



EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPARTMENT OF PROGRAMMING LANGUAGES  
AND COMPILERS

# Uncovering Hidden Dependencies: Constructing Intelligible Path Witnesses using Dataflow Analyses

*Supervisor:*

Porkoláb Zoltán

associate professor, habil. PhD

*Author:*

Umann Kristóf

computer science MSc, started in 2020

*Budapest, 2023*

## Abstract

Static analysis is frequently used to find errors or code smells automatically without executing the program. Although static analysis tools provide early feedback, understanding and fixing complex programming errors cannot be automated. This forms a bottleneck of the most expensive and least available resource, a human expert. For this reason, the clarity and completeness of error reports generated by the analysis tool have a significant impact on the effectiveness of the process.

This paper presents novel techniques to improve the quality of bug reports for static analysis tools that employ path-sensitive program analysis. With the combination of data and control dependency analysis, we can identify the relevance of particular code snippets that were previously missing from the report. We demonstrated the benefits of our approach by implementing a new bug report generator algorithm into the open-source LLVM/Clang Static Analyzer. We show examples from widely used industrial projects written in C or C++ programming language, where the clarity of the bug report improved drastically. Our solution has been released and enabled by default since Clang 10.0.0.

**Keywords:** *static analysis, symbolic execution, control dependency analysis, reaching definitions analysis, Clang Static Analyzer, report generation, code comprehension*

# Contents

|          |                                                                                                |           |
|----------|------------------------------------------------------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                                                                            | <b>3</b>  |
| <b>2</b> | <b>Technical background</b>                                                                    | <b>6</b>  |
| 2.1      | Symbolic execution . . . . .                                                                   | 6         |
| 2.2      | The ExplodedGraph . . . . .                                                                    | 8         |
| <b>3</b> | <b>Report generation</b>                                                                       | <b>9</b>  |
| 3.1      | Goals . . . . .                                                                                | 9         |
| 3.2      | Report generation techniques prior to our research . . . . .                                   | 10        |
| 3.2.1    | Bug path visitors . . . . .                                                                    | 10        |
| 3.2.2    | Interestingness propagation . . . . .                                                          | 11        |
| 3.2.3    | Variable tracking . . . . .                                                                    | 11        |
| 3.3      | Deficiencies . . . . .                                                                         | 11        |
| 3.3.1    | Figure 3.4a.: Control dependency is not recognized . . . . .                                   | 15        |
| 3.3.2    | Figure 3.4b.: Reaching definition is not in the bug path . . . . .                             | 15        |
| 3.3.3    | Figure 3.4c.: Reaching definition is in a different, but inlined<br>stack frame . . . . .      | 15        |
| 3.3.4    | Figure 3.4d.: Reaching definition is in a different and not in-<br>lined stack frame . . . . . | 16        |
| <b>4</b> | <b>Proposed solution</b>                                                                       | <b>17</b> |
| 4.1      | Control Dependence Analysis . . . . .                                                          | 18        |
| 4.1.1    | Defining Control Dependence . . . . .                                                          | 18        |
| 4.1.2    | Integration of control dependence . . . . .                                                    | 19        |
| 4.2      | Data Dependence Analysis . . . . .                                                             | 20        |
| 4.2.1    | Dataflow analyses . . . . .                                                                    | 21        |
| 4.2.2    | Reaching definitions analysis . . . . .                                                        | 22        |
| 4.2.3    | Integration of reaching definitions . . . . .                                                  | 23        |

|          |                            |           |
|----------|----------------------------|-----------|
| <b>5</b> | <b>Results</b>             | <b>25</b> |
| 5.1      | Notable examples . . . . . | 26        |
| 5.1.1    | Example 1. . . . .         | 26        |
| 5.1.2    | Example 2. . . . .         | 30        |
| 5.1.3    | Example 3. . . . .         | 33        |
| <b>6</b> | <b>Related work</b>        | <b>36</b> |
| <b>7</b> | <b>Future work</b>         | <b>38</b> |
| <b>8</b> | <b>Conclusion</b>          | <b>40</b> |
|          | <b>Bibliography</b>        | <b>42</b> |
|          | <b>List of Figures</b>     | <b>48</b> |

# Chapter 1

## Introduction

Maintenance costs take a larger part of the price of the software systems. Most of these expenses are spent fixing bugs. The earlier a bug is detected, the lower the cost of the fix [1]; therefore, various efforts are applied to speed up the *development–bug detection–bug fixing* cycle. The classical test-based approach – although still important – is insufficient on its own. Writing meaningful tests requires high code coverage and takes substantial development workload and time. Also, certain corner cases are easily missed by test writers – and those are the same corner cases where the developers likely made errors. Another approach, the *dynamic analysis method* using tools like *Valgrind* [2], or *Google Address sanitizer* [3] which work runtime, evaluates the correctness only those parts of the system which have been executed. Although such *dynamic analysis methods* are precise and could catch real errors with a very low rate of false reports, they require carefully selected input data, and can also easily miss certain corner cases. Dynamic analysis trades coverage for precision, and reaching even a close to full coverage is usually infeasible.

Contrary to testing and dynamic analysis methods, *static analysis* works at compile-time, based only on the program’s source code, and does not require any input data. It is a popular method for finding bugs and code smells [4, 5, 6, 7]. They do not depend on the selection of input data while they can (at least theoretically) provide full coverage of the code. Compiler warnings are almost exclusively based on various static analysis methods. Many of the applied techniques are fast enough to be integrated into the Continuous Integration (CI) loop, therefore, they have a positive impact on speeding up the development–bug detection–bug fixing cycle. Another advantage of the static analysis method is that it is in many cases

applicable for parts of the code. This is useful when we have no full control over the system, e.g. we use third party libraries, not all source is available, or we just have no resources to check the whole system.

Most static methods apply heuristics, which means that often they may *underestimate* or *overestimate* the program behavior [8]; in other words: static analysis trades precision for coverage. In practice, this means static analysis tools sometimes do not report existing issues which situation is called a *false negative*, and sometimes they report correct code erroneously as faulty, which is called as *false positive*. There is a continuous struggle to improve the tools and methods, but there is a theoretical limitation: paraphrasing Rice’s theorem [9] from ’53: all non-trivial properties of a program are undecidable at compile time. Therefore, at the end all reports need to be reviewed by a professional who has to decide manually whether the report stands, and if so, fix it. This, however, creates a bottleneck in the development process as humans who are experts both in the problem domain and in the implementation techniques are usually the most expensive and the least available resources. It has the uttermost importance to maximize the effectiveness of this step. Considering the mentioned theoretical limitations, the best possible way to do it is to improve the communication between the automated analysis tool and the human actor: i.e., to teach the analysis tool to provide sound, concise and comprehensive reports.

While more complex static analyses can detect even deep-rooted programming errors early, the construction of intelligible bug reports also gets much more difficult. In this paper, we present the challenges faced by a technique called symbolic execution. Symbolic execution explores a high number of execution paths within the program and can constrain the values of runtime variables, allowing it to gain a considerable understanding of the program’s runtime behavior. However, after finding a bug, it struggles to relate back to the source code and is unable to consider a context broader than the path of execution leading to the bug.

We discuss two new techniques to allow a symbolic execution tool to better understand of code contexts outside a given path of execution. We also demonstrate one of these techniques in the open-source analyzer tool Clang Static Analyzer. As one of the more mature and popular static analyzer tools that implement symbolic execution for C, C++ and objective C languages, it is considered stable and reliable to be used on large code-bases for both academic and industrial purposes. We implemented our solution as a part of Google Summer Code 2019 project [10], which is

now a part of the Clang Static Analyzer since version 10.0.0. As this improvement has been enabled in the releases since<sup>1</sup>, we feel there is a real world benefits to our results.

This paper is structured as follows. In Chapter 2. we overview the technical background relating to symbolic execution and how the Clang Static Analyzer implements it. In Chapter 3. we discuss our expectations for an intelligible bug report, and present techniques to generate them and their shortcomings. Chapter 4. details our proposals and implementations. We evaluate our initial prototype in Chapter 5. Related work is surveyed in Chapter 6. Future areas of research and implementation are discussed in Chapter 7. Finally, we conclude our paper in Chapter 8.

---

<sup>1</sup>As of the writing of this paper, the latest Clang release is 13.0.0.

# Chapter 2

## Technical background

An often celebrated advantage of static analysis is its greater code coverage compared to most dynamic analyses. However, this does not come without a cost; arguing about runtime values is often difficult or impossible with only static information. More complex analyses also tend to be expensive in terms of computing resources, and are often several times slower than compilation and consume more memory.

Various techniques approach these challenges from different angles – abstract syntax tree analysis (*AST analysis*) [11] and control flow analysis trade understanding of runtime behavior for faster analysis speed. dataflow analyses [12] are able to argue about the flow of information within the bounds of a given function, and most variants strike a middle ground with in terms of space and time complexity and the effectiveness of the analysis. Symbolic execution [13] takes a rather radical approach, by essentially interpreting the source code, and analyzes a large number of execution paths in the program. This leads to a combinatorial explosion according to the number of possible program states, which makes the analysis rather expensive, but awards tremendous information about runtime behavior.

This chapter discusses symbolic execution in terms of its operation, its implementation in the Clang Static Analyzer [14], and its bug report generation facilities.

### 2.1 Symbolic execution

Concrete execution, what we would consider the “normal” execution of the program or the simulation of such, is done by a specific input to explore a single path

of execution. In contrast, most forms of symbolic execution need no input values and explore multiple paths of execution, covering entire classes of inputs [15]. These input values, and runtime variables of the program are assigned *symbolic* rather than concrete values. An analysis engine models the program behavior with a store, a mapping of variables to symbolic values, and a constraint solver, which contains constraints on symbolic values [16, 17]. The store is updated when a memory location is written, and the constraint solver is updated after each evaluation of a conditioned branch.

For instance, an analyzer tool may map the symbolic value \$1 to as the value of the parameter variable *ptr*. Upon reaching the condition *if(ptr)*, the analyzer would explore a path of execution where the condition is true, imposing the constraint  $\$1 \in (-\infty, 0) \cup (0, +\infty)$ , and a path of execution where the condition is false, imposing the constraint  $\$1 = 0$ . In a later assignment  $qtr = ptr$ , *qtr* would also be mapped to \$1. Later, upon reaching the condition *if(qtr)*, the analyzer will know precisely whether the condition is true or false by inspecting the constraint on \$1.

