École Polytechnique Fédérale de Lausanne

Improving Clang Static Analyzer's False-positive and False-negative Rates

by Jordan Cosme & Gimalac Pierre

Semester Project Report

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Lee Gwangmu
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 6, 2023

Contents

Chapter 1

Introduction

In a never ending world of use and creation of software, the number of bugs keeps increasing and increasing. The news about security flaws are a common recurrence [1] [2]. Software tools have been developed to mitigate some of them, such as fuzzers and sanitizers. However, bugs can also be found at compilation time, which is interesting because it can speed up the development process of a software.

Compiler warnings can catch and find bugs early in the development cycle of a software. For example warnings such as *using the result of an assignment as a condition without parentheses*, *shift count is negative* or *comparison between signed and unsigned integer expressions* can help the programmer in finding source of potential errors and bugs in the program, which might be hard to detect otherwise.

However, compiler warnings are limited in their use and effects. Indeed, they do not take into account the full logic of a program, they usually only operate on the syntax tree, and are often limited to a specific compilation unit. Static analyzers have been developed to improve this. Those are tools which can find bugs triggered by certain paths of execution, they know about the specific behavior of some function or module interface (for example knowing that you should not use a value given to the `free` function in C, and thus detecting paths which trigger a *use-after-free*), and basically any kind of bug which is costly or inaccurate to detect, and thus would not be implemented in a compiler.

In this work, we will use, test, analyze and improve the *Clang Static Analyzer*, the static analyzer from the clang tool suite.

According to its documentation [3], *"The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs. It implements path-sensitive, inter-procedural analysis based on symbolic execution technique"*.

The goal of the tool is to identify executions of the code which are triggering a bug with

certainty, without false-positives (possibly at the cost of more false-negatives). It's architecture is very modular, it has a core part which tracks values, paths, environments, etc, and then a set of plugins (called checkers) which each try to detect a specific bug (eg. use-after-free or divide-by-zero). See section 2.1 for more details about the inner workings of the static analyzer.

The goal of this semester project was to understand the inner workings of the Clang Static Analyzer (hereafter CSA), discover and analyze the origins of its false positives and try to fix some of them.

We decided to first write custom checkers to start understanding how the tool works and is used, before trying to generate false positives and eventually fixing some of them.

After this we went on to generate false positives. We tried both an automatic and a manual approach. The automatic approach was interesting but had important disadvantages which prevented us from actually finding bugs. The manual approach also has some disadvantages, but proved to be more efficient at finding false positives than the automatic approach, in particular since we did not have to try to understand why a bug was reported and whether it was actually a false positive. Section 3.1 explains this process and the difficulties we met.

The subsequent sections of chapter 3 show seven different kinds of false positives[1], and try to analyze and explain them. Note that most of those false positives can also be extended and changed to find more false positives of the same kind. We analyzed the false positives to find out whether the problem came from the checker reporting it or directly from the core part of the static analyzer, and we realized that most of the problems came directly from the limitation of CSA and some of its design choices.

We later spent time to understand how the Clang Static Analyzer works so that we could try to make changes to fix some of the false positives we found. Our goal was to make a somewhat minimal number of changes, as we did not want to rewrite big parts of CSA. Furthermore, Clang has goals of its own, and it is important to abide by them, for example a direct citation from one of Clang pages: *A major focus of our work on clang is to make it fast, light and scalable* [4]. We explain our findings in chapter 2 so that the reader can understand the following chapters more easily.

We worked with *LLVM* 15[2] and we do not guarantee that our work is compatible with any other version than 15. With a different version, the inner workings of CSA might be different, the false positives might not be reported, and our fix might not be portable. See appendix A to see how to build and install the correct version of Clang in order to run our code.

Several research work have been performed on static analyzers, some to compare the efficiency of various such tools [5], some to improve analyzers like we did [6][7], and others simply

[1] Only six really, since one is due to the code being undefined behavior, but we decided to keep it as the example is interesting in itself and prompted us to look further in the inner workings of the memory model

[2] Which at the time of writing this is the last published version, while the last version available on the Arch repository is 14 and the development version is 16

to explain how to use such tools[8].

Chapter 2

Background

2.1 Clang Static Analyzer

The Clang Static Analyzer is basically a program simulator, it mimics the execution of a program using symbolic execution. It means that the analyzer will track the values of the program by variables using abstract values (called symbolic) which can take some actual values, be compared, etc. The tool will split the execution into states with an environment and a store, and then compute the transitions from one state to another to track the different paths that can be taken while executing.

CSA keeps the state of the program, expressions and values of the statements in variables of the immutable class `ProgramState`. More precisely, a state contains constraints and symbolic values of expressions and memory regions.

The `ProgramState` and each point of the possible paths of execution of the program are assembled in what is called the *exploded graph*. Each node of the graph corresponds to a point of the program and contains the `ProgramState` at the current point. It can be displayed cleanly for debug purpose.

The static analyzer starts at the entry point of a program and simulates each line of a program (calls, pre-statement, post-statement,...), using the `visit` function of the *ExprEngine* class using the visitor pattern. When CSA reaches a branch, it will check the constraints in the `ProgramState` to find which of the paths can be taken. If the condition is impossible, then the Static Analyzer simply ignores it and removes it from the graph. Thus, it will never execute it. Furthermore, each state of the exploded graph are cached and can be reused in other paths of the execution.

If the analysis of the current point changes the `ProgramState`, then a new node is created.

A bug is found when some conditions are triggered. Those conditions are defined by what

LLVM has defined as a checker. At this effect, the analyzer uses checkers to find bugs. All checkers are stored and run when appropriate by the `CheckerManager` (cf. section 2.2).

2.1.1 Memory model

The hardest part of modelling and running a C program is keeping track of the possibles values variables and regions can contain. In order to model all of the possible way the memory of a program can be organized, the Clang Static Analyzer uses a very modular set of classes as well as immutable maps to store values.

Two documents are very helpful to understand how CSA works, in particular its models memory, one is a foundational article from 2012 which explains the choice of memory model implemented in the static analyzer, along with its perks and drawbacks compared to other possible implementations[9], and the other is a guide on how to use CSA written by one of the main developers in 2016[10]. There are also two presentations which introduce how to use CSA and a brief part about its inner workings[11][12]. We will give an overall explanation on how the analyzer works in the following sections, in particular how it models values, memory and executions, as well as what the main classes of its implementation are.

Regions

Every little piece of information about the memory can be useful to make the analysis more accurate, it is thus represented using an intricate set of classes which contain all needed information.

The abstract class `MemRegion` is the base of all memory types, it is then extended with useful information[1]: the different kind of memory spaces (stack, heap, constants, global variables, etc) are represented by the `MemSpaceRegion` class[2], other types of memory regions are actually `SubRegion`s of the "basic" kinds of memory regions, in particular subregions can be typed (`TypedRegion`, `TypedValueRegion`), they can be a field of a struct (`FieldRegion`) or an element of an array (`ElementRegion`), there are also specific subclasses for regions representing variables (`VarRegions`, `DeclRegion`, etc), and more.

All the subregions have reference to their parent region in order to track as precisely as possible how everything is located and address the aliasing issue in C as well as possible. The `MemRegionManager` is used to create and fetch regions during analysis.

[1] We omit all C++ specific classes here for simplicity

[2] And in particular its subclasses CodeSpaceRegion, StaticGlobalSpaceRegion, NonStaticGlobalSpaceRegion, GlobalSystemSpaceRegion, GlobalImmutableSpaceRegion, GlobalInternalSpaceRegion, HeapSpaceRegion, UnknownSpaceRegion, StackSpaceRegion, etc

Symbolic values

As the Clang Static Analyzer can not know for sure the value of all variables during execution, it uses symbolic values to represent them, which can be compared and have the same or different values, depending on constraints stored on them.

CSA uses the class `SValBuilder` and its derivates to create symbolic values. These values can be L-values (memory location) of class `Loc` or R-values (actual "values", eg. integers) of class `NonLoc`, they can be undefined (`UndefinedVal`, eg. uninitialized variables), unknown (`UnknownVal`, eg. a parameter of a function or the result of some operation) or known (`DefinedVal` and `KnownVal`), have different types, sizes, signedness, etc.

Store

The actual values associated to regions are stored in the `Store`, an immutable map associated to each `ProgramState` which allows to store and retrieve values for each memory location. The `Store` is actually an opaque type, it has to be handled using the `StoreManager` associated to the state.

Whatever the actual implementation, it groups saved values by "clusters" (memory range which are not supposed to intersect, eg. each variable is a cluster of its own, an array is a cluster, etc), and then in each cluster several values can be saved, for example the default value of an array (this avoids having to initialize all indices when reading a statement such as `int data[10] = -1`) and values associated to indices.

