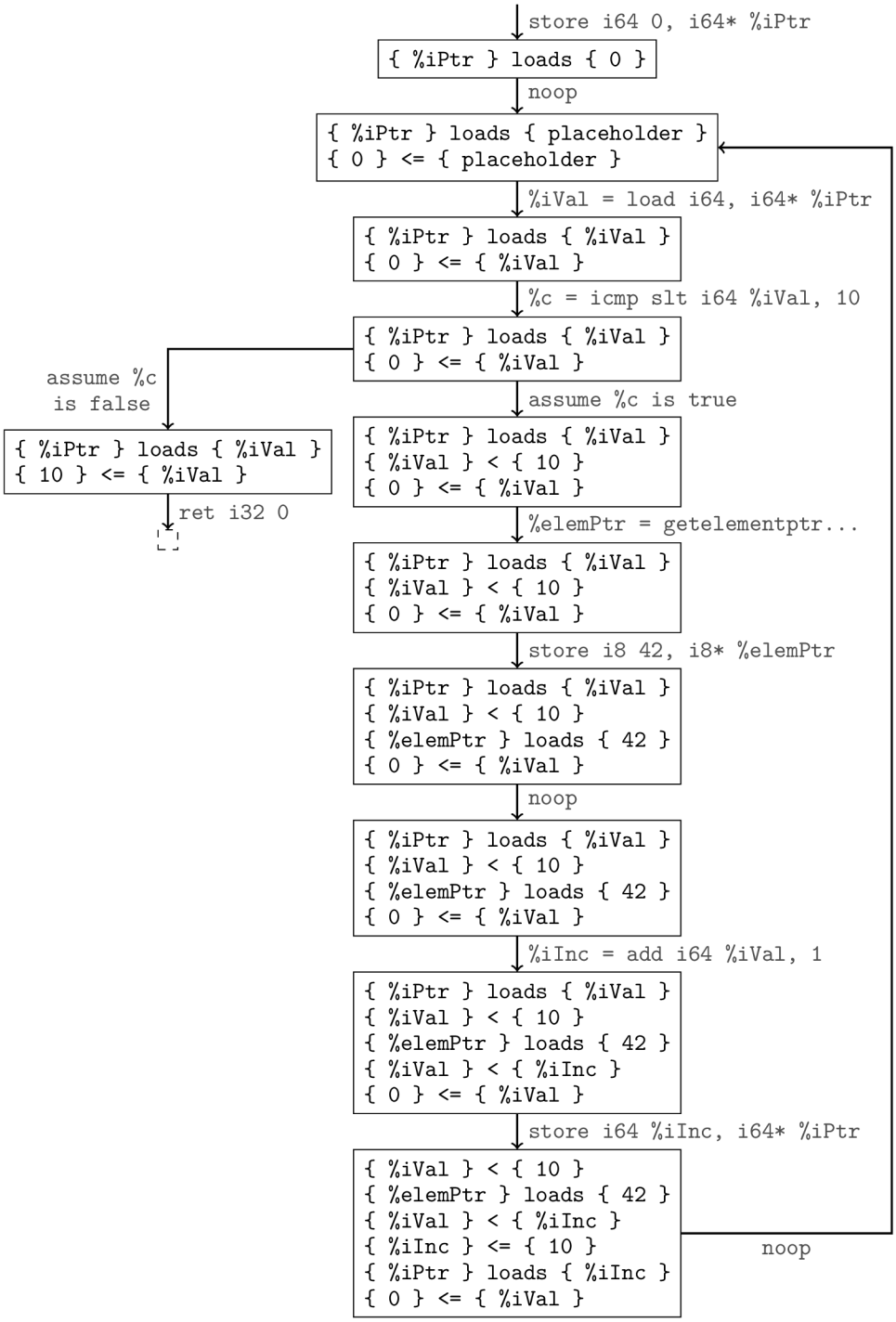

Figure 3.9: The relations after the code has been fully analyzed

3.4 Computing Allocated Areas

This last step of the analysis takes place only after the value relations analysis has been performed, because it requires the information of values' equality.

Firstly, the code is traversed and every `alloca` instructions or heap allocation function (`malloc`, `calloc`, `realloc`) commences the creation of a new `AllocatedArea` instance, as described in subsection 2.3.1. The allocated areas are stored in a fixed order.

