Uncovering Hidden Dependencies: Constructing Intelligible Path Witnesses using Dataflow Analyses[*]

Kristóf Umann[ab], Gábor Horváth[ac], and Zoltán Porkoláb[ad]

Abstract

The lack of sound, concise and comprehensive error reports emitted by a static analysis tool can cause increased fixing cost, bottleneck at the availability of experts and even may undermine the trust in static analysis as a method. This paper presents novel techniques to improve the quality of bug reports for static analysis tools that employ symbolic execution. 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 an improved bug report generator algorithm for the Clang Static Analyzer. After being tested by the open source community our solution became enabled by default in the tool.

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

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 [12]; 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. Another approach,

[*]This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

[a]Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

[b]E-mail: szelethus@caesar.elte.hu, ORCID: 0000-0002-6679-5614

[c]E-mail: xazax@caesar.elte.hu, ORCID: 0000-0002-0834-0996

[d]E-mail: gsd@caesar.elte.hu, ORCID: 0000-0001-6819-0224

DOI: 10.14232/actacyb.299805

the *dynamic analysis method* using tools like *Valgrind* [38], or *Google Address sanitizer* [47] 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* techniques do not require the concrete execution of the program, and are often based only on the program's source code, and do not require any input data. It is a popular method for finding bugs and code smells [10, 50, 42, 17]. 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 analysis methods apply heuristics, which means that often they may *underestimate* or *overestimate* the program behavior [3]; in other words, static analysis *trades precision for coverage*. In practice, this means static analysis tools sometimes do not report existing issues which is called a *false negative*, and sometimes they report correct code erroneously as a problem, which is called as *false positive*. There is a continuous struggle to improve tools and methods, but there is a theoretical limitation: paraphrasing Rice's theorem [44] 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 serious bottleneck in the otherwise automated 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 the step where humans involved [28]. 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, the construction of intelligible bug reports also gets much more difficult. In this paper, we present the report generation 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 usually struggles to relate back to the source code and many time is unable to consider the proper context broader than the actual path of execution leading to the bug.

We discuss possible 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 implemented as an extension to 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. Our report generation improvement was tested by the open source community and accepted to merge into the tool since version 10.0.0. As this improvement has been enabled in the releases since[1], we feel there is a real world benefits to our results. We documented our research, implementation, and some of the evalution processes leading up to this paper in [54].

This paper is structured as follows. In Section 2 we overview the technical background related to symbolic execution and its implementation in the Clang Static Analyzer tool. In Section 3, we discuss our expectations for an intelligible bug report, and present techniques to generate them and their shortcomings. Section 4 details our proposals and implementations. We evaluate our solution implemented for the Clang Static Analyzer in Section 5. Related work is surveyed in Section 7. Future areas of research and implementation are discussed in Section 8. Finally, we conclude our paper in Section 9.

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*) [16] and control flow analysis trade understanding of runtime behavior for faster analysis speed. Dataflow analyses [43] are able to argue about the flow of information within the bounds of a given function, and most variants strike a middle ground in terms of space and time complexity and the effectiveness of the analysis. Symbolic execution [30] takes a rather radical approach, by essentially interpreting the source code, and analyze 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 provides more information about runtime behavior.

This section discusses symbolic execution and its implementation in the Clang Static Analyzer [15].

[1]As of the writing of this paper, the latest Clang release is 12.0.0.

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 set to exercise 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 [33]. 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, which is a mapping of variables to symbolic values, and a constraint solver, which contains constraints on symbolic values [7, 59]. The store is updated when a memory location is written, and the constraint solver is updated after each evaluation of a conditioned branch.

The Clang Static Analyzer [15, 14, 20, 60] 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 1a). Clang [34] 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, we refer to the Clang Static Analyzer under the term *analyzer*.