All load operations go to the store to see what the value associated to the location is. When no value was stored, an `UnknownVal` is returned.

Listing 2.1: Example of a Store

```
"store": { "pointer": "0x56246457173a", "items": [
  { "cluster": "data", "pointer": "0x562464570e60", "items": [
    { "kind": "Default", "offset": 0, "value": "0 S32b" },
    { "kind": "Direct", "offset": 0, "value": "-1 S32b" }
  ]},
  { "cluster": "GlobalSystemSpaceRegion", "pointer": "0x5624645716b8",
    "items": [
      { "kind": "Default", "offset": 0,
        "value": "conj_$0{int, LC1, S17768, #1}" }
    ]
  }
]}
```

A particularly interesting example on how the store works is for arrays.
As explained above, the array will be a single cluster, there can be a default value if the array was declared with one, and then each value written is stored as a "Direct" binding with the associated offset inside the array.
When trying to read an index in an array, if the index appears in the store then all is well, otherwise some specific behavior is implemented: if the index is a constant integer (or equivalently a variable whose value is known with certainty) then an unknown symbolic value is generated and added to the store at this index, so that multiple readings always return the same (unknown) value. If on the other hand the index can have several values, then a fresh unknown symbolic value is generated at each read and the store is left untouched.

Environment

The `Environment` is similar to the `Store` but its purpose is to associate values to statement instead of memory locations (the keys of the map are R-values instead of L-values). `LocationContext` are also stored in the key since the same statement can have different values depending on the context (loops, recursive functions, etc). Similarly to the `Store`, the `Environment` should be modified using the `EnvironmentManager` in the `State`.

Every time a statement is computed, the resulting value is stored in the `Environment` along with some information about the statement itself (an identifier and the actual piece of code), which is very convenient for debug purpose or to display the exploded graph.

Listing 2.2: Example of an Environment

```
"environment": { "pointer": "0x55c067bf0af0", "items": [
  { "lctx_id": 1, "location_context": "#0 Call", "calling": "main",
    "location": null, "items": [
    { "stmt_id": 17924, "pretty": "data[idx]",
      "value": "&Element{data,(conj_$1{int, LC1, S17768, #1}) % 2,int}"
    },
    { "stmt_id": 17930, "pretty": "val1", "value": "&val1" },
    { "stmt_id": 17934, "pretty": "val2", "value": "&val2" },
    { "stmt_id": 17938, "pretty": "val1",
      "value": "conj_$2{int, LC1, S17768, #1}" },
    { "stmt_id": 17941, "pretty": "val2",
      "value": "conj_$3{int, LC1, S17768, #1}" },
    { "stmt_id": 17944, "pretty": "val1 != val2", "value": "0 S32b" }
  ]}
]}
```

The `Environment` is often flushed because values associated to statements can not really be re-used across statements (values can be different even if the statement is the same). So-called

"dead bindings" are then removed to avoid over-using the memory.

The environment can contain apparently similar entries, but which can actually represent conceptually different values, for example in the example above there are two entries for the statement `val1` but one is for the L-value associated to the variable `val1` while the other is the actual value associated to it (which in this case is an `UnknownVal`).

2.1.2 Ranges

The calculation of binary operators such as plus and minus are lazily evaluated and a symbolic value is created instead, such as `SymIntExpr`. When CSA reaches a branch, it will call the function `evalEagerlyAssumeBinOpBifurcation`. When the static analyzer reaches a branch, the actual range calculations is evaluated with the `assume` function which will in turn call `assumeSymRel` to calculate a constraint between a symbolic expression and a concrete integer. In other cases it will call other `assumeSym*` functions.

The `APSInt` class models any size signed integer value, while the `APInt` only represents the unsigned integers. Both classes contains an implementation for all typical operators and a magic function[3] which allows us to get the maximum and minimum value. Though, `APSInt` does not have a shift operator for an `APInt` argument.

To calculate the range, the static analyzer will solve the typical equations such as $a + 1 < 2$ which would become $a < 1$. The function `computeAdjustment` from the `RangedConstraintManager` class obtains the `APSInt` constant $+1$. The function `getRHS` of the `SymIntExpr` class gets the constant 2.

From this, two ranges will be created: one where the expression is evaluated to true. In our example, this is done with a call to `assumeSymLT` which will call `getSymLTRange`. The function `getSymLTRange` calculates both the upper and lower bound of the less than range. The "false" range is then calculated similarly but this time with a call to `assumeSymGE` and `getSymGERange`. The static analyzer will then continue with the corresponding ranges for each two cases. If the upper bound is smaller than the lower bound then a wraparound range is created. I.e. two ranges are created: $[-INFINITY, UPPER\_BOUND]$ and $[LOWER\_BOUND, INFINITY]$[4].

To compute more complicated ranges such as `if(a > 1 && a < 10) {}`, the static analyzer will split them into two different cases. First of all, it will evaluate the case $a > 1$.

A separate data structure (called `RangeSet`) stores both bounds. This data structure is responsible for adding ranges, adding points to the range and for other standard range calculations such

[3] The magic functions are `getMaxValue` and `getMinValue`. It can also be found with the `APSIntType` (which represents a record of `APSInt`)

[4] $INFINITY$ is used as a placeholder for the maximum value for a sized integer (`int`, `short`, etc) and its minus version for the minimum value

as intersection and union between two different ranges. The false case is executed as previously. The static analyzer will then calculate both the true and the false ranges for $a < 10$. When adding those to the data structure, the ranges are then updated accordingly, so there is the calculation of an intersection between both ranges of both conditions. Then both the true and the false cases would be executed similarly as before.

So to summarize, the static analyzer will execute the condition similarly to the following code

```
if(a > 1) {
    if(a < 10) {
        // If block
    }
}
```

2.2 Checkers

We will first define how checkers work, can be implemented and run, and then develop two examples with the *taint* and *div-by-zero* checkers.

2.2.1 Definition and use

The logic to detect a bug is left to what CSA call a checker. A checker is a visitor at the AST level which finds bug according to some logic. Checkers extend various classes depending on where they want to operate, for example before and after a statement with `checkPreStmt` and `checkPostStmt`, before and after a call to a function with `checkPreCall` and `checkPostCall`, on the binding of a value with `checkBind`, etc. There are many other possibilities, for example at the beginning of a function, or before a binary operator (see section 2.2.3 for an example). To implement a checker one has to derive it from the abstract interface `Checker` and extends the desired classes.

Thus, checkers are extremely powerful because someone with no knowledge of the Clang Static Analyzer can write one and load it into CSA and use it. Currently, there are more than 50 types of checkers already implemented into the static analyzer, such as divide-by-zero, use-after-free, null-dereferences, etc.

If we want to use a custom checker with the Static Analyzer. We use

```
$ clang -fsyntax-only -fplugin=lib/custom.so -Xclang -analyze\
                -Xclang -analyzer-checker=demo.custom test.cpp
```

to run a custom checker, `demo.custom`, defined in a compiled `C++` static library called `custom.so` on a file `test.cpp`.

One can run a specific internal CSA checker (in this case divide by zero) using,

```
$ llvm-project/build/bin/clang -fsyntax-only -Xclang -analyze\
        -Xclang -analyzer-checker=core.DivideZero test.cpp
```

To generate the exploded graph, add to the previous command

```
-Xclang -analyzer-dump-egraph=graph.dot
```

And to convert the generated exploded graph `graph.dot` to an html page use the `LLVM` tool graph re-writer.

```
$ llvm-project/clang/utils/analyzer/exploded-graph-rewriter.py\
                                                    graph.dot
```

2.2.2 Taint analysis

Taint is a special checker which tracks variables, arrays, structs (or just some fields), and even entire memory regions whose value were written directly by the user.

Since the goal of CSA is to only report bugs which are certain (to limit the number of false positives), lots of potential bugs are not reported ; however if the user is responsible for the value of some variables then some bugs that are only possible (not certain) should still be reported, considering that it could be used for an attack or accidentally triggering the bug (cf. section 2.2.3 for a good example of this).

The checker provides functions to mark/un-mark some location as tainted, as well as check whether a location is tainted, so that all other checkers can collaborate to report bugs more accurately.

The checker is not part of the default checker, it has to be enabled by using the following arguments

```
-Xanalyzer -analyzer-checker=alpha.security.taint
```

2.2.3 Div-by-zero Checker

The goal of the divide by zero checker is to identify executions which trigger a division by zero.