The Clang Static Analyzer [14, 18, 19, 20] is an open-source tool that implements symbolic execution on the C family of languages. Over the course of a decade, it grew to be regarded as a stable and reliable tool for academic and industrial purposes. It enjoys a variety of advantages of being built directly into the Clang compiler, such as a thoroughly tested and up-to-date abstract syntax tree (*AST*) and control flow graph (*CFG*, pictured in Figure 3.1.). Clang [21] is a compiler frontend for LLVM, the umbrella project that offers a wide selection of algorithms that are useful for optimization. For the remainder of the paper, refer to the Clang Static Analyzer under the term *analyzer*.

Static analysis in clang starts after the conclusion of syntactic and semantic analysis, and the construction of the AST and the CFG. The analyzer then creates a call graph for the input file’s translation unit. The call graph’s nodes are functions, and directed edges describe function invocations from one function to another. If possible, symbolic execution starts from functions with no ingoing edges in the call graph, otherwise in some other function. From this initial function, called the *top-level function*, the analyzer will explore paths of execution using the CFG. For the code snippet in Figure 3.1., *f* will be the top-level function, and a possible path of execution would be *B5 B4 B3 B2 B0*. The CFG describes one specific function at a time yet both *B4* and *B2* contain function calls. This obstacle is resolved by

the analyzer “jumping” from the invocation site to the entry blocks of the callee function’s CFG. We call this process *inlining*. Should we denote the symbol  $f_{oo_{B_i}}$  as the  $i$ th block of `foo`’s CFG, the path of execution mentioned above is as follows:

$$f_{B5} \ f_{B4} \ g_{B3} \ g_{B2} \ g_{B1} \ f_{B4} \ f_{B3} \ f_{B2} \ g_{B3} \ g_{B2} \ g_{B1} \ f_{B2} \ f_{B0}$$

## 2.2 The ExplodedGraph

During analysis, the analyzer builds its data structure to keep track of the program state (most notably the store and the constraint manager) at each point of symbolic execution. This data structure is called the **ExplodedGraph**, which is pictured in Figure 3.3 [22]. The ExplodedGraph is a different data structure to the CFG because it contains far more information (assumptions on values, memory regions). It isn’t even isomorphic with it; a path of execution on which the body of a loop is visited four times, each visit would be represented with a linear path, instead of a directed loop. Also, while a CFG is built for each function, only a single ExplodedGraph is built for the entire analysis.

# Chapter 3

## Report generation

A frequently researched problem of static analysis is to discover as many real bugs as possible while keeping their false positive rate within a margin of error[23, 24]. However, making the generated reports intelligible and easily digestible is rarely discussed. In this chapter, we define a non-comprehensive set of guidelines on what an ideal bug report is, and overview how the analyzer approaches this issue, and its struggles relating to the limitations of the ExplodedGraph.

Bug report generation is done after the entire analysis is concluded by the inspecting the nodes of the ExplodedGraph. To avoid confusion, we define the following terms:

- An *(explored) path of execution* is a directed path in the ExplodedGraph starting from the root terminating in one of its leaves.
- A *bug path* is the shortest path of execution, which terminates in an error node. Error nodes in the ExplodedGraph are nodes where a bug was discovered.
- A *bug report* is a user-readable set of messages and notes that explains the control flow leading to the bug, and the values of related variables.

### 3.1 Goals

The bug path is a collection of all events on a given path of execution and is not a user-readable set of events. Some nodes describe relatively low-level actions, like an lvalue-to-rvalue cast, the cleanup of local variables, or other events that may not be relevant to the actual bug. Hence, we define the ideal bug report to be:

- *Minimal*, so that it is void of events that are irrelevant to the comprehension of the bug report,
- *Complete*, so that it highlights every relevant event.

In a sense, an ideal bug report tells how to reproduce the bug. Unfortunately, these goals are rather vague and leave room for subjectivity. For instance, control flow through a `constexpr`-conditioned branch is obvious from the compiler’s perspective but *may or may not be* obvious to a reader. To keep our study free of personal preference, we precisely define the vaguest word of these goals: relevance.

We say that statement  $a$  is relevant to statement  $b$ , if for a given  $(b, \{set\ of\ variables\})$  pair,  $a$  is a control dependency of  $b$ , or contains an expression that is a data dependency of any of the variables in the pair’s set. We call a  $(statement, \{set\ of\ variables\})$  pair a *slicing criterion* [25].

## 3.2 Report generation techniques prior to our research

To construct a bug report, the analyzer, starting from the error node, inspects the bug path’s nodes individually all the way to the root of the `ExplodedGraph`. At each node, it will look for interesting events worth explaining to the user. Two techniques are employed for each of the goals mentioned above. *Bug path visitors* add user-readable messages and notes to the final bug report, and *interestingness propagation* helps to discard a portion of these.

### 3.2.1 Bug path visitors

Contrary to their name, bug path visitors function as callbacks. As the analyzer visits a new node in the bug path, it notifies each visitor to inspect it. If a visitor finds a noteworthy event, they may construct a diagnostic message. For instance, `ConditionBRVisitor` constructs a message for each evaluated condition (e.g., “Assuming the condition is true”).

Visitors greatly expand the number of variables and values to consider for explaining. Data dependence is a great example; if variable  $x$  caused a bug, we could register `FindLastStoreBRVisitor`, to find  $x$ ’s last write preceding the error node

(e.g., “Value assigned to ‘x’ ”). If that last write is in the form of an assignment (`x = y;`), a new instance of `FindLastStoreBRVisitor` may be registered to explain `y`.

### 3.2.2 Interestingness propagation

By their design, visitors cannot always be aware of whether the constructed message is relevant to the bug report. In anticipation, the analyzer marks some entities (such as the denominator for a division-by-zero bugs) interesting. As the analyzer ascends the bug path, visitors may propagate this interestingness to other entities. In a later stage, after all diagnostics have been constructed, messages in function calls not describing any interesting entity are pruned.

### 3.2.3 Variable tracking

A common desire for bug report generation is to explain all events relating to a variable: why it holds a specific value, control flow around the usages of said variable, and other properties. This is achieved by registering a set of visitors relating to that variable, and mark it interesting. We call this process the *tracking* of said variable.

## 3.3 Deficiencies

Even when describing multiple iterations of a loop, bug paths contain no directed cycles. They are a sequence of program states, leading to a node where the program state is erroneous. This linearity and the lack of information on code that was not explored by the analyzer on that bug path make it challenging to find relevant statements.

Figure 3.1. shows a code snippet where the global variable `flag` controls whether `x` will be initialized, and whether `x` will be dereferenced. Function `g` sets `flag` to some unknown value, and the lack of parameter passing makes this a non-trivial realization from a user’s perspective. Figure 3.2a. shows a report from the analyzer displayed by CodeChecker [26] before our research: the analyzer failed to understand that `x`’s value, and its dereference depends on `flag` and is worth explaining. As a result, it pruned diagnostic messages relating to function calls to `g`. In a later section, we will discuss our results to improve this bug report as shown in Figure 3.2b.

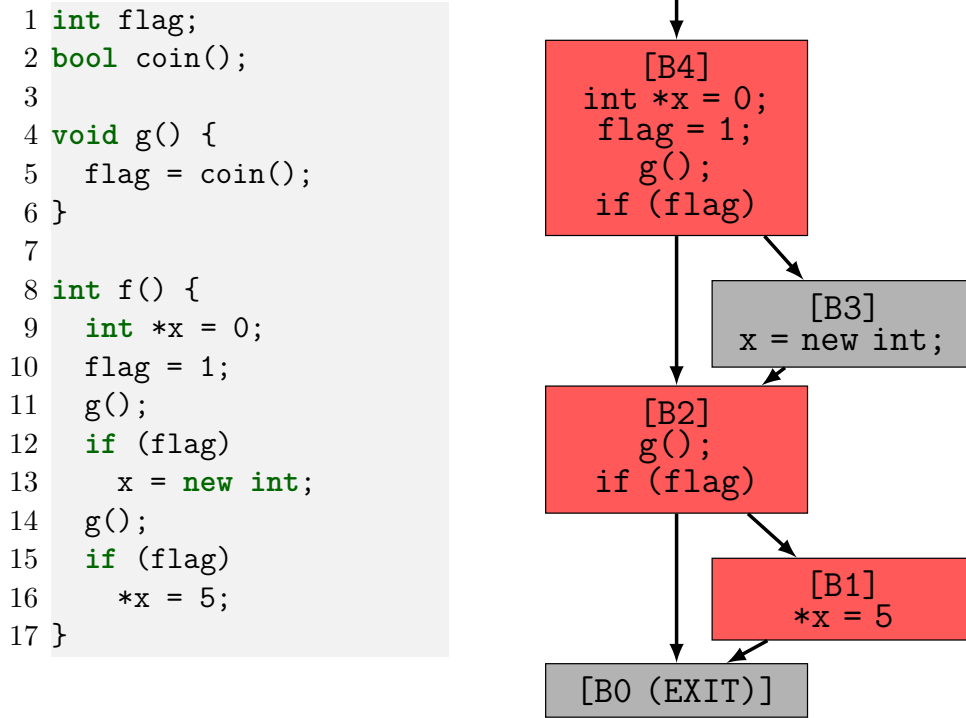


Figure 3.1: A code snippet and function `f`'s control flow graph.