Symbolic execution 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 1a, `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 Bi_{foo} as the ith block of `foo`'s CFG, the path of execution mentioned above is as follows:

$$B5_f\ B4_f\ B3_g\ B2_g\ B1_g\ B4_f\ B3_f\ B2_f\ B3_g\ B2_g\ B1_g\ B2_f\ B0_f$$

2.2 The ExplodedGraph

During analysis, the analyzer builds a 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 2. The ExplodedGraph is a different data structure to the CFG because it contains far more information (assumptions on values, memory regions) [52]. It is not 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. With that said, it is possible to map each ExplodedNode (a node of the ExplodedGraph) to a specific CFGBlock (or simply block, a node of the CFG), or CFGEdge (a directed edge of the CFG).

Mind that the analyzer does not view the path of execution from a "human" perspective. It processes these nodes from the ExplodedGraph unaware of contextual information in the source code, or even nodes on other paths of execution. For instance, the path in red in Figure 2 does not include any information about user's intent to assign `x` a non-null object, and will not be considered.

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 [41, 28]. However, making the generated reports intelligible and easily digestible is rarely discussed. However, many researches [28, 29, 36, 45] point to the fact, that understanding the error report and converting it to an executable action by the developer is crucial for the acceptance and the effective use of static analysis tools.

In this section, we define a non-comprehensive set of guidelines on an ideal bug report, and overview how the analyzers approach this issue, and struggle relating to the limitations of the ExplodedGraph.

Bug report generation is done after the entire analysis is concluded by the inspecting 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 the program points where a bug was discovered (the red path seen in Figure 2).
- 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 (see Figure 1b-1c).

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.*

Definition. *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.*

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, looking for noteworthy events to the slicing criterion (error location, {bug causing variables}). 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 of 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` is responsible for constructing a message for each evaluated condition. When a condition is seen by this visitor in an ExplodedNode, for instance, `if (coinflip())`, the message "Assuming the condition is true" may be constructed.

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' "). Visitors can themselves create new visitors, if warranted. Suppose that that last write is in the form of an assignment (`x = y;`), `FindLastStoreBRVisitor` would register a new instance of itself to explain y.

3.2.2 Interestingness propagation

By 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. Just as visitors may themselves create new visitors during bug report construction, they may also mark new entities interesting, or propage interestingness from one entitity to another. In a later stage, after all diagnostics have been constructed, messages in function calls not describing any interesting entity are pruned.

At last, these two techniques are combined in what we call *expression 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 not explored by the analyzer on that bug path can make it challenging to understand the intent of the programmer.

Figure 1a 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 1b shows a report from the analyzer displayed by CodeChecker [21] 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 1c.

For each example in Figure 3, 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[2], which we discuss in further detail as follows:

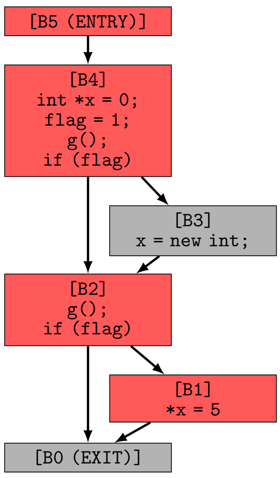
3.3.1 Figure 3a: Control dependency is not recognized

Analysis starts at line 14, noting variable `x` to be a null pointer, and the global variable `flag` to be 1. Then, the function call to `g()` will appropriately set `flag`'s value to unknown. 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. On the former, a dereference of `x` is found on line 21, which is known to be null. The analyzer will cut a bug path out of the ExplodedGraph which terminates in this erronous program state, and configure its bug report generation facilities to start tracking `x`.

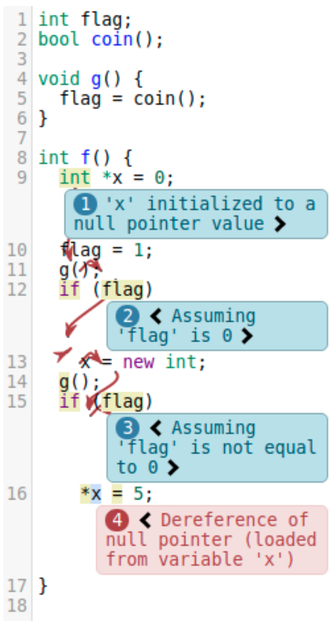
During bug report generation, the analyzer can find the relevant statement to `x` regarding data dependencies, which is its initialization on line 14. It will, however, fail to recognize a relevant statement to the bug – namely, had `flag` not been 0 on line 20, the bug would not have occurred. We will define it more precisely in later