Then, the code is passed once more, this time determining the validity of collected areas at different locations (an area is valid if it has been successfully allocated and has not been freed yet). To mark areas' validity, each location contains a bitvector of the same length as the number of areas in total.

When determining which areas are valid at a location, only the information from its immediate predecessors and the incoming edges is considered, similarly to the inferring of relations described in section 3.3. The method of determining again depends on the number of predecessors.

At the first location of each function (location with no predecessors), all areas are invalid.

If there is a single incoming edge, the valid areas from the predecessor are copied to the current location and then some bits may be changed in dependence to the information from the edge.

When an *instruction* edge carrying an `alloca` instruction is encountered, the corresponding area is marked valid – all stack allocations are considered unconditionally successful.

If an edge carries the intrinsic instruction `llvm.lifetime.end` and there is `alloca` equal to the ended value, then the area is marked invalid, because its scope has just ended. It is asserted that the area whose scope ends can always be determined.

When an edge is marked by a heap allocation call, the corresponding area is not marked valid straight away, because the allocation may have been unsuccessful. If `realloc` has been called, the original memory is also marked invalid, because the underlying memory may have been relocated.

The validation of areas allocated on heap happens only when an *assumeBool* edge is encountered. Memory is then marked valid if the

edge assumes that the resulting pointer is either non-equal or greater than zero. At the same time, when the compared pointer is the result of a `realloc` call and reallocated memory pointer is lesser or equal to zero, it means the original memory is still valid.

If a call to `free` occurs, the freed memory is marked invalid. If it cannot be determined which area was freed, all of them are invalidated, because any of them might have been the freed one. The same happens when `realloc` is called and the reallocated area cannot be determined.

When there are multiple incoming edges, the area is marked valid only if it is valid in all predecessors. If the validity bitvector has not been computed for a parent yet, all bits are assumed false with an only exception – that is if the edge to the parent is a back edge. In such a case, a forward check for areas invalidated inside the loop is made using the instruction set precomputed earlier (subsection 3.2.2).

It suffices to make a single such pass through the code to determine the validity of allocated areas in all locations.

4 The Use of the Analysis in Symbiotic

In this chapter I describe the outcome of the value relations analysis outlined in the previous chapter. The functionality is contained in the `ValueRelationsPlugin` class. First, I describe the plugin's interface and the context in which it is used. Then, I focus on the inner workings of the plugin.

4.1 Plugin's Broader Context

To set the plugin into its context, I will briefly describe the phases of code verification as done by Symbiotic [1].

Symbiotic is a tool for verifying various properties of sequential C programs. The focus of this thesis is on the pointer validity, but the tool can also check for the violation of assertions or insufficient memory management (memory leaks, double frees). The verification is not performed directly on the C code but on the LLVM intermediate representation.

In all simplicity, the verification has three distinct phases.

Instrumentation In this phase, marks are inserted into the code to indicate that given instruction is possibly unsafe in regard to the checked property. Analyses (like the one described in this thesis) are performed to decide which instructions should be instrumented.

Slicing Here some instructions of the instrumented code are removed to ease the next phase. The instructions left are only the ones that may be relevant in reaching an instruction which might violate the checked property.

Symbolic execution The sliced code is executed by the KLEE virtual symbolic machine to check for the reachability of a erroneous location.

As mentioned before, my analysis focuses on the pointer validity, and it serves to mark pointers obtained by indexing an array as safe. It is integrated in the following way: the `ValueRelationsPlugin` is queried

with a pointer and a size and responds with a string signifying whether a read or a write of *size* bytes from the pointer is valid or whether it might be invalid.

4.2 Proving Pointers' Validity

At the moment, the goal of this plugin is to mark safe all pointers that access an array element and have been computed using an index whose value ranges from zero to the size of the array minus one. A basic example of such a code can be seen in figure 1.1. The value relations in the code are also analyzed in subsection 3.3.3.

At the moment, the plugin deals only with pointers into an array. In the LLVM IR, these pointers are computed by the `getelementptr` instruction.

4.2.1 Validity for a single `ValueRelations` instance

Given an instance of `ValueRelations`, a vector of valid `AllocatedAreas` and a pointer resulting from the `getelementptr` instruction, the validity of the pointer can be checked without much complication.

First, the accessed `AllocatedArea` is obtained using the base address of the `getelementptr`. If no `AllocatedArea` is found for that address, then the accessibility nor the allocated size can be determined and the access is marked potentially invalid.