The big part of this is to know the values that an expression can take depending on the path followed, this is done by the core part of the checker, so the only thing left to actually implement in the checker is the verification.

```
class DivZeroChecker : public Checker<check::PreStmt<BinaryOperator>> {
  void checkPreStmt(const BinaryOperator *B, CheckerContext &C) const;
}
```

The Div-by-zero Checker operates before a binary operation: it checks whether the operation is some kind of division (actual division, remainder, division assign or remainder assign), as well as whether the type of the denominator is known, and if so it uses the constraint manager to determine whether the right hand side can be zero and whether it can be non-zero. Both can be true, but both can not be false at the same time. Note that the result of some operations can be unknown range-wise, and in this case all values that the type of the value can take are assumed possible.

If the denominator can not be non-zero, or if it is tainted and can be zero, then the checker reports a bug and returns. Otherwise, it adds a transition to a state where the denominator is non-zero (since after a division in a sound execution the denominator can only be non-zero anyway).

The checker is included in the "core" checkers, so it is run by default when running CSA. If you want to specify that it has to run (in some cases such that when using custom plugins, core checkers can be deactivated), you can add the following arguments:

```
-Xanalyzer -analyzer-checker=core.DivideZero
```

Chapter 3

Analysis of false positives

3.1 Generation

Since our project was focused on understanding the origins of false positives in the Clang Static Analyzer, we spent a few weeks on generating some false positives, as well as analyzing and classifying them to figure out why there existed (actual bug or limitation coming from the design of the analyzer, and in this case why this limitation).

False positives can be looked for through both manual and automatic testing. Since we were just getting started with CSA, we mostly focused on manual testing to understand more how it works (coupled with trying to understand the inner workings to possibly discover new limitations, unimplemented cases or bugs), but still tried automatic generation as well to easily find a bigger quantity of false positives.

3.1.1 Automatic

Following an advice from Andrés Sanchez, an HexHive Laboratory PhD student, we briefly tried generating false positives using csmith[13], a random generator of C program, which claims that the programs it outputs are free of undefined behaviors. This was perfect for our use case as it would allow us to either find false positives for the Clang Static Analyzer or counter this claim !

There is an obvious problem with automatic generation of false positives and false negatives: when we generated a sample code and CSA reports some issue, we can not automatically prove that the analyzer is correct, we still have to look at it manually, understand what the code does and try to simplify it to figure whether it is an actual bug or not. However the automatic approach still had the benefit of very quickly providing some code where CSA reports something.

We used the following command to generate examples, coupled with a script[1] to run it in a loop and stop once CSA reports a warning on the given piece of code. For reasons explained below, the script also filters the generated code so that it is not too long.

```
$ csmith --concise --no-jumps --no-paranoid --no-builtins
```

–concise removes most comments in the generated file

–no-jumps disables the use of goto

–no-paranoid disables pointer-related assertions

–no-builtins disables the use of compiler builtins

We also disabled the deadcode checker when running CSA because the generated pieces of code contained lots of unused variables.

We encountered several issues using this method, in particular the generated code is very long and obfuscated, so understanding whether there is really an issue is very hard, since it is usually tens of functions calling each other and using dozens of variables, both local and global, and a bug reported by CSA can be a potential bug not triggered (eg. returning a local pointer from a function is really a bug only if it is then dereferenced), which is fair for both CSA and Csmith.

See `https://github.com/pgimalac/semester-project-csa/blob/main/automation/csmith/example.c` for such an example. It is 994 lines long, and CSA reports a dangling reference, but which is (probably) not dereferenced, so Csmith's claim is still valid.

To simplify the analysis of the generated code, we tried to limit its size, but then no generated code would report a bug (basically below 800 lines no bug was ever reported).

We eventually decided to stop trying with the automated method and only try things "manually", so that at least we understood exactly what we were trying to trigger.

3.1.2 Manual

We did not use libraries to neither generate false positives nor false negatives. There is a few reasons why using a library is not a good idea to find false positives or false negatives.
First of all, if the static analyzer does not know anything about the code of a library function then it assumes it can return anything or do anything. This is a fair supposition because an unknown function could potentially do anything. We generated a false positive demonstrating this point, cf. section 3.5. Furthermore, If we do not have access to the code, then this makes it difficult to

[1] `https://github.com/pgimalac/semester-project-csa/blob/main/automation/csmith/run.sh`

analyze what is happening. On some cases the analyzer would be correct but on many cases the library functions would not actually be problematic. This is also makes it difficult to improve the static analyzer.
Second, the more code we have, the bigger the exploded graph would be and it would be harder to actually analyze, prove the claim of the static analyzer and eventually find the problems in the static analyzer.
Thus, we wanted to create simple examples to generate false positives for easier debugging and argumentation of the CSA claim.

There are various kinds of false positives: it can be a bug inside an existing checker, a limitation of the core functionalities of CSA or be due to something not being implemented (yet). Actual bugs are interesting as they can be solved, but they're also much harder to find, and are not particularly interesting when trying to explain how the analyzer works and what limitation it implies, which is somewhat the goal of the project. For this reason, we mostly focused on code containing an obvious bug (eg. a division by zero) but which is actually dead code, and trying to trick CSA into thinking it was not dead code.

To create a false positive, we started from a true positive. For example:

Listing 3.1: boilerplate_divide_zero.cpp

```
int example() {
    return 1/0; // warning: Division by zero [core.DivideZero]
                // division by zero is triggered by the CSA
}
```

After this, we modified the code by adding ifs, loops, or functions to try to transform the problematic lines into dead code, cf. section 3.2.

3.2 Ranges

3.2.1 Description

In this section, we give an example of a wrong calculation on a range that leads to a false positive. Note this example can easily be changed to create more examples of false positives.

For the following code, the condition `ifCondition` < 0 will never be true, the calculation we make for the variable `ifCondition` is always bounded by 0 and `INT_MAX`. The internal `if` condition is, of course, dead code, and can not be reached. But CSA says it can be negative. Another way to test it out is to add the condition directly in the outer `if`, thus making the entire `if` condition dead code but the CSA would still be triggered.

Listing 3.2: ranges.cpp

```
#define UPPER_BOUND 127
// The smallest possible value who would trigger a false positive is 3
// The biggest possible value would be INT_MAX

int innerIf(int a){ //, int b) {
    int val = 1;
    if(a > 0 && a < UPPER_BOUND // && b > 0 && b < UPPER_BOUND)
    {
        int ret = 3;

        // CSA does keep its assumption on 'a' but makes a wrong
        // calculation on the upper bound
        int ifCondition = a >> 2;
        if(ifCondition < 0) { // Actually never true, but the CSA says
                              // it can be.

                              // Note the condition can also
                              // be replaced by 'a+b', 'a*b', or 3*a,...
                              // Actually any operator can be used

                              // This could also be generalized
                              // by adding more conditions to the outer
                              // if
            ret = 0;
        }

        val = ret; // Should always be 3
    }

    return 1/val;
}
```

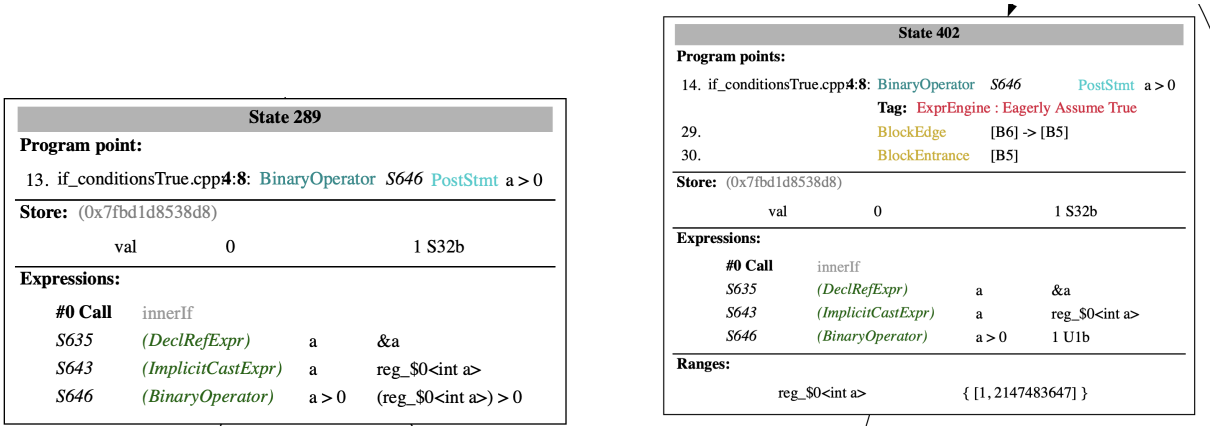
Adding an assertion before the definition of the variable `ifCondition` resolves the problem. We tried to fix this false positive, see section 4.1

3.2.2 Analysis

Let us argument the claim that the range calculation is the problem and is exactly what causes the false positive on the above code. The initial range of `a` is $[INT\_MIN, INT\_MAX]$. This range stays the same before reaching the first `if`. This is because `a` is an `int`.