[2]Figure 1a combines the classes of problems displayed in Figure 3a and Figure 3b

```
int flag;
bool coin();

void g() {
  flag = coin();
}

int f() {
  int *x = 0;
  flag = 1;
  g();
  if (flag)
    x = new int;
  g();
  if (flag)
    *x = 5;
}
```

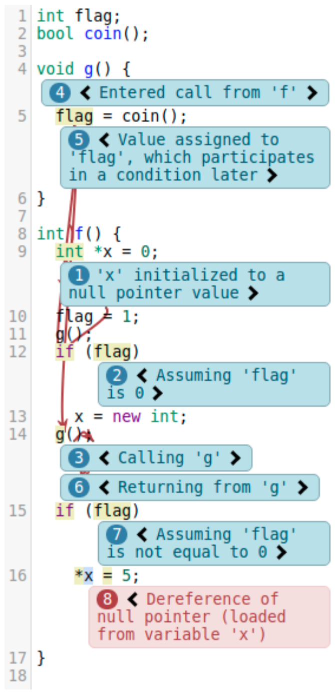


(a) A code snippet and function f's control flow graph.



```
int flag;
bool coin();

void g() {
  flag = coin();
}

int f() {
  int *x = 0;
    1 'x' initialized to a null pointer value
  flag = 1;
  g();
  if (flag)
    2 Assuming 'flag' is 0
    x = new int;
  g();
  if (flag)
    3 Assuming 'flag' is not equal to 0
    *x = 5;
    4 Dereference of null pointer (loaded from variable 'x')
}
```

(b) Before



```
int flag;
bool coin();

void g() {
  4 Entered call from 'f'
  flag = coin();
  5 Value assigned to 'flag', which participates in a condition later
}

int f() {
  int *x = 0;
  1 'x' initialized to a null pointer value
  flag = 1;
  g();
  if (flag)
    2 Assuming 'flag' is 0
    x = new int;
  g();
  3 Calling 'g'
  6 Returning from 'g'
  if (flag)
    7 Assuming 'flag' is not equal to 0
    *x = 5;
    8 Dereference of null pointer (loaded from variable 'x')
}
```

(c) After

Figure 1: A code snippet and analysis results demonstrating how the analyzer struggles to realize that x's value and derefence depends on flag, and as a result, will not construct diagnostic messages to explain relevant events to it.