Second, if matching `AllocatedArea` is found, it is checked whether the area is currently valid to access and if not, the pointer is again marked potentially invalid.

If these two checks both succeed, the number of elements in the `AllocatedArea` is obtained, be it a constant or a variable, and also the size of each element. Then the part of validation which uses the `ValueRelations` may proceed.

In this part, two relations are looked for: whether the index of the `getelementptr`' is greater or equal to zero and at the same time lesser than the number of allocated elements. The pointer is judged valid only if the size read is not greater than the size of the array's elements.

The only catch here is that the memory might have been allocated in differently sized elements than in which it is accessed (e.g., the

memory is accessed as ten four-byte elements, but it was allocated as forty bytes). Still, for most cases, this is handled as well: if the boundary check for a given count and element size fails and the allocated size is a result of multiplication by a constant, then both the element count and element size are altered accordingly, and the check is attempted again.

4.2.2 Considering pointer's multiple contexts

It is desirable to be able to validate even pointers that are dependent on values passed to a function as parameters (as in figure 4.1). To validate such a pointer, it does not suffice to consider relations which hold at the location where the pointer is computed. The relations that need to be taken into account as well are the relations valid at the site, from which the function containing the `getelementptr` is called.

```
void setAscending(uint8_t* array, unsigned size) {
   for (unsigned i = 0; i < size; i++)
       array[i] = i % 256;
}
```

Figure 4.1: A function in which the memory accesses are valid only if `size` is lesser or equal to the number of bytes allocated for `array`

A pointer may be deemed valid only if it is valid in all contexts in which it is called.

In figure 4.2 the element pointer in function `setAscending` is valid through all iterations in the first call, but addresses out-of-bounds memory when called the second time.

```
int main() {
   uint8_t array[10];
   setAscending(array, 10);
   setAscending(array, 20);
}
```

Figure 4.2: A function called twice with different parameters

To correctly determine a pointer's validity, relations added by each call must be examined separately. So if there are any `CallRelations` stored for the function the `getelementptr` instruction is a part of, then these are one by one merged into the relations holding at the `getelementptr`'s location. For each `ValueRelations` resulting from one such merge, the pointer's validity is examined, as described in the previous section. Should a conflicting relation be added by the merge, the merge stops and the pointer is considered valid for this combination of relations, because the conflict suggests, that the `getelementptr`'s location is unreachable with parameters and relations from that particular call location.

If the `setAscending` function from example 4.1 was modified in the way suggested in 4.3, then together with the `main` from 4.2 it would form a valid program, since the memory access location is unreachable upon the second call. The conflict that emerges is that at the `getelementptr`'s location, `size` must be lesser or equal to 10 and in the `CallRelation` there is equality between the `size` formal parameter and the value 20 and these relations are irreconcilable.

```
void setAscending(uint8_t* array, unsigned size) {
   if (size > 10) return;

   for (unsigned i = 0; i < size; i++)
       array[i] = i % 256;
}
```

Figure 4.3: A function in which the memory accesses are valid only if `size` is lesser or equal to the number of bytes allocated for `array`

As of yet, only relations valid at location where the pointer is computed, and the relations from each call of the location's function are taken into account. It would of course be possible to consider also relations higher up the call graph, but the total number of considered combinations would grow exponentially and so I decided to limit that and expand that limit should such a need arise.

5 The Experiments

In this chapter, I evaluate the results Symbiotic when integrated with the plugin as opposed to its performance without it. The experiments are run on a set of benchmarks from SV-COMP's *MemSafety* category.

The benchmarks are C programs, that are either valid, or contain some sort of memory violation (invalid pointer dereference, double free, etc.). Symbiotic's answer is *true* if it is able to determine that no error location is reachable, in the opposite case it answers *false* (with the specification of the violation). If Symbiotic cannot determine the reachability of an error location, the answer is *unknown*. Symbiotic may also timeout, if the execution time exceeds the time limit, stop because it reached the memory limit, or terminate with an error.

Two versions of Symbiotic were evaluated. First one was the version augmented with the described analysis and the plugin for validation of in-loop array access (the version is denoted as *augmented* from now on). Second version was one without the augmentations (denoted as *unaugmented*).

The code of the unaugmented Symbiotic can be found in its GitHub repository [1]. The code for the augmented version differs only in `dg` and `sbt-instrumentation` and was taken from my forks of the original repositories[2].