For each example in Figure 3.4., the analyzer can discover a null dereference bug on line 21, but will also fail to find all relevant statements to it during bug report construction. These examples correspond to four classes of problems, which we discuss in further detail as follows:

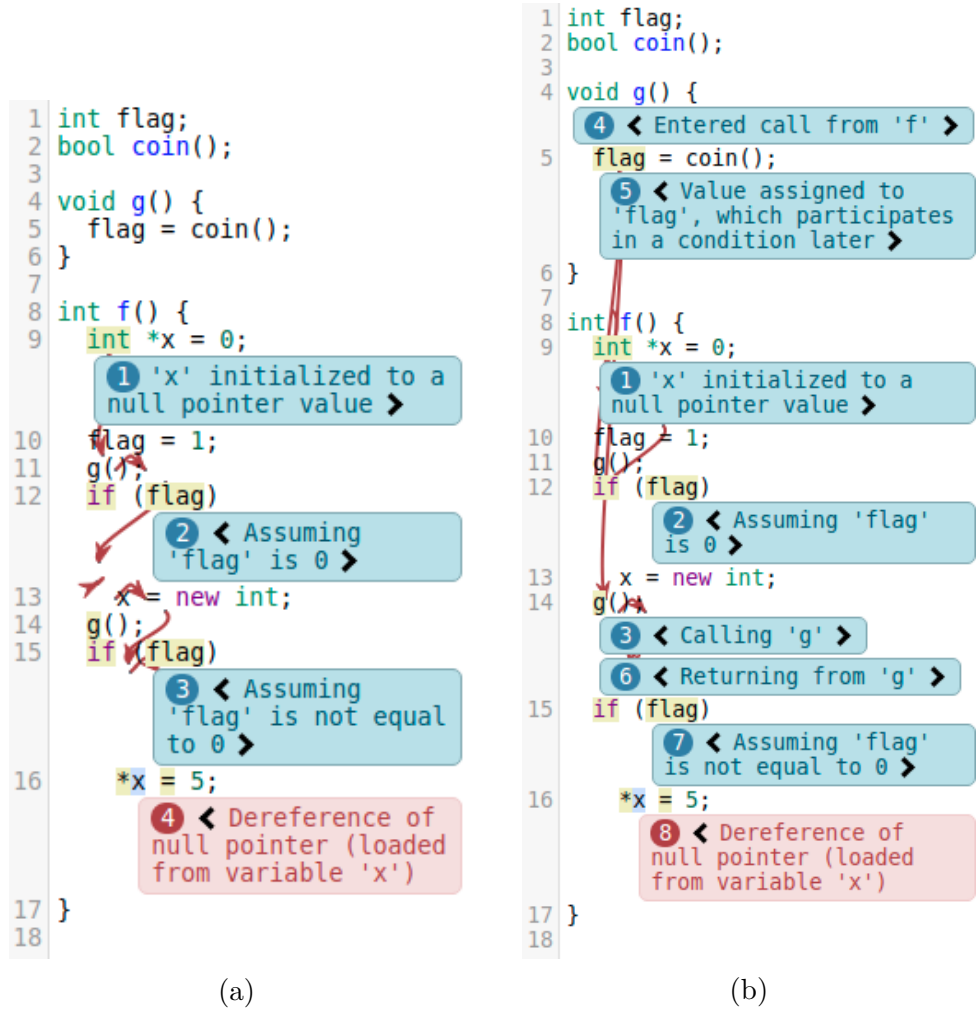


Figure 3.2: A code snippet and analysis results demonstrating how the analyzer struggles to realize that  $x$ 's value and dereference depends on  $flag$ , and as a result, will not construct diagnostic messages to explain relevant events to it.

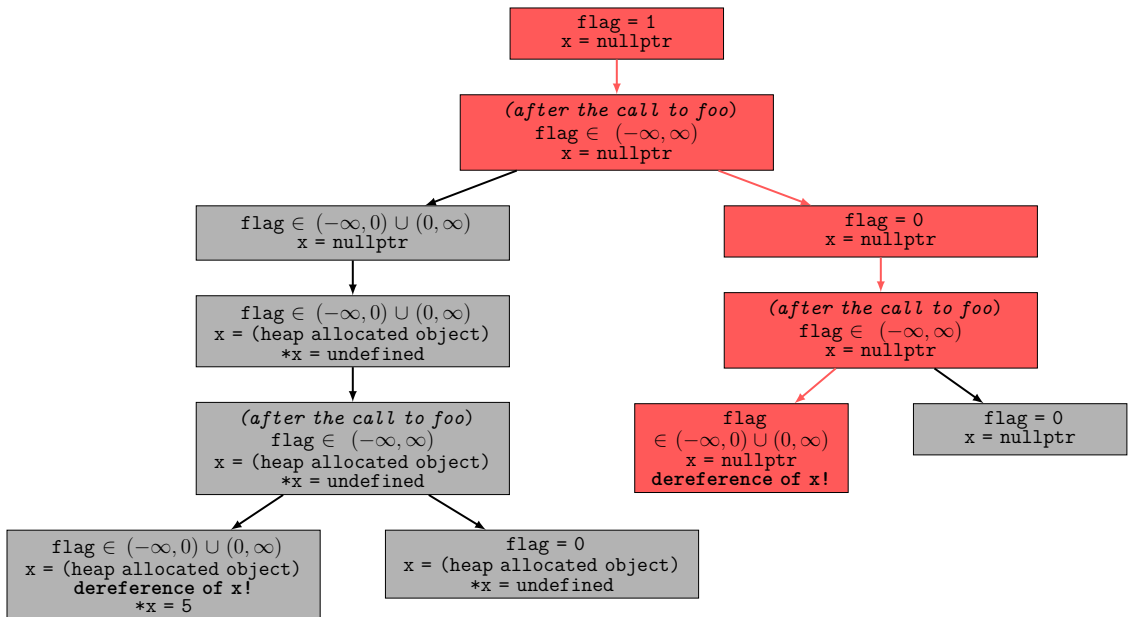


Figure 3.3: A simplified ExplodedGraph after analyzing Figure 3.1.

```

1 int flag;
2 bool coin();
3
4 void g() {
5     flag = coin();
6 }
7
8
9
10
11
12
13 int f() {
14     int *x = 0;
15     flag = 1;
16     g();
17
18
19
20     if (!flag)
21         *x = 5;
22 }

```

(a)

```

1 int flag;
2 bool coin();
3
4 void g() {
5     flag = coin();
6 }
7
8
9
10
11
12
13 int f() {
14     int *x = 0;
15     flag = 1;
16     g();
17     if (flag)
18         x = new int;
19
20
21     *x = 5;
22 }

```

(b)

```

1 int flag;
2 bool coin();
3
4 void g() {
5     flag = coin();
6 }
7
8 void h(int **x) {
9     if (flag)
10         *x = new int;
11 }
12
13 int f() {
14     int *x = 0;
15     flag = 1;
16     g();
17     h(&x);
18     g();
19     if (flag)
20         *x = 5;
21
22 }

```

(c)

```

1 int flag;
2 bool coin();
3
4 void g() {
5     flag = coin();
6 }
7
8 void h(int **x) {
9
10     *x = new int;
11 }
12
13 int f() {
14     int *x = 0;
15     flag = 1;
16     g();
17     if (flag)
18         h(&x);
19     g();
20     if (flag)
21         *x = 5;
22 }

```

(d)

Figure 3.4: Code snippets that highlight deficiencies in the analyzer’s understanding of bugs when generating reports.

### 3.3.1 Figure 3.4a.: Control dependency is not recognized

The bug path associated with the bug starts at line 14, assigning `x` to a null pointer, and setting the global `flag` variable to 1. The bug path will contain the function call to `g()`, and appropriately set `flag`'s value to unknown. Then, on line 20, the analyzer will explore a path of execution on which `flag`'s new value is 0, and one where it is not. Naturally, the bug path will describe the former, concluding on line 21, where the dereference of `x` is found, which is known to be null. The analyzer will configure its bug report generation facilities to start tracking `x`.

During bug report generation, the analyzer can find the relevant statement to `x` regarding data dependencies, which is its initialization on line 14. It will, however, fail to recognize a relevant statement to the bug – namely, had `flag` not been 0 on line 20, the bug would not have occurred. Note that `flag`'s value was set to 1 a few lines earlier, and should `g()`'s definition be unavailable or obscured, it would not be obvious why `flag`'s value is assumed to be 0 on line 20.

### 3.3.2 Figure 3.4b.: Reaching definition is not in the bug path

The bug path is constructed similarly to Figure 3.4a. The analyzer can again find `x`'s initialization as relevant, but as line 18 is not on the bug path, the analyzer fails to recognize that the user likely intended to set `x`'s value properly. This assignment, alongside `x`'s initialization, are so-called *reaching definitions* to `x` on line 21. `flag`'s value on line 17 is no longer a control dependency to the CFG block in which the bug is found, yet it is clear that should `flag`'s value be non-zero on line 17, the bug would not have occurred. Since line 18 is not known to be relevant, line 17 as its control dependency is not recognized either.

### 3.3.3 Figure 3.4c.: Reaching definition is in a different, but inlined stack frame

This example presents another layer of difficulty to Figure 3.4b. – the statement on which `x` could have obtained a non-null value is not only outside the bug path, but is in another function call. The problem goes outside the bounds of a single function call, and this time the analyzer is unable to bridge this gap, requiring an interprocedural solution to discover this relevance. Although symbolic execution is

interprocedural, many lightweight techniques that could supplement its bug report generation facilities are not.

### **3.3.4 Figure 3.4d.: Reaching definition is in a different and not inlined stack frame**

A relevant statement, line 10, is not only outside the bug path, but the containing function `g` was not inlined either. Inlining functions can demand non-trivial modeling, such as lifetime extension, moves, and the evaluation of arguments. This makes discovering relevant statements even more difficult. Generally speaking, the “further” the analyzer has to stray from the bug path, the more challenging bug report construction becomes.

# Chapter 4

## Proposed solution

Program slicing is a field of study in computer science about creating a *program slice*, which is a subset of the program's statements, relevant to a point of interest, usually defined by a (statement, {set of variables}) pair, called a *slicing criterion*. Relevance in this context is defined by whether a statement could influence the value of one of the variables in the slicing criterion. Program slicing combines data and control dependency analysis in a fix-point algorithm to slice irrelevant statements away from the program, converging to a minimal, but complete slice.

The original program slicing algorithm [25] was intraprocedural, and aimed at monolithic, single-procedure programs [27]. Interprocedural variants are explored in explored in numerous studies [28, 29], but they demand the existence of a data structure that describes data and control dependencies across function calls, most commonly a system dependence graph, which at the time of writing was absent from Clang, and its implementation would be a challenging task with Clang's AST and CFG design.

As feasible implementations of slicing algorithms are confined to the bounds of a single function, and most bug paths span multiple functions in the source code, additional techniques are required to make program slicing a valuable part of Clang's bug report generation facilities. Also, as discussed in Section 3.2.1., bug reporter visitors can reason about data dependencies with rather great precision; incorporating this information would also be a meaningful improvement. For these reasons, we approached slicing in terms of its core components, not in its entirety.

In Section 3.3., we have shown four classes of problems the analyzer could not tackle prior to our research. We propose two techniques as a potential solution, as

well as how they can be incorporated into the existing bug report generation infrastructure: control dependency analysis and reaching definitions analysis. While we are cautious about unforeseen challenges, we feel confident that these would solve three of the four cases, and pave the way to approach the fourth. As a demonstration, we implemented control dependency analysis and observed measurable improvements for the first case.