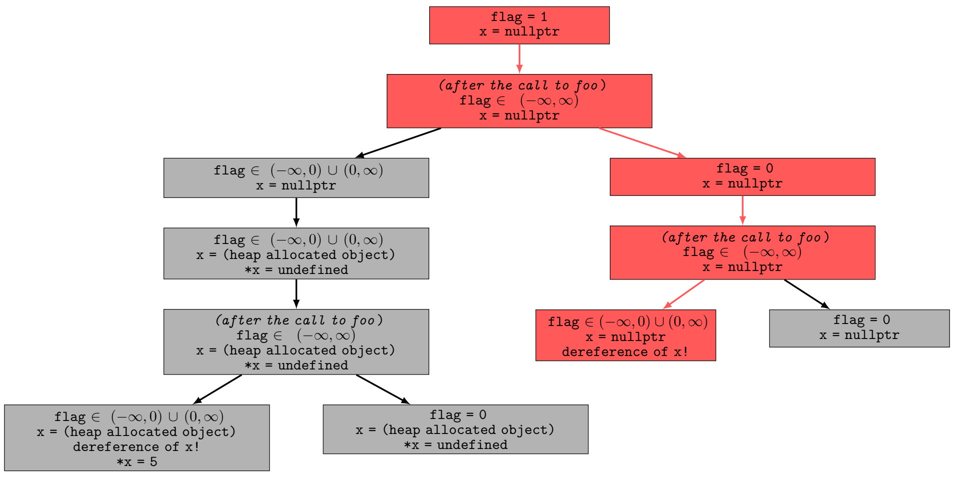

flag = 1
x = nullptr
(after the call to foo)
flag ∈ $(-\infty, \infty)$
x = nullptr
flag ∈ $(-\infty, 0) \cup (0, \infty)$
x = nullptr
flag = 0
x = nullptr
flag ∈ $(-\infty, 0) \cup (0, \infty)$
x = (heap allocated object)
*x = undefined
(after the call to foo)
flag ∈ $(-\infty, \infty)$
x = nullptr
(after the call to foo)
flag ∈ $(-\infty, \infty)$
x = (heap allocated object)
*x = undefined
flag ∈ $(-\infty, 0) \cup (0, \infty)$
x = nullptr
dereference of x!
flag = 0
x = nullptr
flag ∈ $(-\infty, 0) \cup (0, \infty)$
x = (heap allocated object)
dereference of x!
*x = 5
flag = 0
x = (heap allocated object)
*x = undefined

Figure 2: A simplified `ExplodedGraph` after analyzing Figure 1a.

section, but this property makes line 20 a *control dependency* of line 21. Control dependency is defined on the CFG, not the bug path, hence the analyzer being oblivious to it at this phase. 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 obvious why `flag`'s value is assumed to be 0 on line 20.

3.3.2 Figure 3b: Reaching definition is not in the bug path

The analysis, and the eventual costruction of the bug path is done similarly to Figure 3a. The analyzer can again find `x`'s initialization as important, 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 and `x`'s initialization are so-called *reaching definitions* to `x` on line 21 – loosely, there exists a path in the CFG from them to line 21 without any interleaving assignments to `x`. `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. Reaching definitions is also a property of the CFG, so line 18 is not recognized as important, leading the analyzer to believe that its control dependency, line 17, which *is* on the bug path is not worth explaining in further detail either.

3.3.3 Figure 3c: Reaching definition is in a different, but inlined stack frame

This example presents another layer of difficulty to Figure 3b – 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. In the previous cases, control dependency and reaching

```
int flag;
bool coin();

void g() {
  flag = coin();
}






int f() {
  int *x = 0;
  flag = 1;
  g();



  if (!flag)
    *x = 5;
}
```

(a)

```
int flag;
bool coin();

void g() {
  flag = coin();
}






int f() {
  int *x = 0;
  flag = 1;
  g();
  if (flag)
    x = new int;


  *x = 5;
}
```

(b)

```
int flag;
bool coin();

void g() {
  flag = coin();
}

void h(int **x) {
  if (flag)
    *x = new int;
}

int f() {
  int *x = 0;
  flag = 1;
  g();

  h(&x);
  g();
  if (flag)
    *x = 5;
}
```

(c)

```
int flag;
bool coin();

void g() {
  flag = coin();
}

void h(int **x) {

  *x = new int;
}

int f() {
  int *x = 0;
  flag = 1;
  g();
  if (flag)
    h(&x);
  g();
  if (flag)
    *x = 5;
}
```

(d)

Figure 3: Code snippets that highlight deficiencies in the analyzer's understanding of bugs when generating reports.

definitions could have been recognized within `f` itself; here, an interprocedural technique is required, whereas a CFG is constructed for only a single function at a time. While we can say that the assignment to `x` in `h` is a reaching definition to `x` on line 21, this information needs to be carried from one CFG to another. Although symbolic execution is interprocedural, control dependency analysis and reaching definitions analysis (and many similar lightweight techniques) are not.

3.3.4 Figure 3d: 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 (the analyzer has not entered this function on the path where the bug is discovered). Inlining functions can demand non-trivial modeling, such as lifetime extension, moves, and the evaluation of arguments. This makes bridging the gap in between CFGs all the more difficult. Generally speaking, the "further" the analyzer has to stray from the bug path, the more challenging bug report construction becomes.

4 Proposed solution

Program slicing is a field of study 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 [57] was intraprocedural, and aimed at monolithic, single-procedure programs [40]. Interprocedural variants are explored in numerous studies [26, 11, 56], 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 the current 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, adjustments would be required to make program slicing a valuable part of Clang's bug report generation facilities. A great candidate to bridge this gap might be bug reporter visitors, as they can reason about data dependencies with rather great precision across function calls. However, they would be partially redundant with the data dependency analysis built into program slicing. 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 are control dependent on the expression in the switch statement.