Most specifically, the following component versions were used: `symbiotic:e784a5ed`, `sbt-slicer:ce747eca`, `klee:8340acee`.

In the unaugmented version, the remaining components were: `dg:e89761ff` and `sbt-instrumentation:0e5f3f91`. The augmented version used `dg:dd129855` and `sbt-instrumentation:177dc711`.

The `dg` and `sbt-instrumentation` repositories of the augmented version are a part of this thesis' attachment.

In total, Symbiotic run on 558 benchmarks, of which 350 were safe and 208 exhibited some sort of memory violation. The benchmarks can be found in the `sv-benchmarks` repository[3], identified by `6c4d8bc9`.

---

1. `https://github.com/staticafi/symbiotic`
2. `https://github.com/GiraffeReversed/dg`, `https://github.com/GiraffeReversed/sbt-instrumentation`
3. `https://github.com/sosy-lab/sv-benchmarks/tree/master/c`

The time and memory-consumption measurements were taken on a machine with *Intel Core i7-8700 CPU* at the frequency of 3.20GHz. Each benchmark was provided with 180 seconds of CPU time and 2 GB of memory. The experiments were run using the `benchexec` [8] tool, version 2.2. The tool serves for the automation of benchmarking and reliable measurement of consumed resources.

The table 5.1 summarizes the result of both versions. There are eleven benchmarks that are undecided in the unaugmented version, but the augmented version is able to decide them. At the same time, the plugin does not add any new incorrect responses.

The undecided benchmarks are those, for which Symbiotic responded with *unknown*, timed out, reached the memory limit, or terminated with an error.

Table 5.1: The cumulative results of both experiment runs

	unaugmented	*augmented*
correct true	155	166
correct false	159	159
incorrect true	3	3
incorrect false	1	1
undecided	240	229

Five of the newly decided benchmarks were implementations of different well-known algorithms (two implementations of bubble-sort, an insertion-sort, a selection-sort, and an algorithm for finding the longest increasing subsequence), which shows that the plugin can deal with non-trivial programs with nested loops as well.

5.1 The Effect on Instrumentation and Slicing

More effects of the added plugin can be seen upon closer examination of the instrumentation phase.

As described in section 4.1, the plugin participates in instrumenting the verified code. If it marks an instruction safe, then this instruction is not instrumented, thus allowing for a larger portion of the verified code to be sliced away. KLEE can then process the slice faster, sometimes fast enough for the benchmark to be decided in the time limit.

When comparing the number of instrumented instructions, there are seventy two benchmarks where the number decreased in the augmented version. The fact that the number is not higher is not surprising, since the majority of the benchmarks in the *MemSafety* category deals with the safety of heap memory allocation and dynamic data structures, not with array accesses.

Out of these seventy two benchmarks, forty nine are left undecided by the unaugmented version of Symbiotic. Eleven of those undecided benchmarks are newly decided in the augmented version.

The remaining thirty eight benchmarks share a common pattern. First, an array is allocated. Then it is set with nondeterministic values in loop using an index to access specific elements, this access is marked safe by the plugin. After that, the array is processed in some way, element by element.

Nevertheless in this last part, the array is not accessed using an index whose range would be limited by the array's size. Most often, the array is zero-terminated and then iterated until the zero is reached. In other cases, the array pointer itself is incremented a fixed number of times, where the number of times depends on the size of the array. As of yet, the plugin cannot validate these types of accesses and therefore they are not decided in time.

The number of instrumented instructions decreased by 2.4 on average in the seventy two benchmarks, with the average being 6.2 when regarding only the eleven newly decided benchmarks. The maximum decrease was twelve and was achieved by one of the newly decided benchmarks.

As for the number of instructions after slicing, there were twenty six benchmarks, both decided and undecided by the unaugmented version, in which the number of instructions in the sliced code decreased in the augmented version as a result of less instructions being instrumented. The difference can be seen in figure 5.1.

Out of the twenty six benchmarks, in twelve the number of instructions in the sliced code dropped to one, which means that all instructions that could cause some memory violation have been marked safe by some analysis. Two of these twelve benchmarks are decided in the unaugmented version already, remaining ten benchmarks belong to the newly decided benchmarks. For the last newly decided benchmark,

the number of instructions after slicing did decreased, but not all the way to one.