## 4.1 Control Dependence Analysis

On most occasions, control dependence center around conditional statements (e.g., `if`, `for`, `switch`), where the value of the condition dictates which part of the code (e.g., branches of `if` statements) will be executed next. For instance, cases of a switch-case statement is control dependent on the switch statement.

We defined relevance in a way that slice  $a := (stmt_a, vars_a)$  is relevant to slice  $b := (stmt_b, vars_b)$ , if  $stmt_a$  is a control dependency of  $stmt_b$ . Suppose that we constructed  $b$  after discovering a null pointer dereference error (for instance,  $(line16, \{x\})$  for Figure 3.1), and have already marked  $stmt_b$  as interesting. We argue that regarding  $stmt_a$  as relevant is beneficial for bug report construction; had control not flown from  $stmt_a$  to  $stmt_b$ , the bug would not have occurred.

In this section, we overview our proposal and implementation of adding control dependency analysis to bug report construction.

### 4.1.1 Defining Control Dependence

We define control dependence on the CFG with the help of *dominance* and *post-dominance*. We say that block (a node of the CFG)  $A$  *dominates* block  $B$  ( $A \text{ dom } B$ ), if every path from the entry block to  $B$  must go through  $A$ . We say  $A$  *strictly dominates*  $B$  ( $A \text{ sdom } B$ ), if  $A$  dominates  $B$  and  $A \neq B$ . We say  $B$  *postdominates*  $A$  ( $B \text{ pdom } A$ ), if every path from  $A$  to the exit block must go through  $B$ , and again, similarly,  $B$  *strictly postdominates*  $A$  ( $B \text{ spdom } A$ ) if  $B$  postdominates  $A$  and  $B \neq A$  [30].

For CFG seen in Figure 3.1, the entry block (strictly) dominates every (other) block, and the exit block (strictly) postdominates every (other) block.  $B4$  dominates

$B3$ , but  $B3$  doesn't dominate  $B2$ , since the path  $B5 \rightarrow B4 \rightarrow B2$  excludes  $B3$ .  $B2$  postdominates  $B4$ , but  $B3$  doesn't postdominate  $B4$ .

We say that block  $B$  is *control dependent* on block  $A$  ( $B \text{ cd } A$ ) if there exists an edge from  $A$  to  $C$  such that  $B$  postdominates  $C$ , and if  $B$  isn't equal to  $C$ ,  $B$  doesn't postdominate  $A$ . In looser terms, this expresses that  $B$  is control dependent on  $A$  if  $B$  doesn't postdominate  $A$ , but post dominates all blocks “in between them”. As an example, in Figure 3.1,  $B1$  is control dependent on  $B2$ , but  $B0$  is not control dependent on  $B2$ , as  $B0$  post dominates  $B2$ . We extend control dependency to statements as follows: If block  $B$  is control dependent on block  $A$ , we also say that all statements in  $B$  are *control dependent* on the condition expression in  $A$ , if such exists.

Control dependencies can be calculated with *post dominance frontier* sets [31]. The post dominance frontier set of block  $A$  ( $PDF(A)$ ) is the set of  $B$  nodes from the inverse CFG<sup>1</sup> such that  $A$  dominates a predecessor of  $B$  but does not strictly dominate  $B$ :

$$PDF(A) := \{B | (\exists P \in Pred(B)) \wedge (A \text{ pdom } P \wedge \neg A \text{ spdom } B)\}$$

Calculating PDF sets quickly yields control dependence:

$$A \text{ cd } B \Leftrightarrow B \in PDF(A)$$

We implemented dominance frontier sets with the algorithm described in [32].

### 4.1.2 Integration of control dependence

As bug path visitors continuously expand the code contexts (values, variables) to explain, we chose to weave our control dependency calculator into a new visitor. A new instance of our visitor is registered *for each* new tracked expression value. As a new node in the bug path is visited, the visitor checks whether the statement described in the node is a control dependency of the tracked value. If so, it will instruct the analyzer to track the condition of that statement.

---

<sup>1</sup>An inverse of a CFG is constructed by reversing all of its edges in the graph.

Figure 3.4a. demonstrates a code snippet where the analyzer can detect a null pointer dereference of `x`, but fails to realize that had the value of `flag` may have been a guard of this error. With our improvement, the bug report generation would work as follows: The analyzer would start tracking `x`, registering several visitors, including our own. As it ascends the bug path and finds the `ExplodedNode` describing the evaluation of `!flag` on line 20, our visitor would realize that the basic block that contains this condition expression is indeed a control dependency of `x`. As such, it will start the tracking of `flag`. `FindLastStoreBRVisitor` would find `flag`'s last store on line 5, and a diagnostic message will be constructed about it. Since `flag` is tracked, it is also an interesting variable and the analyzer will prevent the pruning of messages exploring `g()`.

While this information was indeed found on the bug path, inspection of its nodes alone did not reveal the relevance in between `flag` and `x`, and led to the analyzer discarding information about `g()` to keep the bug report minimal. Control dependence analysis unearthed and preserved the importance of this function call.

In application, we employ a number of heuristics to limit the impact of our solution to only display diagnostics when they provide a meaningful addition to the bug report. When displaying a bug report to the user, the source code is decorated with diagnostic messages and notes, and often relevant information about condition values is readily available in the same function as the condition itself. We found that additional notes in the same function did not add much value the user experience, and on occasion needlessly polluted the report. We chose to display new diagnostic messages only when information relating to a condition was found in a function call that would otherwise be disregarded.

## 4.2 Data Dependence Analysis

Control dependence was a property of basic blocks, and by extension, statements within those blocks. We briefly discussed data dependence in the context of a bug path visitor; namely, `FindLastStoreBRVisitor` can find the last store to a variable. As this process is recursive, all transitive data dependencies can be discovered in a *on the bug path*.

Data dependency analysis on the bug path benefits from all the information the analyzer gathered during symbolic execution. Common obstacles in this realm might

already be resolved: the analyzer’s memory model keeps track of pointers and their pointees, and if possible, function calls are inlined and evaluated. The linearity of the bug path makes this kind of analysis also relatively efficient. However, as we have demonstrated before, code outside of what the bug path describes might also be relevant to a bug’s occurrence.

In this section, we discuss a dataflow algorithm called reaching definitions. When inquiring about which parts of the program was meant to affect the value of a bug causing variable, reaching definitions is an elegant solution to find relevant statements. Unfortunately, it is all but absent of the challenges symbolic execution solves, and many others.

### 4.2.1 Dataflow analyses

As the name suggests, control flow analyses describe the flow of control within a program by inspecting the *structure* of the CFG; dataflow analyses complements this by analyzing how information (e.g., values of variables, state of mutexes, whether a value will be read from in a later basic block) flows, changes, and is accessed from one node to another by inspecting the *contents* of the CFG. Many notable dataflow analyses are defined by calculating an initial set of properties for each basic block, and propagating these properties along the edges of the CFG, so that properties “flow” from one block to another. Propagations may be described with dataflow equations; these are then repeatedly solved until these property sets change no more, reaching a fixpoint. Common initial property sets include GEN and KILL. Although their definitions vary, they are similar to the following: for basic block B,  $GEN[B]$  is the set of variables read in B, and  $KILL[B]$  is the set of variables written in B.

As an example, live variable analysis [33] calculates the set of live variables in a given basic block. A variable is live if its value *may* be read in subsequent blocks. Formally, a variable  $x$  is live in block  $i$ , if block  $j$  uses the value of  $x$ , and there exists a path from  $i$  to  $j$  without any interleaving assignments to  $x$ .  $LIVE_{in}[B]$  is the set of variables live at the beginning of block B, and  $LIVE_{out}[B]$  is the set of variables live at the end of B.<sup>2</sup> In Figure 3.1.,  $x$  is live in blocks B4, B3 and B2, but

---

<sup>2</sup>As basic blocks are sequences of operations executed sequentially, they might not be granular enough, as the same variable may be written multiple times in a given block, which would result in a loss of liveness information. This problem can be solved by splitting up such blocks to only contain at most one write of each variable.

not in B1 and B0. It is indeed possible to express liveness with dataflow equations:

$$\begin{aligned} LIVE_{in}[B] &= GEN[B] \cup (LIVE_{out}[B] \setminus KILL[B]) \\ LIVE_{out}[B] &= \bigcup_{S \in succ[B]} LIVE_{in}[S] \end{aligned}$$

This definition *overapproximates* the actual set of live variables. Suppose in Figure 3.1. `g()` is known to always set `flag`'s value to false. Although `x` would be a dead variable throughout the entire function, liveness analysis, and dataflow analyses in general are incapable of telling whether a path of execution in the CFG is feasible.

C/C++ presents several challenges to overcome in calculating GEN/KILL sets. Due to the aliasing problem presented by pointers, it can be difficult or impossible to tell which variables are read or written. Another significant obstacle is posed by function calls, as dataflow analyses are defined to reason about a single CFG at a time. These limiting factors force the analysis to over- or underestimate its results even further. Clang in particular faces a number of additional problems; its CFG design links back to the AST, which very closely represents the source code. This makes an expression such as  $((a, b) = f(ptr))$  require handling of numerous cases to precisely understand. Notable exceptions are Clang's thread safety analysis [34] and lifetime analysis [35, 36, 37, 38].

In LLVM/Clang, most dataflow analyses are used for optimization, which happens after the source code was transformed into LLVM IR, which is largely void of this design problem. Unfortunately, by the time the lowering to LLVM IR is reached, too much information may be lost to relate analysis results back to the original source code. While there are several projects trying to bridge the gap in intermediate representations that is sufficiently low level but can be used for diagnostics construction, at the time of writing, Clang is lacking a robust dataflow infrastructure [39].

### 4.2.2 Reaching definitions analysis

We call the write of variable `x` a *definition* of `x`. Functions calls and aliasing can greatly obscure the nature of a write, so it is important to conservatively regard every statement that *may* write a variable as a definition of said variable. When describing

analyses concerning definitions, we define  $GEN[B]$  sets such that they contain the set of definitions present in B instead (B generates x's definition), and  $KILL[B]$  sets such that they contain every other definitions in the CFG that generate the same the same variables as B.