We defined relevance in part such that slice $a := (stmt_a, vars_a)$ is relevant to slice $b := (stmt_b, vars_b)$, if $stmt_a$ is a control dependency of $stmt_b$. The following exercise demonstrates why this is reasonable: Suppose that we constructed b after discovering a null pointer dereference error (for instance, $(line16, \{x\})$ for Figure 1a), and have already marked $stmt_b$ as interesting. We argue that regarding its control dependency, $stmt_a$, as relevant enables the analyzer to better understand the context of this bug; 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 *postdominance*. We say that block A *dominates* block B ($A \operatorname{dom} B$), if every path from the entry block to B must go through A. We say A *strictly dominates* B ($A \operatorname{sdom} B$), if A dominates B and $A \neq B$. We say B *postdominates* A ($B \operatorname{pdom} A$), if every path from A to the exit block must go through B, and similarly, B *strictly postdominates* A ($B \operatorname{spdom} A$) if B postdominates A and $B \neq A$ [1]. An example can be seen regarding dominance in Figure 4.

We say that block B is *control dependent* on block A ($B \operatorname{cd} A$) if there exists an edge from A to C such that B postdominates C, and if B is not 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 1a, $B1$ 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 [19]. The post dominance frontier set of block A ($PDF(A)$) is the set of B nodes from

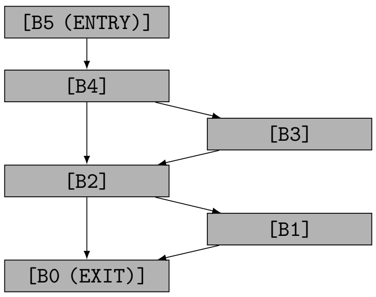


Figure 4: A simple control flow graph. The entry block (strictly) dominates every (other) block, and the exit block (strictly) postdominates every (other) block. B4 dominates B3, but B3 does not dominate B2, since the path B5 $\rightarrow$ B4 $\rightarrow$ B2 excludes B3. B2 postdominates B4, but B3 does not postdominate B4.

the inverse CFG[3] such that A dominates a predecessor of B but does not strictly dominate B:

$$\begin{aligned} PDF(A) := \{B | (\exists P \in \mathit{Pred}(B)) \wedge \\ \wedge\, (A \,\mathrm{pdom}\, P \wedge \neg A \,\mathrm{spdom}\, B)\} \end{aligned}$$

Calculating PDF sets quickly yields control dependence:

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

We implemented dominance frontier sets with the algorithm described in [48].

For a CFG with E edges and N CFGBlocks, calculating PDF sets has a worst-case complexity of $\mathcal{O}(E + N^2)$, but is often linear in practice [19].

4.1.2 Integration of control dependence

As bug path visitors continously 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 location where the tracking started. If so, it will instruct the analyzer to track the condition of that statement. Essentially, each visitor instance holds a (statement where tracking starts, {tracked variable}) slicing criterion.

Figure 3a 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

[3]An inverse of a CFG is constructed by reversing all of its edges in the graph.

visitors, including our own. As it ascends the bug path and finds the ExplodedNode describing the evaluation of `!flag` on line 20, our visitor checks whether line 20 is a control dependency of where it started tracking from (line 21). As a result, it will instruct the analyzer to track `flag`. `FindLastStoreBRVisitor` would find `flag`'s last store on line 5, and a diagnostic message will be constructed to describe it. Since `flag` is tracked, it is also an interesting variable and the analyzer will prevent the pruning of messages inside `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

Data dependency analysis on the bug path (done by `FindLastStoreBRVisitor`) 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. Data dependence analysis on the CFG, which is done with a *dataflow algorithm*, is not in such a privileged position. However, as we have demonstrated before, analyses on the CFG could yield information on code outside of what the bug path that can be valuable.

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.