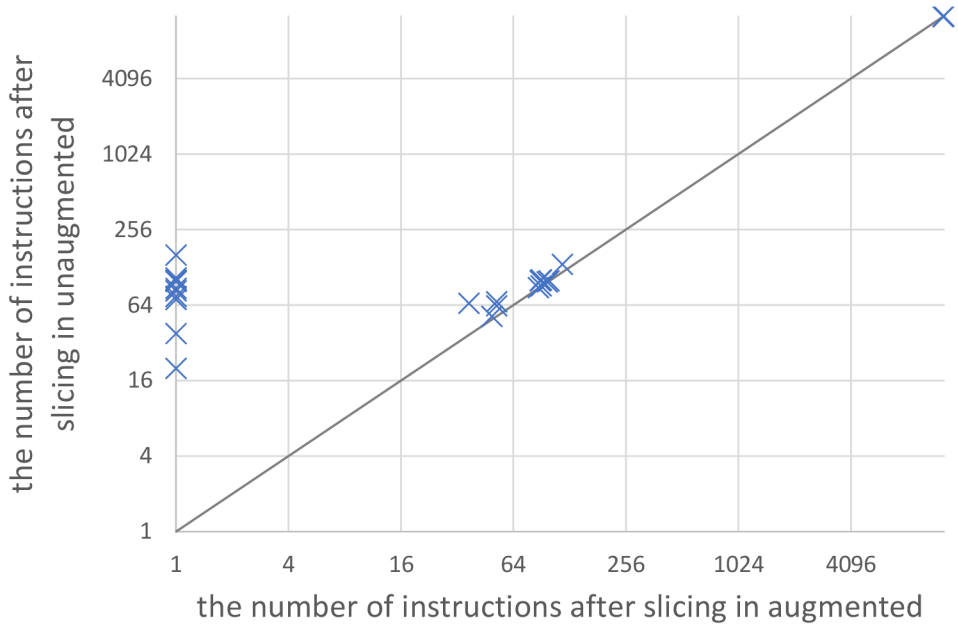

Figure 5.1: A scatter plot of the number of instructions after slicing (the less the better)

5.2 The Effect on Resource Consumption

As for the consumption of resources, it increased in the augmented version when compared with the unaugmented one. The total verification time increased by approximately 100 ms on average (2 %), 6 ms median, and the memory consumption increased by 3.4 MB (3 %) on average, 25 kB median.

To determine the increment, I decided to disregard the CPU time and memory consumption of those benchmarks that either timed out or terminated because they reached the memory limit in both versions. That is because when measuring the verification time, the increase cannot be measured on benchmarks which reached they limit in the unaugmented version also and at the same time, the plugin also consumes memory and so the memory limit may be reached

sooner even though the plugin did not bring any improvement in the verification time. Analogous reasoning holds for evaluating the memory consumption too. That left 412 benchmarks.

The time of instrumentation phase increased on average by 80 ms (13 %), 6 ms median, in the augmented version on average among *all* but one benchmark, which times out on instrumentation.

Conclusion

The goal of this thesis was to design and implement a method that would be able to decide whether a given in-loop access of an array element via an index is valid or not. This goal was to be achieved by implementing a value relations analysis of code in LLVM IR.

In the thesis, I describe the structures and the process of the implemented analysis in detail. I justify different decisions made to collect valid relations necessary to prove pointers' validity. I outline the actual process of the array access validation as well.

The validation process has been implemented in the form of an instrumentation plugin for the verification tool Symbiotic.

I evaluated the results of multiple benchmarks from the *MemSafety* category of SV-COMP. It shows that the validity of several previously undecided benchmarks can be decided thanks to the newly employed analysis.

However, the experiments also show, that the current plugin could be improved by the ability to validate array accesses done by other means than using an index whose value is limited by the array's size. Fortunately, the value relations analysis is well-suited to support such an advance too.

And what more, the value relations analysis does not have to be used only to check for memory violations, but also to decide whether various assertions about value relations might fail, thus improving Symbiotic's verification abilities even further.