Binary Operator: $a > 0$

After generating the exploded graph and checking what happens before and after the first condition, namely > 0, the static analyzer generated the states for both evaluations of the binary operator, $a > 0$. I.e., we have both the states before and after the `if` condition.
In the pictures below, (cf. fig. 3.1), the left-hand graph is the generated state before the if statement and the right-hand one is the state CSA created after assuming the condition > 0 to be true.
After the first condition, the range should be $[1,\ INT\_MAX]$. As we can see on the right-hand side graph, we have the expected and correct range. Note the *Eagerly Assume False* part has been evaluated and generated by the CSA but it has not been included as it does not lead to a false positive. It is not problematic and does exactly what is expected. Indeed, the following range does not lead to false positive $[INT\_MIN,\ 0]$ as it never enters the if condition and at this point `val == 1`.



(a) Before the if condition

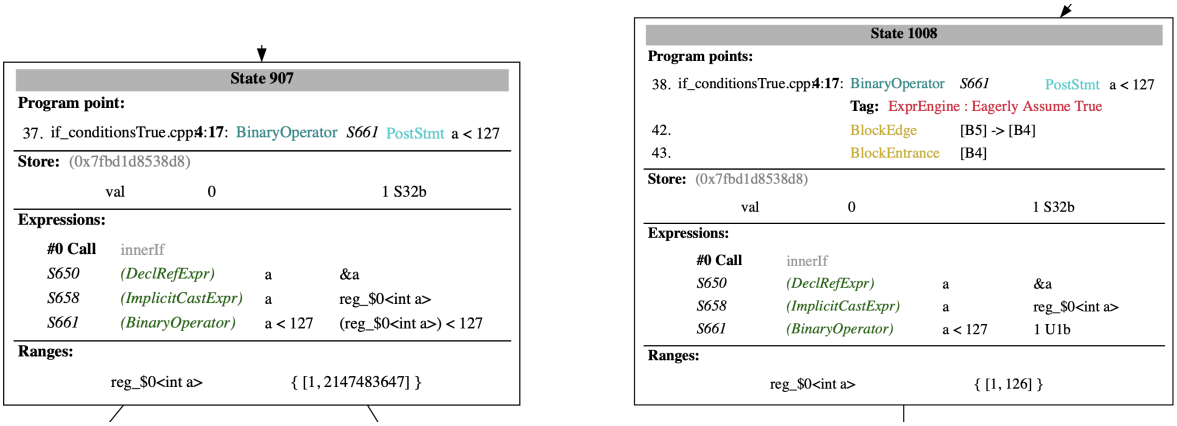
(b) True case

Figure 3.1: The states before and after the condition $a > 0$. The false case is omitted.

Binary Operator: $a < 127$

A few states later, the CSA arrives at the evaluation of the binary operator $a < 127$. Once again, we only show the graphs before and after the condition. We also omit the false condition as it does not lead to a false positive and is correct. Indeed, the following range does not lead to a false positive $[127,\ INT\_MAX]$.

Once again the left-hand side is the generated state before the evaluation of the condition, while the right-hand one is the one created after the symbolic evaluation of the binary operator. The ranges are as expected and correct. Indeed, after the symbolic evaluation of the previous binary operator $a > 0$, we had $[1,\ INT\_MAX]$. The new evaluation of the binary operator $a < 127$ leads to the range $[1, 126]$, cf. fig. 3.2.



(a) Before the if condition

(b) True case

Figure 3.2: The states before and after the condition $a < 127$. The false case is omitted.

Binary Operator: $ifCondition < 0$

The states at the line `int ifCondition = a » 2` is not shown as the state is not really interesting. Furthermore, as explained earlier in the background section, the calculation of the new range is not made at this line because of the lazy evaluation but it is done further down the line. At this stage a new symbolic value is created to represent `(reg_$0 < int a >) >> 2`.

A few states later, before the `if(ifCondition < 0)`, we have that the range is once again correct, cf. fig. 3.3).

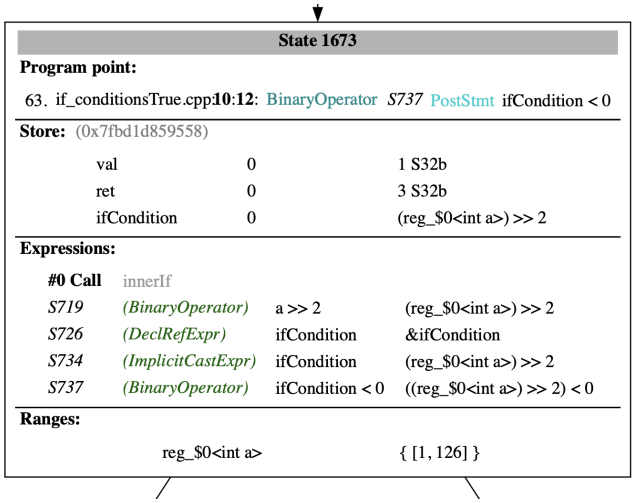
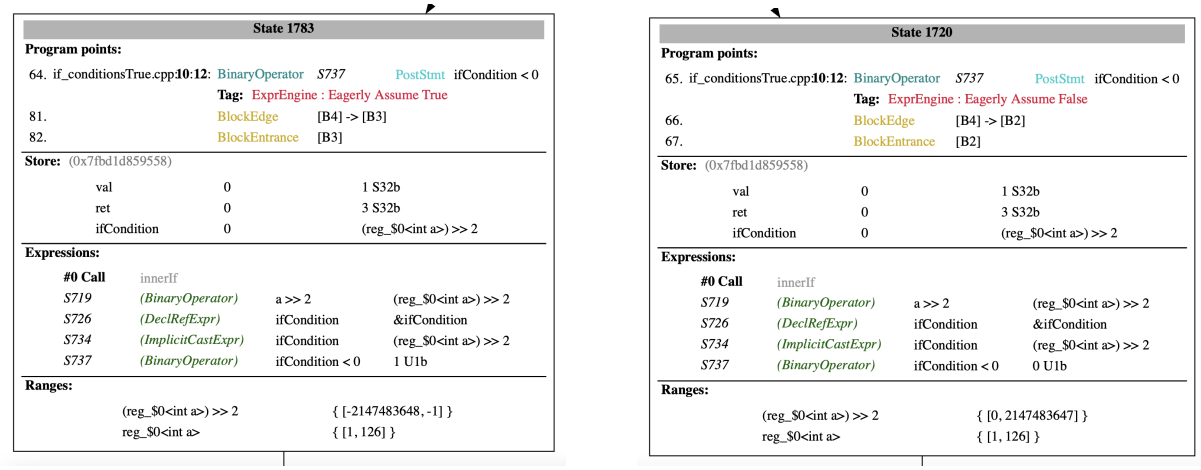


Figure 3.3: The state before `ifCondition < 0`

After the symbolic execution of the code, the exploded graph becomes, cf. fig. 3.4. Now, we can see the range for the new symbolic value created earlier `(reg_$0 < int a >) >> 2` is actually wrong. It should be $[0, 31]$ for the false case and be the empty range for the true case.

The range of a is also wrong. It should be $[1, 126]$ for the false case, and the empty set for the true case. Making the `if` case dead code.



State 1783

Program points:

64. if_conditionsTrue.cpp**10**:**12**: BinaryOperator *S737* PostStmt ifCondition < 0
Tag: ExprEngine : Eagerly Assume True
81. BlockEdge [B4] -> [B3]
82. BlockEntrance [B3]

Store: (0x7fbd1d859558)

val	0	1 S32b
ret	0	3 S32b
ifCondition	0	(reg_$0<int a>) >> 2

Expressions:

#0 Call	innerIf		
S719	*(BinaryOperator)*	a >> 2	(reg_$0<int a>) >> 2
S726	*(DeclRefExpr)*	ifCondition	&ifCondition
S734	*(ImplicitCastExpr)*	ifCondition	(reg_$0<int a>) >> 2
S737	*(BinaryOperator)*	ifCondition < 0	1 U1b

Ranges:

(reg_$0<int a>) >> 2	{ [-2147483648, -1] }
reg_$0<int a>	{ [1, 126] }

(a) True case

State 1720

Program points:

65. if_conditionsTrue.cpp**10**:**12**: BinaryOperator *S737* PostStmt ifCondition < 0
Tag: ExprEngine : Eagerly Assume False
66. BlockEdge [B4] -> [B2]
67. BlockEntrance [B2]

Store: (0x7fbd1d859558)

val	0	1 S32b
ret	0	3 S32b
ifCondition	0	(reg_$0<int a>) >> 2

Expressions:

#0 Call	innerIf		
S719	*(BinaryOperator)*	a >> 2	(reg_$0<int a>) >> 2
S726	*(DeclRefExpr)*	ifCondition	&ifCondition
S734	*(ImplicitCastExpr)*	ifCondition	(reg_$0<int a>) >> 2
S737	*(BinaryOperator)*	ifCondition < 0	0 U1b

Ranges:

(reg_$0<int a>) >> 2	{ [0, 2147483647] }
reg_$0<int a>	{ [1, 126] }

(b) False case

Figure 3.4: The states after the symbolic execution of the `ifCondition < 0`

3.2.3 More false positives using the same example

The above code can be changed to lead to more false positives, for example if we make a different calculations on the `ifCondition` variable. For a constant, any use of an operator that is not + or - would lead to a false positive. Though, some obvious cases are taken into account. For example, a multiplication by 0, or by 1.