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. Though might be defined somewhat differently from algorithm to algorithm, they are usually 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 [18] 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. [4] In Figure 1a, `x` is live in blocks B4, B3 and B2, but 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 1a `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 through aliasing. 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 AST, to which the CFG links back to, was designed for diagnostics construction, not for such an analysis [53]. While lacking an intermediate representation higher than LLVM IR but lower then Clang AST makes it rather difficult to implement in Clang for the purpose of finding programming errors, there are a few, such as Clang's thread safety analysis [27] and lifetime analysis [49, 24, 25, 32].

4.2.2 Reaching definitions analysis

We call the write of variable x a *definition* of x. Any statement that *may* write x (e.g. through aliasing) is also regarded as a definition of x. When describing analyses concerning definitions, we define $GEN[B]$ sets such that they contain

[4] 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. In such a case, valuable liveness information is lost *inside* the block. This problem can be solved by splitting up basic block to only contain a single statement.

the set of definitions present in basic block B („B writes x"), and $KILL[B]$ sets such that they contain every other definitions in the CFG that generate the same variables as B („B overwrites the value x may have gotten in other blocks").

The ingoing reaching definitions [18] 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:

$$REACH_{in}[B] = \bigcup_{P \in pred[B]} REACH_{out}[P]$$

$$REACH_{out}[B] = GEN[B] \cup (REACH_{in}[B] \setminus KILL[B])$$

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 1a, if we denote definitions by a (variable, line number) pair, B1's reaching definitions set would be the following:

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

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 occuring. 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 evaluation of 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 3b. Reaching definitions analysis overapproximates the set of statements that are considered definitions, meaning that the analyzer should keep track of whether a definition was found as a result of overapproximation. With that said, 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 3c 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 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 present on the bug path. Nonetheless, we would need to enhance our reaching definitions algorithm with some pointer aliasing capabilities.

For the last class of problems demonstrated in Figure 3d, 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 will be a result of overapproximation, so the analyzer might not trust is enough to explain control flow around it. To reason about `h`, we are forced the 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.

The concept behind the interaction of reaching definitions with control dependency analysis displays the many of the characteristics of static backward program slicing.

5 Results

We evaluated our work from two perspectives. First, we gathered data on open source projects by running the Clang Static Analyzer on their source code before and after our improvements. We inspected almost all reports individually and tried to subjectively argue for or against whether the reports' readability improved. We also tried to find certain objective metrics to measure the impact of our work.

Second, we sent out surveys to participants with varying degree of expertice in C/C++ and static analysis to learn whether other developers would find our improvements beneficial.

5.1 Measurements on open source projects

We tested our solution on, as seen in Table 1, the following open-source C and C++ projects: Bitcoin [51], CppCheck [37], Gravity [8], gRPC [23], LLVM and Clang [35], OpenSSL [39], Protobuf [22], Rtags [6], S2N [2], TinyVM [31], Xerces [4] and XGBoost [58]. Combined, these projects cover a wide variety of coding techniques, codebase sizes, and different versions of the languages' standards.

In Table 1 we show how many reports did our contribution affect. Out of the 12 open source projects, reports remained unchanged in 7. Out of 1096 bug reports, 2.4% received additional notes. We intentionally fine tuned our solution to limit its impact, and have observed that preserved information from previously disregarded function calls always meaningfully added to the intelligility of the analyzed path of execution.

Table 1: Evaluation of control dependency tracking in terms of how many reports received additional notes. The last row shows findings every other project other than the first five remained unchanged.

	Total reports	Changed	% of changed
CppCheck	44	7	15.9 %
Gravity	16	1	6.3 %
gRPC	229	15	6.6 %
LLVM + Clang	249	2	0.8 %
Xerces	106	1	0.9 %
Others combined	451	0	0 %

In the case for Xerces, Gravity, some of the CppCheck reports, we were especially pleased on how the extra information on conditions provided further high-level information. We found that conditions closer to the bug point are more likely to be directly data dependent on the bug causing variable. Upon learning more about the condition, we also learned of more high-level properties on the bug causing variable.

In the case for LLVM, gRPC, and the other half of the CppCheck reports, when the pivotal point (like assigning null to a pointer) was very close to the bug point, the extra information did not add much to the already decent report. Even in these cases however, the rest of the report (altough not important to understand why the bug occured) were easier to understand.