Bibliography

1. *Symbiotic* [online]. 2018–2020 [visited on 2020-04-22]. Available from: http://staticafi.github.io/symbiotic/.
2. *The LLVM Compiler Infrastructure* [online]. LLVM Project, 2003–2020 [visited on 2020-04-16]. Available from: https://llvm.org/.
3. *LLVM Language Reference Manual* [online]. LLVM Project, 2003–2020 [visited on 2020-04-16]. Available from: https://llvm.org/docs/LangRef.html.
4. ALLEN, Frances E. Control flow analysis. *ACM SIGPLAN Notices* [online]. 1970, vol. 5, no. 7 [visited on 2020-04-19]. ISSN 03621340. Available from: http://portal.acm.org/citation.cfm?doid=390013.808479.
5. ROSEN, B. K.; WEGMAN, M. N.; ZADECK, F. K. Global value numbers and redundant computations. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88* [online]. New York, New York, USA: ACM Press, 1988, pp. 12–27 [visited on 2020-04-19]. ISBN 0897912527. Available from: http://portal.acm.org/citation.cfm?doid=73560.73562.
6. CORMEN, Tomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms* [online]. 3rd ed. Cambridge (Massachusetts): The MIT Press, 2009 [visited on 2020-04-20]. ISBN 978-0-262-03384-8. Available from: https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf.
7. *Call for Participation – Procedure* [online]. European Joint Conferences on Theory & Practice of Software, 2020 [visited on 2020-05-18]. Available from: https://sv-comp.sosy-lab.org/2020/rules.php.
8. WENDLER, Philipp; BEYER, Dirk. *sosy-lab/benchexec: Release 2.2*. Zenodo, 2019. Version 2.2. Available from DOI: 10.5281/zenodo.3463154.

A Archive Structure and Compilation

The implementation of the analysis and the plugin described in this thesis is a part of the attachment. The source code of the analysis is contained in the folder `dg/include/dg/llvm/ValueRelations`, the source code for the plugin can be found in `sbt-instrumentation/analyses` in files `value_relations_plugin.*`. The code is written in the C++ programming language.

The code is the exact same as the code used for the respective components of the augmented version of Symbiotic as described in chapter 5.

A.1 Compiling and Running the Analysis

It is possible to run the analysis on a code using the `llvm-vr-dump` tool.

To compile the tool, it is necessary to have the following tools and libraries installed:

`CMake` minimal version 2.8

`LLVM` preferably 6.0 or 8.0.1, since the code has not been tested with other versions

`zlib`

To build the `llvm-vr-dump` tool, it should suffice to run `cmake .` in the `dg` folder and then `make llvm-vr-dump`. The compiled binary can be then found in the `tools` folder.

With `llvm-vr-dump` successfully built, it is possible to analyze LLVM bitcode (`.bc` files) or the human-readable form (`.ll`). To analyze a code, run `llvm-vr-dump <path-to-code>`. When run without options, the tool analyzes the code and prints the duration of the analysis. Use the `-dot` option if you want the code representation with collected relations to be dumped to the standard output in the `dot` format. It is also possible to specify `-max-iter=<max-number-of-passes>` to pass the code a fixed number number of times. If the number is zero, then only the tool only builds the graph and performs the analysis of its structure.

The following snippet shows how to build the tool on a clean instance of Ubuntu 18.04.

```
# Example of build for clean Ubuntu 18.04

# If using Docker, run:
# docker run -it ubuntu:18.04

apt-get update
apt-get -y install cmake llvm zlib1g-dev clang

cd /tmp
# Manual step: copy the archive contents here
cd value-relations-thesis/dg
cmake .
make llvm-vr-dump

# Example of running the analysis of bubblesort
clang -c -emit-llvm \
   ../example/bubblesort-alloca-2.c \
   -o bubblesort.bc
./tools/llvm-vr-dump bubblesort.bc
```

A.2 Compiling and Running the Plugin as a Part of Symbiotic

To see the results of Symbiotic augmented with the new plugin, it is necessary to build Symbiotic first. To do this, clone the repository from `https://github.com/GiraffeReversed/symbiotic.git`, then follow the installation guide[1]. For more information, refer to the `README.md` file in the archive.

After the build has successfully finished, run Symbiotic with the `--prp=memsafety` option to check for memory safety violations. To list the remaining options, run Symbiotic with the `--help` option.

1. `https://github.com/GiraffeReversed/symbiotic/blob/master/README.md`

A.3 Provided Examples

There are several examples of programs accessing an array with a ranged index in the archive. There is the code from figure 1.1 and also the C code of the eleven benchmarks Symbiotic is able to decide with the new plugin.

To compile another code to the LLVM IR, the easiest way is to use the `clang` compiler and run `clang -c -emit-llvm <path-to-code>` to compile to bitcode, and `clang -S -emit-llvm <path-to-code>` for the human-readable representation.