Furthermore, we can create more false positives if we add more variables to the outer if. One could also create *three-if* stages with contradicting conditions. The `UPPER_BOUND` macro can also be changed to a minimum value of 3.

3.3 Class variable

3.3.1 Description

For the following code, the variable `x` will always be equal to 1. As we increment the internal variable of the `Vector`, `sizeV`, the *if* condition in the `returnFive` function will never be triggered and is thus dead code. The `Vector` class has been manually created to remove any problem with external function for the CSA.

Listing 3.3: class_variable.cpp

```
// This is a stripped version of the standard std::vector so as to
// simplify the example.
// Thus, only the useful functions have been implemented.

// Note also that usually the 'incrementSize' function does not exist
// in a vector class.
// It has only been created to simplify the example.
class Vector {
private:
    unsigned long long sizeV = 0;

public:
    Vector() {
    }

    void incrementSize() {
        sizeV++;
    }

    unsigned long long size() const {
        return sizeV;
    }
};

class Test {
private:
    Vector vector;

public:
    int returnFive() {
        int x = 0;
        vector.incrementSize(); // vector.sizeV == 1
        x = vector.size(); // vector.size() == 1

        if(x == 0) { // x is never equal to 0.
            return 1/x; // But the CSA still triggers this line
        }

        return 5;
    }
};
```

Note that if we move the internal variable of the `Test` class to the function `conditions` then the CSA does not trigger a false positive anymore.

3.4 Private Dead Code

3.4.1 Description

The following false positive is to highlight that the CSA does not take into account private functions that can not be called and that are never called are dead code.

Listing 3.4: private_dead_code.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <climits>


class Test {
public:
    int functionOne() {
        return 1;
    }

    int functionTwo() {
        return 2;
    }

private:
    int neverCalled(int a) {
        int count = 0;

        for(int i = 0; i < a; ++i) { // a could be smaller than zero
                                     // and
                                     // thus the loop would never run
            count = a;
        }

        if(a == 0) {
            return 0;
        }

        return 1/count;  // when the loop does not run count == 0
```

```
                        // which would trigger a bug but
                        // this function is private
                        // and is never called by anyone.
                        //Thus making it dead code.
    }
};
```

3.5 Library functions

3.5.1 Modelisation

Description

Some functions (in particular most functions from the libc) are modelled, so that CSA can handle their return values (and know that not any value can be returned for example) as well as what parameter they should be given.

In this example, CSA is aware that strcmp returns a value in -1, 0, 1, so that the condition is impossible, and thus it does not warn of a division by zero in the first if.

On the other hand, it has no idea what rand returns, so it assumes that it can return a negative value, while the documentation specifically says that it can not. We found this bug by looking around in the checkers source code, in particular in "StdLibraryFunctionsChecker".

Listing 3.5: unmodelled_libc_function.c

```
#include <stdlib.h>
#include <string.h>

int main() {
    const char *s1 = "a";
    const char *s2 = "b";

    // the analyzer is aware that strcmp returns in {-1, 0, 1}
    // => no warning here
    int ret = strcmp(s1, s2);
    if (ret == 2) {
        return 1 / 0;
    }

    // but not that rand returns in [0, RAND_MAX]
```

```
    // => warning here
    ret = rand();
    if (ret < 0) {
        return 1 / 0;
    }
}
```

3.5.2 Modeling error

Description

This example shows an issue in how malloc and other functions are modeled.

When we call malloc at the beginning of the function, two warnings are raised by CSA, for the conditions with x and y (the two global variables). However, if we replace malloc by any other function, then no warning is raised. See `https://discourse.llvm.org/t/possible-bug-with-malloc-modeling/66124` for a discussion on this topic where a main developer explains that it should actually be the other way around.

The default behavior for un-modeled functions should be to assume they can do anything, including change global variables defined in the current unit, while on the other hand malloc is supposed to be well-known and should not be considered able to write global variables.

Listing 3.6: malloc_modeling.c

```
int x = 0;
static int y = 0;

int main() {
    int z = 0;

    // free is only called to prevent a warning about un-freed memory
    free(malloc(sizeof(int)));

    if (x != 0) {
        // with malloc, warning here
        return 1 / 0;
    }

    if (y != 0) {
        // with malloc, warning here
        return 1 / 0;
```

```
    }

    if (z != 0) {
        return 1 / 0;
    }

    return 1;
}
```

3.6 Weird memory accesses

3.6.1 Description

The idea of this example was to have a piece of code which changes a variables without CSA being able to detect it.

The example is undefined behavior (array out-of-bounds access) so not very relevant, but we still thought that it could be theoretically interesting, and it actually encouraged us to look deeper into how CSA modeled memory and we were able to find way more interesting false-positives.

Variable x is initialized at value 0, then changed through an out of bound access of data, and lastly we return 1000/x, triggering the divide by zero warning while value x is actually not 0 anymore.

Listing 3.7: weird_memory_access.c

```
int main() {
    int x = 0;
    int data[1];

    // undefined behavior but the idea was to showcase
    // a situation where the analyzer can not understand
    // that a variable was changed
    data[-1] = 1;

    return 1000 / x;
}
```

3.7 Bit operations

3.7.1 Description

Integer types are not modelled at the bit level, so bit operations are assumed to produce any result, and a simple ‘x & ~x‘ is considered to be possibly non-zero.

Listing 3.8: bit_operation.c

```
#include <stdlib.h>

int main() {
    unsigned x = (unsigned)rand();
    if (x & (~x)) {
        // x & ~x is impossible, whatever x
        return 1 / 0;
    }
    return 1;
}
```

3.7.2 Mitigation

This false positive can be mitigated by using z3 to crosscheck, using the following parameter (note that clang has to be built with z3 for this to be supported):