In the context of how memory and runtime intensive static analysis is, the costs of bug report construction are usually assumed to be negligable. We expect our contribution in particular to very little impact even in the context of bug report construction, as all of the control flow analyses are calculated at a prior step in the compilation process, and we simply reuse it.

5.2 Survey

We sent out surveys 11 people to measure whether a developer not taking part in our research would find our improvements beneficial. Out of them, 9 participated All of them were male, their avarage age was 30 at the time of the survey, ranging from 24 to 56. Participation was free and voluntary.

While all of our participants were software developers, 3 of them mainly wrote code in Python, 2 of them were teaching C++ at our university but wrote little C/C++ code outside the classroom. The remaining 4 were full-time C++ developers.

All of the participants were familiar with static analysis, with 4 of them being active contributors to Clang itself (but to our research). The remaining 5 worked on visualizer tools for static analyzers, but not the analyzers themselves.

We selected 11 bug reports from those that we collected on analyzing open source projects. After our contribution, all of these reports contained additional information than prior to it. We will call these versions of the same bug report the "after" and the "before" versions.

All surveys contained all of the bug reports, but each report was only presented in either "before" or "after" state. Each survey way unique in terms of which reports were shown in which state, but all survey contained roughly the same number of "before" and "after" reports.

In total, we received 99 bug report evaluations. On the following question: "Sometimes, I was unsure how the analyzer analyzed this path of execution, and wished for more explanation.", answers could be given on a range from 1 to 5, with 1 strongly disagreeing and 5 strongly agreeing. As seen in Figure 5, on avarage, before our contribution users answered with 2.901/5, but this desire was somewhat lower after our contribution, a 2.804/5, while users rated "Some notes were annoying and made it more difficult to understand the bug." with the same score before and after our improvement.

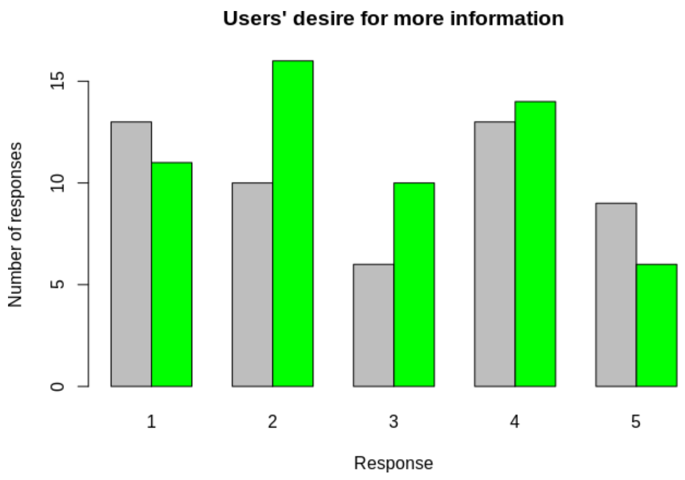


Figure 5: Responses to the question "Sometimes, I was unsure how the analyzer analyzed this path of execution, and wished for more explanation.". 1 strongly disagrees, 5 strongly agrees. The columns in grey display the score on bug reports prior to, and the columns in green display the score after our improvement.

5.3 Threats to validity

As for the evaluations on open source projects, our selection lacks meaningful amount of modern C++ code, specifically, C++14 or newer.

As for our survey, while in terms of expertise in C/C++, our participants varied in range, they were are rather knowledgable about the Clang Static Analyzer, with 7 of the 9 having made at least one contribution to it. Our survey could have benefitted from a greater range on familiarity with static analyzers. All of the participants that responded to our survey were male, a fact that could also use some diversifying.

Most importantly, our sample size of 9 participants and 11 bug reports is small. It is our view that a participant should have at least an intermediate C/C++ knowledge, and at least some familiarity with the concept of static analysis in real world applications, and it proved difficult to find people who met these criteria.

6 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.