The ingoing reaching definitions [33] set of B ( $REACH_{in}[B]$ ) is the set of definitions reaching B. The outgoing reaching definitions set of B ( $REACH_{out}[B]$ ) is the incoming set minus the definitions killed by B, as well as the definitions generated by B. We can define reaching definitions with the following dataflow equations:

$$\begin{aligned} REACH_{in}[B] &= \bigcup_{P \in pred[B]} REACH_{out}[P] \\ REACH_{out}[B] &= GEN[B] \cup (REACH_{in}[B] \setminus KILL[B]) \end{aligned}$$

If the set of definitions to x at block B (a subset of  $REACH_{in}[B]$ ) contains no elements, that means x is first defined in B, or not yet defined. If the set contains a single element, that means we can precisely tell which statement defines x's value in B. If it has two or more elements, then x's value might be different if different execution paths are chosen to reach B. In Figure 3.1., if we denote definitions by a (variable, line number) pair, B1's reaching definitions set would be the following: 3

$$REACH_{in}[B1] = \{(flag, 15), (x, 9), (x, 13)\}$$

### 4.2.3 Integration of reaching definitions

The reaching definition set of B1 is very telling: it highlights that the definition of x in B3 reaches the block where x was a cause of a bug, which could be a hint that the developer intention to prevent the bug from occurring. Although the analyzer did not visit B3 on this path of execution and is absent from the bug path, one can tell that a control dependency of B3, namely the condition in B4, is. This realization could trigger the analyzer to start tracking flag on line 12. This would force the creation of diagnostic messages for the last store to flag, which is in a function called on line 11.

With reaching definitions, we would be able to find a point of interest to the bug but outside the bug path, and could instruct the analyzer to explain control flow around these points better. This would theoretically solve the class of problems

demonstrated in Figure 3.4b. The analysis is conservative on which statements are considered definitions, so the analyzer must only consider a statement relevant if no such overapproximation occurred about it. With that said, theoretically, the analyzer might be able to fill the gaps of information dataflow algorithms usually struggle with; suppose a pointer is written right before a division-by-zero error is discovered. Reaching definitions might be forced to conservatively assume that said pointer points to the denominator; however, it could ask the analyzer whether this aliasing is possible, and might be able to disregard the pointer assignment.

The class of problems displayed in Figure 3.4c. is more difficult to detect accurately. Reaching definitions is confined to the bounds of `f`'s CFG, and will not detect `x`'s potential write on line 10. One aspect that makes this case more approachable is the fact that the function call to `h` is present on the bug path, and the analyzer will resolve that the parameter of `h` will alias with `x` in `f`. This won't make reaching definitions interprocedural but would grant a limited toolset to reason across a limited set of functions. Nonetheless, we would need to enhance our reaching definitions algorithm with some pointer aliasing capabilities.

For the last class of problems demonstrated in Figure 3.4d., we lose the ability to ask the analyzer to resolve parameter passing. While reaching definitions would find that `x` might be written on line 18, it can not prove that it will be, and would not instruct the analyzer to explain control flow around it. To reason about `h`, we are forced to reimplement some of the analyzers inlining technology to make reaching definitions to understand more than one function on its own. Should such a technology exist, we would need to survey how deep of a function call chain should we investigate to look for points of interest. This highlights how much more difficult it is to discover relevant information from the program the further we stray from the bug path.

With that said, the concept behind the interaction of reaching definitions with control dependency analysis displays the many of the characteristics of static backward program slicing.

# Chapter 5

## Results

We tested our solution on, as seen in Table 5.1, the following open-source C and C++ projects: Bitcoin [40], CppCheck [41], Gravity [42], LLVM and Clang [43], OpenSSL [44], Rtags [45], S2N [46] and Xerces [47]. Combined, these projects cover a wide variety of coding techniques, codebase sizes, and different versions of the languages’ standards.

The analysis on CppCheck shows a greater number of changes than anticipated; we observed that almost all of these changes were in comparison operators for `std::string`. We would have preferred if the analyzer could create summaries [48] for this class and a specialized message. Although these notes were not as meaningful, they were not intrusive, as they annotated a function call, and embedded notes were easy to skip. For LLVM, we expect that the number of changed reports would be lower after applying LLVM specific checkers [49]. As for the rest of the projects, we found that the overwhelming majority of additional notes made the bug reports

|              | Total reports | Changed reports | % of changed |
|--------------|---------------|-----------------|--------------|
| Bitcoin      | 12            | 0               | 0 %          |
| CppCheck     | 98            | 69              | 70.4 %       |
| Gravity      | 16            | 2               | 12.5 %       |
| LLVM + Clang | 1022          | 389             | 38.1 %       |
| OpenSSL      | 255           | 6               | 2.3 %        |
| Rtags        | 15            | 1               | 6.6 %        |
| S2N          | 29            | 0               | 0 %          |
| Xerces       | 106           | 7               | 6.6 %        |

Table 5.1: Evaluation of control dependency tracking in terms of how many reports received additional notes.

better. We found that the more complex a report is, the more helpful these messages generally became.

Although we are yet to survey broader user experience related to these new messages, the feedback we received so far have been overwhelmingly positive. Our proposal and eventual implementation was welcomed among regular Clang contributors, who have appreciated the benefits of control dependency analysis, reaching consensus to make it an on-by-default feature in 2019, and has been since version 10.0.0.

## 5.1 Notable examples

In this section, we highlight a few bug reports where control dependency tracking made a poor bug report significantly more readable.

While evaluating a large number of reports during static analysis, it is a good idea to read from the bottom up, as the root cause of the bug, for instance the last assignment to a variable before it participates in a division by zero error, may be close to the error point. This means essential part of the bug report might be shorter than the full report. Starting from the bottom allows the user to disregard the first few of the report as non-consequential.

In the following examples, we advise to read the reports from the top down, unless stated otherwise.

### 5.1.1 Example 1.

This example is from the project CppCheck, in the file `lib/symboldatabase.cpp`. On Figure 5.1., the bug report is shown which was generated without our improvement. At one point, variable `tok2` is assumed to be null, which eventually leads to a null dereference bug. To decide whether this report is a true positive, and if so, how to fix it, a good question to ask is “*Are there any interleaving condition points that should’ve prevented the flow of control from reaching a dereference of `tok2` while its null?*”.

This is rather challenging in this bug report: at one point, the local variable `new_scope` is defined, and is already known to be null in the next condition (if the analyzer would have assumed its value on the condition point, it would have placed

a note there, implying that the analyzer learned of its value earlier). Is this because `findScope` unconditionally returns a null pointer, and its effect is only observable on its parameters? If not, why are there no explanations?

On Figure 5.2., we show the relevant part of the bug report, but after our improvements. A pair of new notes appeared on the function call to `findScope`, and leads to its definition. It forwards us to a call to a non-cost member function with the same name. There, we learn that this function can indeed return non-null values, but the analyzer managed to find a path of execution where this function returns null.

Our improvement recognized that `new_scope` is a control dependency to the bug point, and information about it should be presented.

## BEFORE:

```

// skip over template args
while (tok2 && tok2->str() == "<" && tok2->link()) {
    9 < Assuming 'tok2' is null >
    tok2 = tok2->link()->next();
    while (Token::Match(tok2, "::: %name%"))
        tok2 = tok2->tokAt(2);
}

```

⋮  
 <numerous lines of code>  
 ⋮

```

const Token * name = tok->next();
if (name->str() == "class" && name->strAt(-1) == "enum")
    name = name->next();
Scope *new_scope = findScope(name, scope);
if (new_scope) {
    // only create base list for classes and structures
    if (new_scope->isClassOrStruct()) {
        // goto initial '{'
        if (!new_scope->definedType)
            mTokenizer->syntaxError(nullptr); // #6808
        tok2 = new_scope->definedType->initBaseInfo(tok, tok2);
        // make sure we have valid code
        if (!tok2) {
            break;
        }
    }
}

```

⋮  
 <numerous lines of code>  
 ⋮

```

} else if (new_scope->type == Scope::eEnum) {
    if (tok2->str() == "::")
        tok2 = tok2->tokAt(2);
}
new_scope->bodyStart = tok2;
new_scope->bodyEnd = tok2->link();

```

12 < Called C++ object pointer is null  
 For more information see the [checker documentation](#).

Figure 5.1: Bug report before our improvement from CppCheck.

## AFTER:

```

const Token * name = tok->next();
if (name->str() == "class" && name->strAt(-1) == "enum")
    name = name->next();
Scope * new_scope = findScope(name, scope);
if (new_scope) {
    // only create base list for classes and structures
    if (new_scope->isClassOrStruct()) {
        ...
    }
}

Scope * findScope(const Token * tok, Scope * startScope) const {
    return const_cast<Scope *>(this->findScope(tok, const_cast<const Scope *>(startScope)));
}

const Scope * SymbolDatabase::findScope(const Token * tok, const Scope * startScope) const
{
    const Scope * scope = nullptr;
    // absolute path
    if (tok->str() == "::") {
        tok = tok->next();
        scope = &scopeList.front();
    }
    // relative path
    else if (tok->isName()) {
        scope = startScope;
    }
    while (scope && tok && tok->isName()) {
        if (tok->strAt(1) == "::") {
            scope = scope->findRecordInNestedList(tok->str());
            tok = tok->tokAt(2);
        } else if (tok->strAt(1) == "<" && Token::simpleMatch(tok->linkAt(1), "> ::")) {
            scope = scope->findRecordInNestedList(tok->str());
            tok = tok->linkAt(1)->tokAt(2);
        } else
            return scope->findRecordInNestedList(tok->str());
    }
    // not a valid path
    return nullptr;
}

```

Annotations and Call Stack Notes:

- 11 < Calling 'SymbolDatabase::findScope' >
- 18 < Returning from 'SymbolDatabase::findScope' >
- 12 < Entered call from 'SymbolDatabase::createSymbolDatabaseFindAllScopes' >
- 13 < Calling 'SymbolDatabase::findScope' >
- 16 < Returning from 'SymbolDatabase::findScope' >
- 17 < Returning null pointer, which participates in a condition later >
- 14 < Entered call from 'SymbolDatabase::findScope' >
- 15 < Returning null pointer, which participates in a condition later >

Figure 5.2: New notes after our improvement in the report from CppCheck.

### 5.1.2 Example 2.

This example is from the project Xerces in the file `src/xercesc/validators/schema/TraverseSchema.cpp`. Reading the report on Figure 5.3. from the bottom up shows that the first parameter on `reportSchemaError` is dereferenced as a null pointer. Moving up, we can see that the null pointer originated from the caller function’s local variable, `content`. We entered this code block because `simpleTypeRequired` is true, which was set because a chain of conditions, among them the fact that `baseTypeInfo` is non-null, lead to that assignment. Earlier, we can see that the execution was not halted by an exception, in part because `baseValidator` is null (shown by the flow of control, which is visualized by the arrows). `content`’s last store is present in the full the bug report, but we omitted it on this figure.