```
-Xanalyzer -analyzer-config -Xanalyzer crosscheck-with-z3=true
```

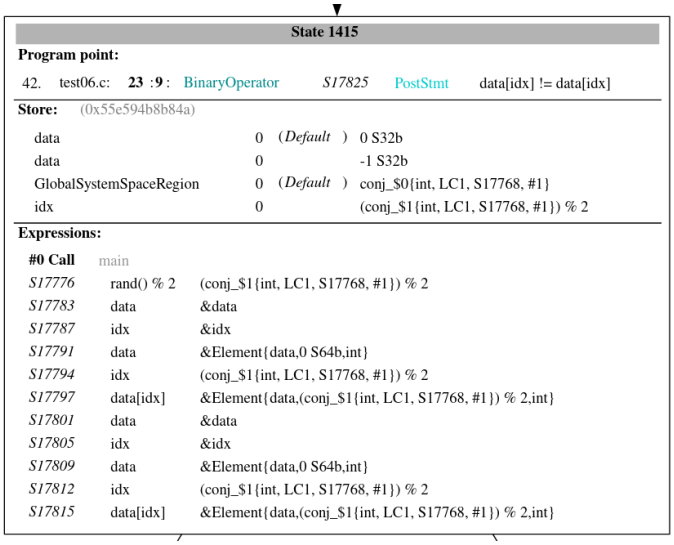
This option allows to crosscheck the condition at bit level using z3, thus to refute some conditions with more precision.

3.8 Memory modelling of arrays

3.8.1 Description

As explained in section 2.1.1, reading a constant index of an array basically considers the array as n distinct variables so everything works as well as it can, but reading a non-constant index (which has not been written before) returns an unknown symbol (and not the "same unknown" each time), which means that the simple example below reports a false positive.

Here is the state of the exploded graph just after computing the expression `data[idx] != data[idx]`, we can observe that `data[idx]` is not in the store (we do not know what value it is associated to) but that the default value of the array data is (and it is -1 signed 32 bits), the return value of `rand()` is associated to the (unknown) symbolic value `conj_$1(int, LC1, S17768, #1)`, `idx` is associated to this same symbolic value modulo 2, and the L-values `data[idx]` both are `Element{data,(conj_$1int, LC1, S17768, #1) % 2,int}` (symbolic value representing an L-value being an array element, with the origin array, the index and the type).



State 1415

Program point:

42. test06.c: **23** :**9** : BinaryOperator *S17825* PostStmt data[idx] != data[idx]

Store: (0x55e594b8b84a)

data	0	(*Default*)	0 S32b
data	0		-1 S32b
GlobalSystemSpaceRegion	0	(*Default*)	conj_$0{int, LC1, S17768, #1}
idx	0		(conj_$1{int, LC1, S17768, #1}) % 2

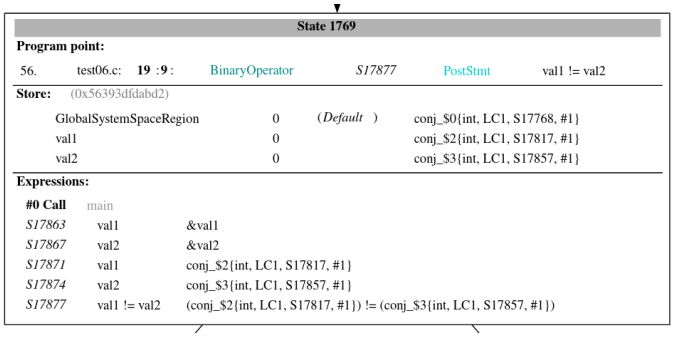
Expressions:

#0 Call main

S17776	rand() % 2	(conj_$1{int, LC1, S17768, #1}) % 2
S17783	data	&data
S17787	idx	&idx
S17791	data	&Element{data,0 S64b,int}
S17794	idx	(conj_$1{int, LC1, S17768, #1}) % 2
S17797	data[idx]	&Element{data,(conj_$1{int, LC1, S17768, #1}) % 2,int}
S17801	data	&data
S17805	idx	&idx
S17809	data	&Element{data,0 S64b,int}
S17812	idx	(conj_$1{int, LC1, S17768, #1}) % 2
S17815	data[idx]	&Element{data,(conj_$1{int, LC1, S17768, #1}) % 2,int}

Figure 3.5: The state just after `data[idx] != data[idx]`

What does not appear here is the load of the values (casting them from L-value to R-value, loading the associated value from memory) where both `data[idx]` get replaced by (new) unknown value. To make it appear, we tweak the example a little bit by storing both values in variables before comparing. Here is the new exploded graph state for this example, with most other entries removed.



State 1769

Program point:

56. test06.c: **19** :**9** : BinaryOperator *S17877* PostStmt val1 != val2

Store: (0x56393dfdabd2)

GlobalSystemSpaceRegion	0	(*Default*)	conj_$0{int, LC1, S17768, #1}
val1	0		conj_$2{int, LC1, S17817, #1}
val2	0		conj_$3{int, LC1, S17857, #1}

Expressions:

#0 Call main

S17863	val1	&val1
S17867	val2	&val2
S17871	val1	conj_$2{int, LC1, S17817, #1}
S17874	val2	conj_$3{int, LC1, S17857, #1}
S17877	val1 != val2	(conj_$2{int, LC1, S17817, #1}) != (conj_$3{int, LC1, S17857, #1})

Figure 3.6: The state just after `val1 != val2`

In this one, we can see that both values were loaded as different unknown symbolic values, respectively `conj_$2{int, LC1, S17817, #1}` and `conj_$3{int, LC1, S17857, #1}`, which explains why comparing them considers that they can be different.

We tried to fix this false positive, see section 4.2.

3.8.2 Code

Listing 3.9: array_access1.c

```
int main() {
    int data[2] = {-1};
    int idx = rand() % 2;

    if (data[idx] != data[idx]) {
        return 1 / 0;
    }

    return 0;
}
```

Listing 3.10: array_access2.c

```
int main() {
    int data[2] = {-1};
    int idx = rand() % 2;

    int val1 = data[idx];
    int val2 = data[idx];

    if (val1 != val2) {
        return 1 / 0;
    }

    return 0;
}
```

Chapter 4

Improvements

In this chapter, we are presenting two improvements to the Clang Static Analyzer: ranges and memory modelling of arrays.

The idea of the first improvement comes from the false positive example with ranges, cf. section 3.2, and is aimed to implement more operators. We also explains in more detail what is causing the problem and how resolve some of it.

The idea of the second improvement comes from the false positive with arrays (section 3.8), the example is very simple but comes directly from how the memory model works and how symbolic values are stored in the environment and the store. The correction we implemented is rather simple (and very far from perfect), but this work encouraged us to look deeper into how CSA models programs, memory, executions and symbolic values, in particular for arrays. We will explain how the false positive could be solved, how we actually tried to solve it, what the current status of the fix is and what drawbacks the chosen method has.

4.1 Ranges

The fix for the ranges can be found at `https://github.com/cjordan7/llvm-project/tree/3b71f96784bc9daae7d16581e9c2274b5a8af7f5`

The Clang Static Analyzer original code to calculate ranges are for now only implemented for any sized integers, which is modelled by the `APSInt` class. We extended the class with our own shift operator where the operand can be shifted by an `APInt`.

In the original code, doubles are not yet handled. Furthermore, only the plus and minus operators are currently implemented. All comparisons operators have been implemented for the both of them.

Our modification of the ranges only takes into account constants for all defined operators for any sized integers.

Our implementation contains the following modifications: we saved which binary operator is currently being treated in a variable called `OperationOpcode` when the function `assumeSymRel` is executed. The function `computeAdjustment` gets the constant for each binary operators similarly as for the add and plus operators. The functions `getSymLERange`, `getSymLTRange`, and their similar functions have been extended to calculate the ranges for other common binary operators. We use a switch and `OperationOpcode` to find the correct calculation. Now, let us speak about how to calculate each range. Note, the code is similar to the explanation given below.
Without a lack of generality, we will only show the ranges for the operator $<$ and for ints only. The ranges of the operators $<=$, $>$, $>=$, $=$ and $!=$ are similar.

To implement this, we leveraged the existing code. Indeed, we do not need to re-implement a `RangeSet` data structure but we can use it. We only need to be able to handle cases that are not already there. To do so, we extended the number of operators that are handled. We used the same idea as for the plus and minus operator.

4.1.1 Multiplication and division operators

The same idea as for the plus and minus operator case has been used. The idea is simply to solve the equation, or inequality, to get the new range. For example, if we have $a + 1 < 10$, then we get $a < 9$ and thus both ranges for a are $[INT\_MIN,\ 8]$ if the condition is true and otherwise is $[9,\ INT\_MAX]$ if the condition is false. If we suppose that a is an `int`.

Multiplication Operator

For the multiplication operator, we have $a * x < y$, where x and y are constants, and x is positive. Thus, when resolving the equation, we get $a < y/x$. Note, we do not need to take into account the case where x is zero, this is one of the trivial case that has already been taken into account. So both of the ranges would thus be $[INT\_MIN,\ y/x - 1]$ and $[y/x,\ INT\_MAX]$. But notice that the positive elements such that $a * x > INT\_MAX$ implies an overflow and would thus also be negative and smaller than y. To find such number, we simply resolve $a > INT\_MAX/x$. Thus, if the equation is true, we have $[INT\_MIN,\ y/x-1]$ and $[INT\_MAX/x+1,\ INT\_MAX]$[1] otherwise, $[y/x,\ INT\_MAX/x]$.

If x is negative, then we simply have to solve $a > y/x$. The calculation of both the true and false ranges is similar as above.

[1] We use the wraparound concept to get both ranges.

Pitfalls of Multiplication

This solution for the multiplication is not perfect. Indeed, if we have a negative range such as $[INT\_MIN,\ 0]$ and we multiply by 2 then all negative numbers that have a zero at the third leftmost bit would be positive. But our implementation says they are negative.

The range is actually not fully correct. If we suppose, for simplification, that we multiply by 4, we have $a * 4 < y$. This is equivalent to shifting by 2. All the numbers such that the third leftmost bit is 0 would be positive and not negative. Thus the negative ranges who would also be positive are $[0b100\bar{1}, 0b100\bar{0}]$ and $[0b110\bar{1}, 0b110\bar{0}]$. where $\bar{1}$ and $\bar{0}$ are used as a notation to fill the bit numbers with 1 or 0 respectively.

In this example it would be easy to also add both of those ranges to the set of calculated ranges that the CSA already uses. But if we multiply by a bigger number for example 2^{21}, then we would would have as many ranges. This would be impossible to realistically keep track of them because of speed and memory constraints.
Furthermore, we have another problem. If we multiply by 3 instead of 4, the ranges are not as straight forward.

Division Operator

For the division operator, we apply the same idea. Once again x and y are constants, and x is positive. We solve $a/x < y$, which becomes $a < y * x$. Thus, if the equation is true, the range is $[INT\_MIN,\ y * x - 1]$ otherwise it is $[y * x,\ INT\_MAX]$. For the division operator, we do not have the same problem as with the multiplication operator. Indeed even if $a < y * x$ is mathematically correct it is not fully equivalent to $a/x < y$ when we work with integers. So if $y * x = s$ overflows and becomes negative, the inverse operation s/x would not be positive. Thus if $y * x > INT\_MAX$, then the range for the true case is $[INT\_MIN,\ INT\_MAX]$. We can detect this overflow by a simple boolean test such as: $y > INT\_MAX/x$.

Once again, if x is negative, then we simply have to solve $a > y/x$. The calculation of both the true and false ranges is similar as above.

4.1.2 Shift operators

Left shift operator is a special case of the multiplication operator and the right shift operator is a special case of the division operator. So the both of them have been implemented accordingly with a few differences.

The solution of the equations $(a >> x) < y$ or $(a << x) < y$ are respectively done in the following way $a < (y << x)$ or $a < (x >> y)$. The parenthesis have only been added for easier

reading.

Note the multiplication pitfall also exists for the left-shift operator. But in this case we actually know all the elements that would overflow. Indeed all $y + 1$ left and right most bits equal to zero and one respectively would change its sign. We could actually keep this constraint in memory. For example for a positive range, we would have $(x >> (y + 1)) \,\&\, 0 == 0$ is positive otherwise is negative.

The same remark for the division operator also exists for the right-shift operator. If $y << x$ overflows, then we simply set the upper bound of the range to the max value of the integer.

If the operand of either the left or right shift operators is negative, then by the definition of the *C* and *C++* standards and *Objective C* we have an undefined behavior [14], [15], [16]. Note *Objective C* does not have a standard but the Clang compiler still warns the user of an undefined behavior.

If we have that special undefined behavior, we simply default back to the current unmodified implementation. We do it for two reasons: First, it is not the role of the static analyzer to inform the user of an undefined behavior. And second, this is not problematic because when we run the Clang Static Analyzer, other parts of Clang already took it into account and will give a warning, *shift count is negative*, about the behavior and thus can be corrected by the user to avoid a bug.

4.1.3 Modulo operator

The modulo operator could not be implemented because of the fundamental flaw of the current implementation of the ranges.

Obviously the case such as $a\%a$ and $a\%1$ both equal to zero can be implemented and have been implemented.

Let us give an example to better understand why this is the case. Let suppose that we have two ranges such as $[INT\_MIN,\ -1]$ and $[0,\ INT\_MAX[$. And we calculate a new range for modulo 4. We get 4 different sets. Thus the modulo ranges for the positive range are the sets $\{0, 4, 8, ...\}$, $\{1, 4, 8, ...\}$, $\{2, 5, 9, ...\}$, $\{3, 6, 10, ...\}$. We have 4 similar sets for the negative range. The current implementation only takes into account ranges and not sets.

So if we wanted to get the range of $a\%4 == 0$, we would have to return the set $\{0, 4, 8, ...\}$. This can not be done with the current implementation. We also could not implement some of the same ideas for the potential improvements of the multiplication and shift left operator. We can not keep the full set in memory because of speed and memory constraints.
The solution for a possible improvement of the multiplication would not work. But we could use the same idea as the shift operator. Meaning we would keep the constraint in memory without any calculation and if we have a division such as $1/a$. We could ask the constraint and see that

$a\%4 == 0$ is actually possible, thus implying a division by zero because a could be zero.

4.2 Memory modelling of arrays

This improvement is much smaller than the previous one, but stems from the very way CSA models memory and stores symbolic values, so it also took us a long time to understand what was going on (cf. section 2.1.1, section 3.8 and section 4.2.1), think of ways to solve the issue, actually try to implement one and see its limitations.

4.2.1 Description of the issue

The issue is already detailed section 3.8 but we will explain it briefly again here.

The most simplified example of the issue is the following:

Listing 4.1: array_fp_simplified.c