It is clear why `simpleTypeRequired` is known to be true at the condition point, but as to why were both `baseTypeInfo` and `baseValidator` known to be non-null and null respectively, is not explained by the report. Their definition gives a clue, it is related to `typeInfo`, but how does the analyzer *know* the values returned by those getter functions so precisely?

On Figure 5.4., which displays parts of the bug report after our improvement, we are greeted by new notes explaining what happens inside `processBaseTypeInfo`. Inside the function, we see two variables with the same types that `baseTypeInfo` and `baseValidator` had being initialized to null. Later, `baseComplexTypeInfo`’s value changes, and is assumed to be non-null, while `baseDTValidator`’s value remains unchanged. The last two statements of the function uses setter functions on `typeInfo` with these variables.

Upon reviewing the definition of `baseTypeInfo` and `baseValidator`, we can see that the getter functions they are initialized with pair with these setter functions, explaining how the analyzer knew their precise value.

Our improvement saw that `simpleTypeRequired` is a control dependency to the bug-causing function call, and started tracking it. Its last store was control dependent (in part) by `baseTypeInfo`, which initiated its tracking. `baseValidator` is tracked as it played a part in preventing the program from throwing an exception, but this could have been omitted, as the value of `baseTypeInfo` would have prevented that anyways.

## BEFORE:

```

processBaseTypeInfo(simpleContent, baseName, localPart, uri, typeInfo);
ComplexTypeInfo* baseTypeInfo = typeInfo->getBaseComplexTypeInfo();
DatatypeValidator* baseValidator = typeInfo->getBaseDatatypeValidator();
if (baseValidator != 0 && baseTypeInfo == 0) {
    // check that the simpleType does not preclude derivation by extension
    if ((baseValidator->getFinalSet() & SchemaSymbols::XSD_EXTENSION) == typeInfo->getDerivedBy()) {
        reportSchemaError(simpleContent, XMLUni::fgXMLErrDomain, XMLErrors::DisallowedSimpleTypeExtension,
                           baseName, typeName);
        throw TraverseSchema::InvalidComplexTypeInfo;
    }
}

:

if (baseTypeInfo) {
    if (baseTypeInfo->getContentType() != SchemaElementDecl::Simple) {
        11 < Assuming the condition is true >
        // Schema Errata: E1-27
        if (typeInfo->getDerivedBy() == SchemaSymbols::XSD_RESTRICTION)
            12 < Assuming the condition is true >
            && ((baseTypeInfo->getContentType() == SchemaElementDecl::Mixed_Simple
                13 < Assuming the condition is false >
                || baseTypeInfo->getContentType() == SchemaElementDecl::Mixed_Complex)
                14 < Assuming the condition is true >
                && emptyParticle(baseTypeInfo->getContentSpec())) {
                simpleTypeRequired = true;
            }

        :

        // Schema Errata E1-27
        // Complex Type Definition Restriction OK: 2.2
        if (simpleTypeRequired) {
            reportSchemaError(content, XMLUni::fgXMLErrDomain, XMLErrors::CT_SimpleTypeChildRequired);
            17 < Passing null pointer value via 1st parameter 'elem' >
            18 < Calling 'TraverseSchema::reportSchemaError' >
            throw TraverseSchema::InvalidComplexTypeInfo;
        }

        :

void TraverseSchema::reportSchemaError(const DOMELEMENT* const elem,
    19 < Entered call from 'TraverseSchema::traverseSimpleContentDecl' >
    const XMLCh* const msgDomain,
    const int errorCode) {
    fLocator->setValues(fSchemaInfo->getCurrentSchemaURL(), 0,
        ((XSDElementNSImpl*) elem)->getLineNo(),
        20 < Called C++ object pointer is null
        For more information see the checker documentation.
        ((XSDElementNSImpl*) elem)->getColumnNo());
    fXSDErrorReporter.emitError(errorCode, msgDomain, fLocator);
}

```

Figure 5.3: Bug report before our improvement from Xerces.

## AFTER:

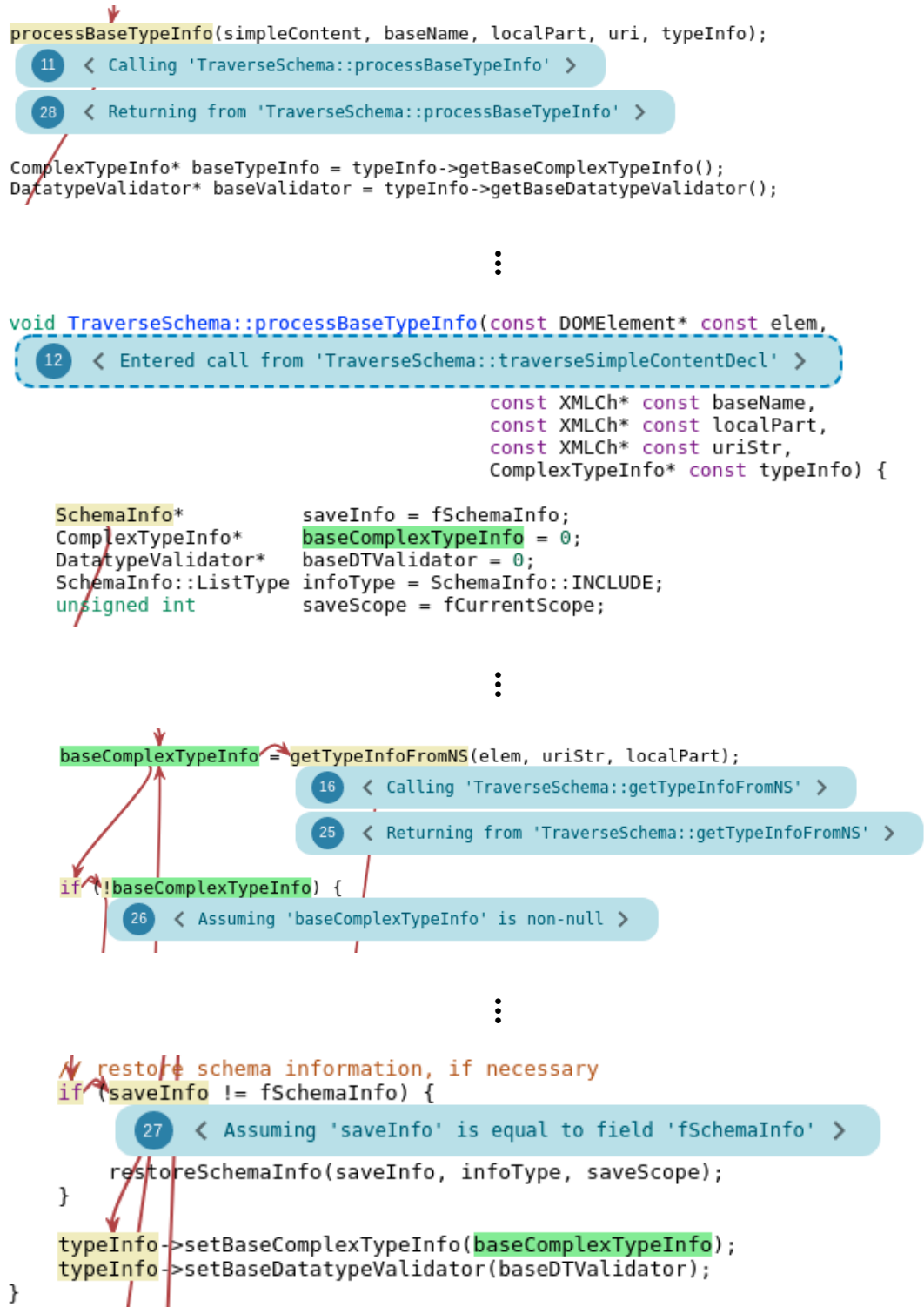


Figure 5.4: New notes after our improvement in the report from Xerces.

### 5.1.3 Example 3.

This example is from LLVM, in the file `clang/lib/CodeGen/CGObjCGNU.cpp`. On Figure 5.6., we see a bug report before our solution. Reading from the bottom up, we see that `OID` is dereferenced as a nullpointer. In a branch inside a range-based for loop, we see the only statement that could have written this variable before its definition. `II` seems to play an important role in the retrieved range, which is initialized based on the parameter of the function, `Name`. Following the flow on control up, we see that `isWeak` is known to be false. Notably, we see `GetClassVar` being initialized by a call to `SymbolForClassRef`, which takes both `Name` and `isWeak` as parameter.

To decide whether this bug report is a true positive or not, we must, in part, show that `Name` can hold values that allows the flow of control to reach the range-based for loop, but never reach the assignment to `OID` inside it. Since the return value of `SymbolForClassRef` seems to influence whether the function returns early, the intention could have been that problematic values of `Name` should result in this early exit. It is also unclear why `isWeak` is known to be false, the intention of the programmer could have been to prevent the flow on control reaching the error point with its value as well.

On Figure 5.6., we see the relevant parts of the bug report after our improvement. A pair of notes around the call to `SymbolForClassRef` link to the function's definition. There, we learn where `isWeak`'s value was assumed, and we get a better picture of the return value that was later provided for the initialization of `ClassSymbol`. Also, the correlevance of `Name`, `isWeak` and the early return is proven, so an expert on this domain can likely judge the validity of the report.