```
int main() {
    int data[2] = {-1};
    int idx = rand() % 2;

    if (data[idx] != data[idx]) {
        return 1 / 0;
    }

    return 0;
}
```

The idea is that every time you read from an array with a non-constant index[2], the function `getSVal` is first called with the statement to obtain the symbolic L-value corresponding to the array element, which is of type `ElementRegion`, and contains the origin array, the index and the type (see fig. 3.5 for an example).

Listing 4.2: `getSVal` definitions in `ProgramState.h`

```
/// Returns the SVal bound to the statement 'S' in
/// the state's environment.
SVal getSVal(const Stmt *S, const LocationContext *LCtx) const;
// < checks in the environment > //
```

[2] Remember that when using constant indices the array is basically considered as n distinct variables, the symbolic value fetched is directly linked to the variable, so comparing the element is like comparing a variable to itself, which works fine

```
/// Return the value bound to the specified location.
/// Returns UnknownVal() if none found.
SVal getSVal(const MemRegion* R, QualType T = QualType()) const;
// < checks in the store > //
```

Then, the symbolic value is cast to an R-value, during this process the other `getSVal` is called to fetch the actual value in memory of the array element (an L-value is a location and an R-value is a value), but since it is unknown it returns an unknown symbolic value, cf. the comment on the 2nd `getSVal` above.

This unknown value is somewhat "a new unknown" every time the array is read from, which means that reading twice the same element results in two different unknowns and CSA cannot know that they are always the same.

4.2.2 Possible fixes

We thought of different ways to fix this issue, with each their benefits and drawbacks. They're mostly the same initial idea of actually returning a custom symbolic value (the "custom" part varies depending on the version) instead of the unknown value, which we simplified bit by bit when we realized that it would be too hard to implement or too costly computationally or too inaccurate.

Any fix would suffer from the aliasing issue, which is that almost any array or pointer can point to the same memory location, but the default implementation is the most conservative possible (always assume that elements of an array can be different, even the same element) so improving it a little was still possible.

Complete fix

Our first idea was the most complex and optimistic, it consisted in associating to each region a version number which would be increased when an unknown index is written to, as well as having a global version number which would also be increased when a non-local pointer or array is written to (to avoid the aliasing issue as well as possible), and eventually returning an `ElementRegion` instead of the unknown symbolic value `getSVal` returns when it can not find the element in the store, but adding the two version numbers included in it.

Then, comparing two such value would work fine because it would reduce to comparing the source array (or equivalently pointer address, to ensure it is the same region), the index (to ensure it is the same element), the local version number (to ensure it is the same value in case

an unknown index was written to) and the global version number (to ensure it was not changed using an external pointer).

Managing local and external pointers and arrays could allow to be more precise in tracking the different values and preventing some false positives, but it is also very tricky and adds a lot of complexity to this potential fix, so we decided to drop it.

Full-external fix

If we do not consider which pointers are local or external, we can just consider that all pointers and arrays can alias at any index, so there is no need to associate to each region a version number, the global version number alone is enough.

With this version, array element and pointers can be compared as well as possible between two writes, and comparisons across writes are always considered potentially different.

However, this fix still requires changing quite a few things in CSA (in particular adding a global constant version number in the state), which is a nightmare considering how big and complex the code base is, so we still decided to simplify it even more.

Simplified version

We wanted to avoid as much as possible having to add a version number, even a single global one, so we simplified the fix a little more: instead of a version number we thought of just returning the same `ElementRegion` instead of the unknown symbolic value, and then any time there was a write to a pointer or an array to invalidate the entries in the environment and the store which referred to an `ElementRegion` (either removing them completely, which would result in using an unknown afterwards, or just replacing them with a new symbol, but the same one for each `ElementRegion`).

The drawback compared to the previous version is that if we just completely remove the `ElementRegion` from the environment and the store then we every write to a pointer makes us forget about past variables. However if we instead replace them with an associated fresh symbolic values (eg. the same `ElementRegion` is replaced by the same symbolic value) then we can have something strictly equivalent but hopefully easier to implement.

Another drawback is the computational cost, because the environment and the store are somewhat just a map from location to value, so invalidating the locations means iterating through all of it and checking each of the entries. However we thought that it was worth it in exchange for an easier implementation (we did not have much time left at this point).

4.2.3 Status of the implemented fix and limitations

For the reasons explained in the previous section as well as a lack of time we chose to implement the simplified version described in section 4.2.2.

We began with editing the `evalLoad` function in `ExprEngine.cpp` (which is called by the second `getSVal`) to return the L-value instead of an unknown symbolic value when the location is not in the store. It fixed the two examples we saw in section 3.8 since then instead of comparing two unknown values it just compare the locations (the `ElementRegion`) which are the same thus the issue is gone.

It solved the initial bug (where no writes are done to the array), but not the more complex examples.

However once we start writing into pointers then it is clearly not enough, here is are two similar false negatives which underline the weakness of doing only this simple change (without clearing the environment on write).

The code can be found at `https://github.com/pgimalac/llvm-project/tree/csa-array-fix`.

Listing 4.3: array_fix_false_neg.c