## BEFORE:

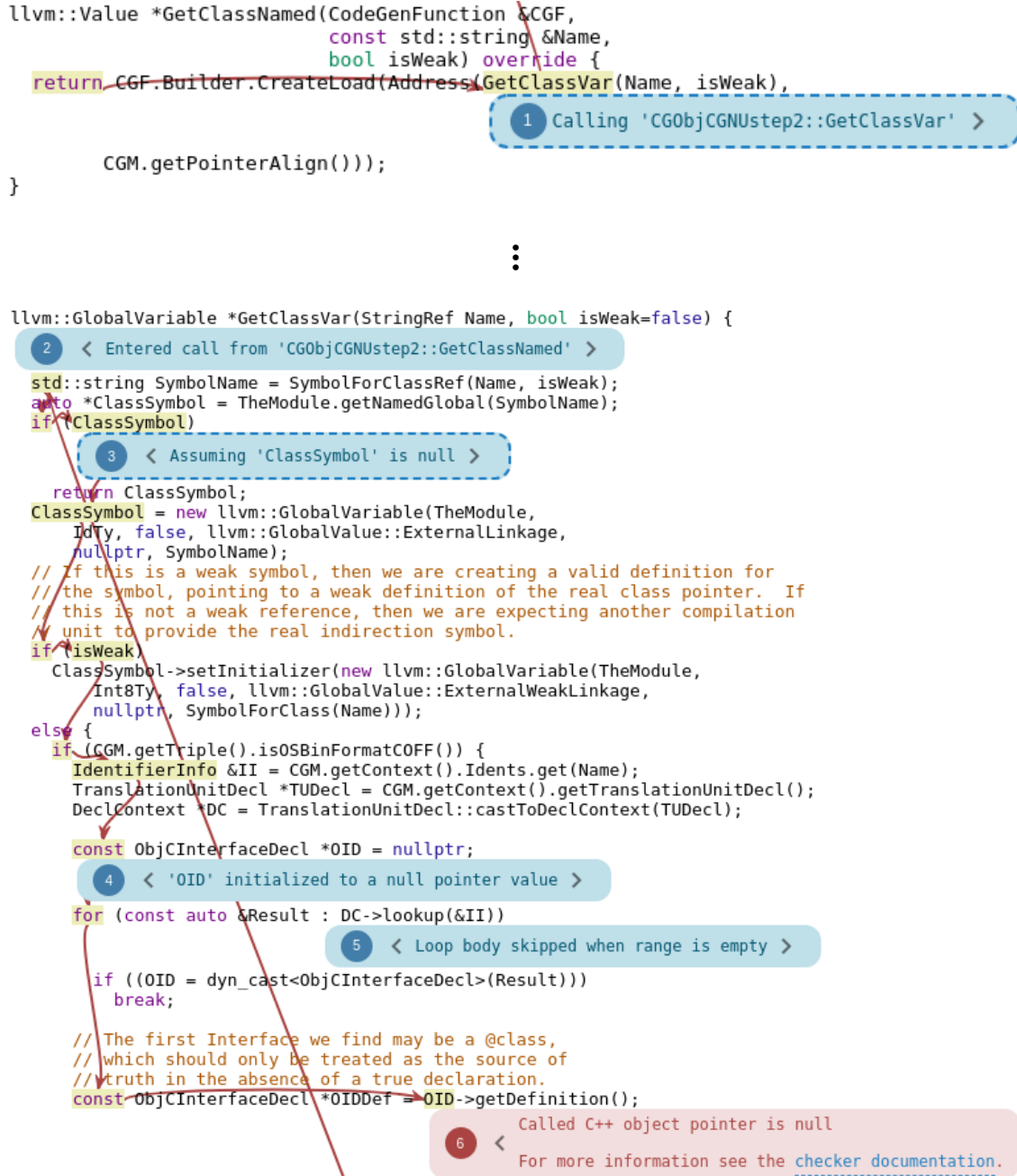


Figure 5.5: Bug report before our improvement from LLVM.

## AFTER:

```

llvm::GlobalVariable *GetClassVar(StringRef Name, bool isWeak=false) {
  2 < Entered call from 'CGObjCGNUstep2::GetClassNamed' >
  std::string SymbolName => SymbolForClassRef(Name, isWeak);
  3 < Calling 'CGObjCGNUstep2::SymbolForClassRef' >
  6 < Returning from 'CGObjCGNUstep2::SymbolForClassRef' >
  auto *ClassSymbol = TheModule.getNamedGlobal(SymbolName);
  if (!ClassSymbol)
    7 < Assuming 'ClassSymbol' is null >
    return ClassSymbol;
  ClassSymbol = new llvm::GlobalVariable(TheModule,
    IdTy, false, llvm::GlobalValue::ExternalLinkage,
    nullptr, SymbolName);
  // If this is a weak symbol, then we are creating a valid definition for
  // the symbol, pointing to a weak definition of the real class pointer. If
  // this is not a weak reference, then we are expecting another compilation
  // unit to provide the real indirection symbol.
  if (isWeak)
    ClassSymbol->setInitializer(new llvm::GlobalVariable(TheModule,
      Int8Ty, false, llvm::GlobalValue::ExternalWeakLinkage,
      nullptr, SymbolForClass(Name)));
  else {
    :
  }

  std::string SymbolForClassRef(StringRef Name, bool isWeak) {
    4 < Entered call from 'CGObjCGNUstep2::GetClassVar' >
    if (isWeak)
      5 < Assuming 'isWeak' is false, which participates in a condition later >
      return (ManglePublicSymbol("OBJC_WEAK_REF_CLASS_") + Name).str();
    else
      return (ManglePublicSymbol("OBJC_REF_CLASS_") + Name).str();
  }

```

Figure 5.6: New notes after our improvement in the report from LLVM.

# Chapter 6

## Related work

The use of static analysis is growing among industrial companies. In 2008, the authors of paper [50] reported that while the static analysis methods are frequent research areas in the academy, there are no many usage examples in the industry. Current trends show a more frequent usage of the static analysis tools [51, 52], but their advancement is still not generic.

Industry leader software companies show the most positive approach towards static analysis and its application in every day development. Google, Apple, Microsoft, Facebook, and others also participate in the development of such tools. Paper [53] reports about the lessons learned while developing static analysis tools at Google. The authors list the most frequent problems resulting in the developers not using static analysis tools or ignoring their warnings. These are the lack of tool integration into the developer’s workflow; the fact that *many warnings are not actionable*; the high number of false positives; situations where the bug is theoretically possible but in practice it does not manifest; the possibly high cost of the fix; and that *the users do not understand the warnings*.

The authors emphasize the importance of *actionable* messages: the warnings should include a suggestion to the (possible) fix, which in the best case could be applied mechanically. However, the authors state that many serious issues cannot be detected correctly or automatically fixed. In that case of the latter, the fix depends on the correct understanding of the report. They also claim that the developer’s happiness is a crucial factor for the successful introduction of static analysis on an organizational level. Non-understandable reports cause frustration among engineers and work against trust in static analysis tools.

The authors in [24] investigate why the use of static analysis tools is not as widespread as it would be possible. Unlike earlier studies, they focused on the developer's perception on using the tools, including the interaction with the user interface. The research was conducted via 40-60 minutes long semi-structured interviews with 20 developers with experience ranging from 3 to 25 years. Among these 20 developers, 14 people expressed negative impact on *the way in which the warnings are presented*. Apart from the possibility of overwhelming false positive warnings they mentioned that the reports are non-intuitive.

As the result of their research, they conclude that the developers are not able to understand what the tool is telling her, and it is a definite barrier to use static analysis tools. Nineteen of our 20 participants, felt that many static analysis tools do not present their results in a way that gives enough information for them to assess what the problem is, why it is a problem and what they should be doing differently.

A similar methodology was conducted in the research published in paper [54]. The authors surveyed 40+ developers and interviewed 11 industrial experts to understand the possibilities of better prioritization of static analysis tool reports. Among other interesting results, they found that ... *warnings hard to integrate in case they do not have teammates having enough expertise for fixing them. However, those warnings can be easily understood if the tools provide exhaustive descriptions.*

# Chapter 7

## Future work

We feel cautiously optimistic about our proposal regarding reaching definitions analysis, though we are yet to implement it and gather real world-results. This links back to the problems posed by Clang’s infrastructure: its CFG was constructed to make the construction of user-readable diagnostics easy, not so much for dataflow analysis. We made considerable progress in implementing a reaching definition analysis but paused to reflect on whether changes to the current repertoire intermediate representations are in order. Creating a new IR is a considerable undertaking, so we are researching the best course of action to take on this front.

The analyzer is aware that it is limited in terms of the information it can harness. For instance, calling functions with unavailable definitions often force a clear of its constraints on a subset of variables. Similar events are often large contributors to the appearance of false positive reports. After the analysis is concluded, the analyzer will inspect each bug find whether they are likely false positives, and regularly suppresses a portion of them. The more the analyzer understands what parts of the program are relevant to a bug, the more precisely it can suppress such reports; we are currently researching how to integrate our results and proposals into this library.

Our paper primarily focuses on how to add missing information to a bug report to make it *complete*, but too large reports can be just as detrimental to the user experience. Long reports often result from the fact that the analyzer describes a number of events in the bug path, even if only a limited number of messages would be sufficient around the error node. It would be interesting to explore how the combination of control dependency and reaching definitions analysis could help the analyzer discard part of the bug report.

Reaching definitions analysis could be a valuable component for new checkers to find even more intricate bugs by complementing symbolic execution with dataflow information.

# Chapter 8

## Conclusion

Static analysis is a promising approach to find bugs in computer software. Symbolic execution in particular is a powerful technique to find deeply rooted programming errors. It is an interprocedural path sensitive analysis, gaining a sophisticated understanding of how values would behave in a runtime environment without actually executing the program. However, it often struggles to turn these discoveries to easily comprehensible bug reports, demanding even more of the most expensive resource and least available in a software development project: human experts.

We demonstrated that the root cause of poor bug reports to otherwise valuable discoveries are caused by the fact symbolic execution can only reason about a single path of execution at a time. After the analysis is concluded, a tool such as the Clang Static Analyzer inspects the sequence of program states leading to the error, and constructs a set of diagnostic messages and notes to explain the flow control and change of values. However, these program states are oblivious to what could have happened on alternative paths of execution, as well as control dependence.

We propose two techniques to complement bug report generation. Control dependency analysis can tell that a condition point may have played a large part in the bug's occurrence. Reaching definitions analysis finds parts of the code could have changed the value of a bug causing variable had control flown there. We project the interaction of these techniques to replicate a program slicing-like behavior, significantly increasing an analyzer tools' understanding of the causes behind a bug. By adding control dependency analysis to the Clang Static Analyzer's bug report generation facilities, which has been a part of stable releases since version 10.0.0., we demonstrated how the discovery of such information allows a tool to construct

better bug reports.

## **Acknowledgment**

We would like to extend our thanks and gratitude towards Artem Dergachev, who was a mentor of the Google Summer of Code project under which most of the planning, implementation and troubleshooting took place.