```
int main() {
    int data[2] = {-1};

    int idx = rand() % 2;

    int val1 = data[idx];
    int idx2 = rand() % 2;
    data[idx2] = 1;
    int val2 = data[idx];

    if (val1 != val2) {
        return 1 / 0;
    }

    return 0;
}
```

The first load of `data[idx]` can not find anything in the store so it returns the L-value, then a random index (which can be the same as `idx`) is written and added to the store, and eventually the second load of `data[idx]` returns the location just like the first, because the index in the store is not the same. The comparison then considers that they can be different and a bug is reported.

The example below is similar, but instead of writing to a random index we write to `(2 - idx) % 2`, which is actually exactly the same as `idx`. Again in this situation both `val1` and `val2` are loaded as L-values (the exact same) because there is no entry with index `idx` in the store ! This issue can be considered to come from CSA not understanding that both indices are the same, but with the second part of the fix it would at least have been reported.

Listing 4.4: array_fix_false_neg2.c

```
int main() {
    int data[2] = {-1};

    int idx = rand() % 2;

    int val1 = data[idx];
    data[(2 - idx) % 2] = 1;
    int val2 = data[idx];

    if (val1 != val2) {
        return 1 / 0;
    }

    return 0;
}
```

In the end we did not have time to implement correctly the second half of the fix, we implemented flushing the environment but then realized that we also had to flush the store, since the L-values returned can be assigned to a variable (which will effectively add a binding from the variable to the location in the store), and then we have a similar issues as we had with the environment. Flushing the store was harder than the environment because we can't iterate it easily (there is a function to iterate it but it doesn't give enough information to filter it).

4.2.4 Issues remaining after the fix

The goal of the fix was to prevent the initially explained false positive, without causing new false positives or false negatives (ie. to do strictly better than the previous implementation, as much as possible). It was not able to fix some other somewhat related issues that are due to the way programs are modelled.

The example below is a false positive with the fix of CSA explained above, because it warns that a division by zero can happen while it actually can not. `data[idx]` is initially unknown so loaded as an L-value (an `ElementRegion`), but then the value is written as `-1` so the second load actually returns `-1` this time, and these values can be different.

Listing 4.5: array_fix_false_pos.c

```
int main() {
    int data[2] = {-1};

    int idx = rand() % 2;

    int val1 = data[idx];
    data[idx] = -1;
    int val2 = data[idx];

    if (val1 != val2) {
        return 1 / 0;
    }

    return 0;
}
```

The issue here is actually more that the first load did not return -1 ; it would need to check that no change to the array has been done (which with the current way the store works would mean iterating through it) and so that the default -1 value should be used.

Chapter 5

Conclusion

In this report we gave a brief overview on how the *Clang Static Analyzer* and its checkers work, then how we found some false-positives and where they come from, and eventually tried to fix two of the bugs we found and explained how and why.

The first improvement was on how ranges are computed when there is an operation, the improvements are good enough for both the division and the right-shift operator. But improving the multiplication, the modulo and the left-shift operator is much more complicated. The implemented solution is not perfect, and neither are the proposed improvements. Some of them are costly in terms of memory and calculation cost. This in turn would not abide by the goals of Clang: to have a *fast and scalable approach*.

The second one was improving the comparison of array elements with non-constant indices. The previous implementation was such that comparing an element to itself was considered to be possibly different (even with no writes in between), and we modified the implementation of how values are fetched from the environment and the store in order to avoid this particular issue. We thought of several ways to do it but eventually chose the simplest one, mostly because of time constraints. With this fix, the false positive described in section 3.8 is no longer reported, but since it's not perfect other similar false positives can still be reported.

More research would be needed to improve the static analyzer. We found quite a few false positives while the tool is supposed to have none, so clearly there is room for improvement, and we only studied very few checkers (and mostly the smaller ones, some can be thousands of lines of code long and thus are probably more prone to bugs), but most of the ones we found came from the limitations of the symbolic execution method and of CSA itself. Furthermore, we only worked on one checker.

Bibliography

[1] The Guardian. *Apple security flaw 'actively exploited' by hackers to fully control devices*. [Online; accessed 29-December-2022]. 2022. URL: https://www.theguardian.com/technology/2022/aug/18/apple-security-flaw-hack-iphone-ipad-macs.

[2] NIST. *NATIONAL VULNERABILITY DATABASE*. [Online; accessed 29-December-2022]. 2022. URL: https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&search_type=all&isCpeNameSearch=false.

[3] The Clang Team. *Clang Static Analyzer*. [Online; accessed 28-December-2022]. 2022. URL: https://clang.llvm.org/docs/ClangStaticAnalyzer.html.

[4] The Clang Team. *Clang - Features and Goals*. [Online; accessed 28-December-2022]. 2022. URL: https://clang.llvm.org/features.html#performance.

[5] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. "An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 544–555. ISBN: 9781450393799. DOI: 10.1145/3533767.3534380. URL: https://doi.org/10.1145/3533767.3534380.

[6] Péter György Szécsi, Gábor Horváth, and Zoltán Porkoláb. "Improved Loop Execution Modeling in the Clang Static Analyzer". In: *Acta Cybernetica* (2020).

[7] Marcelo Arroyo, Francisco Chiotta, and Francisco Bavera. "An user configurable clang static analyzer taint checker". In: *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE. 2016, pp. 1–12.

[8] Kristóf Umann and Zoltán Porkoláb. "Detecting uninitialized variables in C++ with the Clang Static Analyzer". In: *Acta Cybernetica* 25.4 (2022), pp. 923–940.

[9] Zhongxing Xu, Ted Kremenek, and Jian Zhang. "A Memory Model for Static Analysis of C Programs". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 535–548. ISBN: 978-3-642-16558-0.

[10] haoNoQ. *clang-analyzer-guide*. [Online; accessed 02-January-2023]. 2016. URL: https://github.com/haoNoQ/clang-analyzer-guide/.

[11] Ted Kremenek. "Finding software bugs with the clang static analyzer". In: *Apple Inc* (2008).

[12] A. Dergachev. *2019 LLVM Developers' Meeting: A. Dergachev "Developing the Clang Static Analyzer"*. [Online; accessed 29-December-2022]. 2019. URL: https://www.youtube.com/watch?v=g0Mqx1niUi0.

[13] CSmith. *Csmith, a random generator of C programs*. [Online; accessed 29-December-2022]. 2022. URL: https://github.com/csmith-project/csmith.

[14] ISO C++ Standards Committee. *C++ standard draft*. [Online; accessed 29-December-2022]. 2022. URL: https://github.com/cplusplus/draft.

[15] ISO C++ Standards Committee. *Programming Languages — C++*. [Online; accessed 29-December-2022]. 2022. URL: https://isocpp.org/files/papers/N4860.pdf.

[16] Open STD. *C - Project status and milestones*. [Online; accessed 29-December-2022]. 2022. URL: https://www.open-std.org/jtc1/sc22/wg14/www/projects#9899.

Appendix A

Building clang

Having a custom build of clang under hand is very convenient (and sometimes necessary) to run the commands in this report. In particular, we worked with LLVM 15 (the last published version, while the last version available on the Arch repository is 14 and the development version is 16) and we do not guarantee that our work will work with any other version than 15.

Furthermore, if you want to generate the exploded graph (see section 3.2, among others) of an example to visualize what is going on inside CSA, you will need to have a clang version built with debug assertions, and if you want to enable crosschecking with z3 (see section 3.7) then you will need to build with z3.

The commands below will clone the repository, checkout to LLVM 15 version, then prepare to build for x86 platforms a release version of clang with debug info, with assertions and z3 enabled and lastly build using Unix Makefiles (with 8 threads).

Listing A.1: Building a custom clang

```
$ git clone https://github.com/llvm/llvm-project.git
$ cd llvm-project
$ git checkout llvmorg-15.0.0
$ mkdir build install
$ cd build
$ cmake -DLLVM_TARGETS_TO_BUILD=X86 -DCMAKE_BUILD_TYPE=RelWithDebInfo
    -DCMAKE_INSTALL_PREFIX=$(pwd)/../install
    -DLLVM_ENABLE_PROJECTS="clang;compiler-rt" -G "Unix␣Makefiles"
    -DLLVM_ENABLE_ASSERTIONS=ON -DCMAKE_CXX_FLAGS=${CMAKE_CLANG_FLAGS}
    -DLLVM_ENABLE_Z3_SOLVER=ON -DBUILD_SHARED_LIBS=ON ../llvm
$ make -j8
```