Followup work on the project was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

# Bibliography

- [1] Barry Boehm and Victor R. Basili. “Software Defect Reduction Top 10 List”. In: *Computer* 34.1 (Jan. 2001), pp. 135–137. ISSN: 0018-9162. DOI: 10.1109/2.962984. URL: <http://dx.doi.org/10.1109/2.962984>.
- [2] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: <http://doi.acm.org/10.1145/1273442.1250746>.
- [3] Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: USENIX ATC’12 (2012), pp. 28–28. URL: <http://dl.acm.org/citation.cfm?id=2342821.2342849>.
- [4] Al Bessey et al. “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World”. In: *Commun. ACM* 53.2 (Feb. 2010), pp. 66–75. ISSN: 0001-0782. DOI: 10.1145/1646353.1646374. URL: <http://doi.acm.org/10.1145/1646353.1646374>.
- [5] Synopsys. *Coverity*. <https://scan.coverity.com/> (last accessed: 28-02-2019). 2019.
- [6] Roguewave. *Klocwork*. <https://www.roguewave.com/products-services/klocwork> (last accessed: 28-02-2019). 2019.
- [7] GrammaTech. *CodeSonar*. <https://www.grammatech.com/products/\codesonar> (last accessed: 28-02-2019). 2019.
- [8] Møller Anders and I. Schwartzbach Michael. *Static program analysis*. 2012. URL: <https://users-cs.au.dk/amoeller/spa/spa.pdf>.
- [9] Henry G. Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Trans. Amer. Math. Soc.* 74 (1953), pp. 358–366.

- [10] Kristóf Umann. *Enhancing bug reports in the Clang Static Analyzer*. accessed 03-Dec-2021. 2018. URL: <https://szelethus.github.io/gsoc2019/>.
- [11] Clang Tidy. *Clang-Tidy*. <https://clang.llvm.org/extra/clang-tidy/> (last accessed: 28-02-2019). 2019.
- [12] Thomas Reps, Susan Horwitz, and Mooly Sagiv. “Precise interprocedural dataflow analysis via graph reachability”. In: (1995), pp. 49–61.
- [13] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: <http://doi.acm.org/10.1145/360248.360252>.
- [14] Clang SA. *Clang Static Analyzer*. <https://clang-analyzer.llvm.org/>. 2019.
- [15] Ted Kremenek. “Finding software bugs with the clang static analyzer”. In: *Apple Inc* (2008).
- [16] Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *ACM Comput. Surv.* 51.3 (2018).
- [17] Zhongxing Xu, Ted Kremenek, and Jian Zhang. “A Memory Model for Static Analysis of C Programs”. In: *ISoLA’10* (2010), pp. 535–548. URL: <http://dl.acm.org/citation.cfm?id=1939281.1939332>.
- [18] Checker Developer Manual. *Clang Static Analyzer: Checker Developer Manual*. [https://clang-analyzer.llvm.org/checker\\_dev\\_manual.html](https://clang-analyzer.llvm.org/checker_dev_manual.html) (last accessed: 28-02-2019). 2019.
- [19] Artem Dergachev. *Clang Static Analyzer: A Checker Developer’s Guide*. <https://github.com/haoNoQ/clang-analyzer-guide> (last accessed: 28-02-2019). 2016.
- [20] Anna Zaks and Jordan Rose. *Building a Checker in 24 hours*. <https://www.youtube.com/watch?v=kdxlsP5QVPw>. 2012.
- [21] Chris Lattner. “LLVM and Clang: Next Generation Compiler Technology”. Lecture at BSD Conference 2008. 2008.
- [22] Kristóf Umann. “The Penultimate Challenge: Constructing bug reports in the Clang Static Analyzer”. 2019 LLVM Developers’ Meeting, San Jose, CA. 2019. URL: <https://llvm.org/devmtg/2019-10/talk-abstracts.html#tech17>.

- [23] J. Park, I. Lim, and S. Ryu. “Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild”. In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 2016, pp. 61–70.
- [24] B. Johnson et al. “Why don’t software developers use static analysis tools to find bugs?” In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.
- [25] Mark Weiser. “Program Slicing”. In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357. DOI: 10.1109/tse.1984.5010248. URL: <https://doi.org/10.1109%2Ftse.1984.5010248>.
- [26] Ericsson. *CodeChecker*. URL: <https://github.com/Ericsson/codechecker>.
- [27] Karl J. Ottenstein and Linda M. Ottenstein. “The program dependence graph in a software development environment”. In: *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments - SDE 1*. ACM Press, 1984. DOI: 10.1145/800020.808263. URL: <https://doi.org/10.1145%2F800020.808263>.
- [28] Susan Horwitz, Thomas Reps, and David Binkley. “Interprocedural slicing using dependence graphs”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.1 (1990), pp. 26–60.
- [29] D. Binkley and M. Harman. “A large-scale empirical study of forward and backward static slice size and context sensitivity”. In: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings*. 2003, pp. 44–53. DOI: 10.1109/ICSM.2003.1235405.
- [30] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Reading, MA: Addison-Wesley, 1986.
- [31] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Transactions on Programming Languages and Systems* 13.4 (1991), pp. 451–490. DOI: 10.1145/115372.115320. URL: <https://doi.org/10.1145%2F115372.115320>.

- [32] Vugranam C. Sreedhar and Guang R. Gao. “A linear time algorithm for placing  $\phi$ -nodes”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press, 1995. DOI: 10.1145/199448.199464. URL: <https://doi.org/10.1145%2F199448.199464>.
- [33] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.
- [34] DeLesley Hutchins, Aaron Ballman, and Dean Sutherland. “C/C++ Thread Safety Analysis”. In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014. DOI: 10.1109/scam.2014.34. URL: <https://doi.org/10.1109%2Fscam.2014.34>.
- [35] Herb Sutter. *Lifetime safety: Preventing common dangling*. Tech. rep. Technical Report. Microsoft Corporation, 2018.
- [36] Gábor Horváth and Matthias Gehre. “Implementing the C++ Core Guidelines’ Lifetime Safety Profile in Clang”. European LLVM Developers Meeting, Brussels. 2019. URL: [https://llvm.org/devmtg/2019-04/talks.html#Talk\\_18](https://llvm.org/devmtg/2019-04/talks.html#Talk_18).
- [37] Gábor Horváth and Norbert Pataki. “Categorization of C++ Classes for Static Lifetime Analysis”. In: *Proceedings of the 9th Balkan Conference on Informatics*. ACM, 2019. DOI: 10.1145/3351556.3351559. URL: <https://doi.org/10.1145%2F3351556.3351559>.
- [38] Réka Kovács, Gábor Horváth, and Zoltán Porkoláb. “Detecting C++ lifetime errors with symbolic execution”. In: *Proceedings of the 9th Balkan Conference on Informatics*. 2019, pp. 1–6.
- [39] Kristóf Umann. *A survey of dataflow analyses in Clang*. accessed 2021. jan. 30. Oct. 2020. URL: <https://lists.llvm.org/pipermail/cfe-dev/2020-October/066937.html>.
- [40] The Bitcoin Core. *Bitcoin Core*. URL: <https://bitcoincore.org/>.
- [41] Daniel Marjamäki. *CppCheck: a tool for static C/C++ code analysis*. 2013. URL: <http://cppcheck.sourceforge.net/>.
- [42] Marco Bambini. *Gravity*. URL: <https://github.com/marcobambini/gravity>.

- [43] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: (2004), p. 75.
- [44] OpenSSL Software Foundation. *OpenSSL*. URL: <https://openssl.org/>.
- [45] Anders Bakken. *Rtags*. URL: <http://www.rtags.net>.
- [46] Amazon Web Services. *S2N homepage*. URL: <https://github.com/awslabs/s2n/>.
- [47] Apache Software Foundation. *Apache Xerces*. URL: <https://xerces.apache.org/>.
- [48] Eric Goubault, Sylvie Putot, and Franck Védryne. “Modular Static Analysis with Zonotopes”. In: *Static Analysis*. Springer Berlin Heidelberg, 2012, pp. 24–40. DOI: 10.1007/978-3-642-33125-1\_5. URL: [https://doi.org/10.1007/978-3-642-33125-1\\_5](https://doi.org/10.1007/978-3-642-33125-1_5).
- [49] *Apply the Clang Static Analyzer to LLVM-based projects*. <https://summerofcode.withgoogle.com/archive/2019/projects/4876350878384128/>. 2019. (Visited on 01/30/2021).
- [50] Nathaniel Ayewah et al. “Using Static Analysis to Find Bugs”. In: *IEEE Software* 25.5 (2008), pp. 22–29. DOI: 10.1109/ms.2008.130. URL: <https://doi.org/10.1109/2Fms.2008.130>.
- [51] Moritz Beller et al. “Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016. DOI: 10.1109/saner.2016.105. URL: <https://doi.org/10.1109/2Fsaner.2016.105>.
- [52] Fiorella Zampetti et al. “How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines”. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017. DOI: 10.1109/msr.2017.2. URL: <https://doi.org/10.1109/2Fmsr.2017.2>.
- [53] Caitlin Sadowski et al. “Lessons from building static analysis tools at Google”. In: *Communications of the ACM* 61.4 (2018), pp. 58–66. DOI: 10.1145/3188720. URL: <https://doi.org/10.1145/2F3188720>.

- [54] C. Vassallo et al. “Context is king: The developer perspective on the usage of static analysis tools”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 38–49. DOI: 10.1109/SANER.2018.8330195.

# List of Figures

|     |                                                                                                                                                                                                                                                                       |    |
|-----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.1 | A code snippet and function <code>f</code> 's control flow graph. . . . .                                                                                                                                                                                             | 12 |
| 3.2 | A code snippet and analysis results demonstrating how the analyzer struggles to realize that <code>x</code> 's value and dereference depends on <code>flag</code> , and as a result, will not construct diagnostic messages to explain relevant events to it. . . . . | 13 |
| 3.3 | A simplified <code>ExplodedGraph</code> after analyzing Figure 3.1. . . . .                                                                                                                                                                                           | 13 |
| 3.4 | Code snippets that highlight deficiencies in the analyzer's understanding of bugs when generating reports. . . . .                                                                                                                                                    | 14 |
| 5.1 | Bug report before our improvement from CppCheck. . . . .                                                                                                                                                                                                              | 28 |
| 5.2 | New notes after our improvement in the report from CppCheck. . . .                                                                                                                                                                                                    | 29 |
| 5.3 | Bug report before our improvement from Xerces. . . . .                                                                                                                                                                                                                | 31 |
| 5.4 | New notes after our improvement in the report from Xerces. . . . .                                                                                                                                                                                                    | 32 |
| 5.5 | Bug report before our improvement from LLVM. . . . .                                                                                                                                                                                                                  | 34 |
| 5.6 | New notes after our improvement in the report from LLVM. . . . .                                                                                                                                                                                                      | 35 |