6.1 Example 1

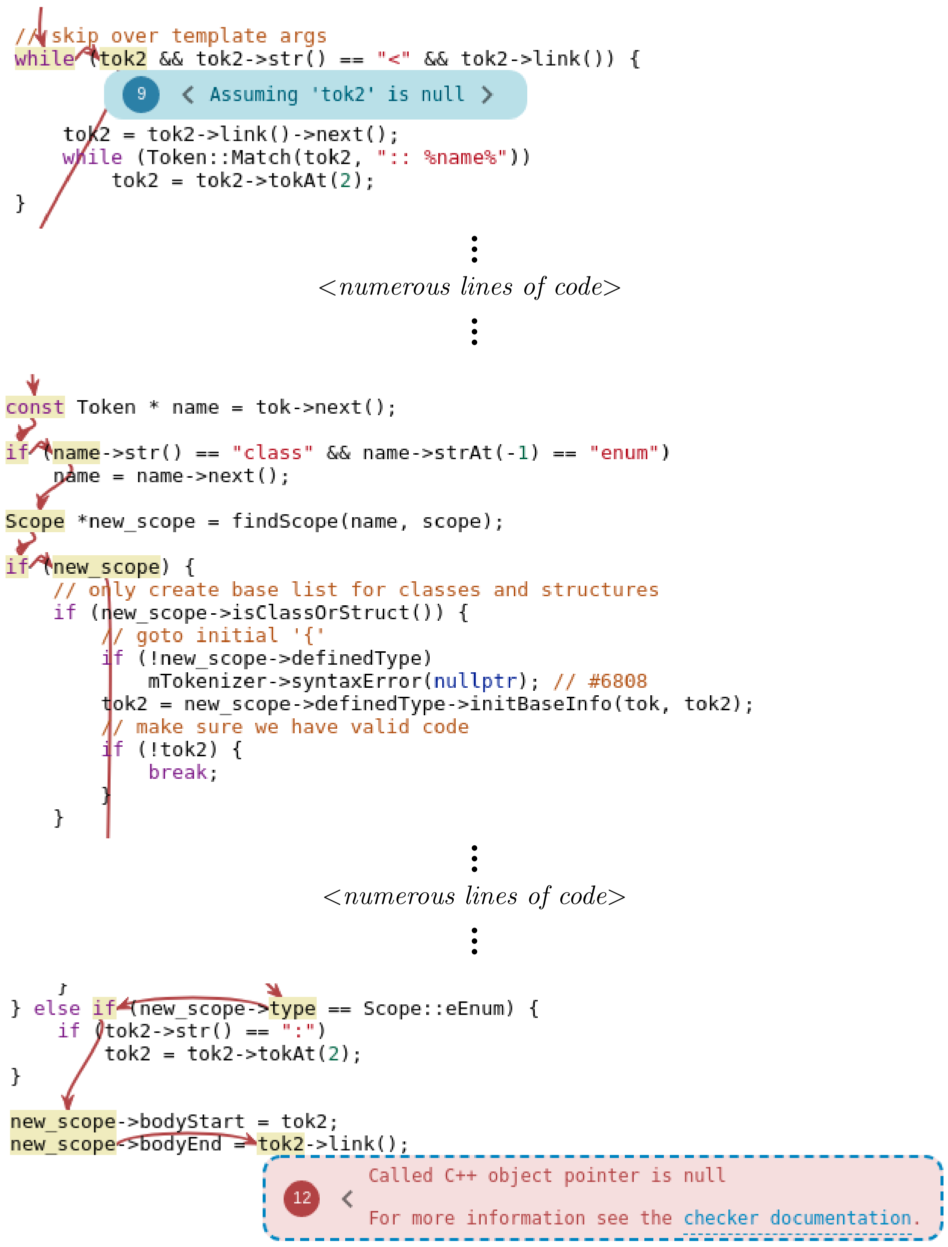
This example is from the project CppCheck, in the file `lib/symboldatabase.cpp`. In Figure 6, the bug report is shown which was generated without our improvement. At one point, variable `tok2` is assumed to be null, which eventually leads to a null dereference bug. To decide whether this report is a 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 derefence 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?

In Figure 7, 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;
        }
    }
```

⋮

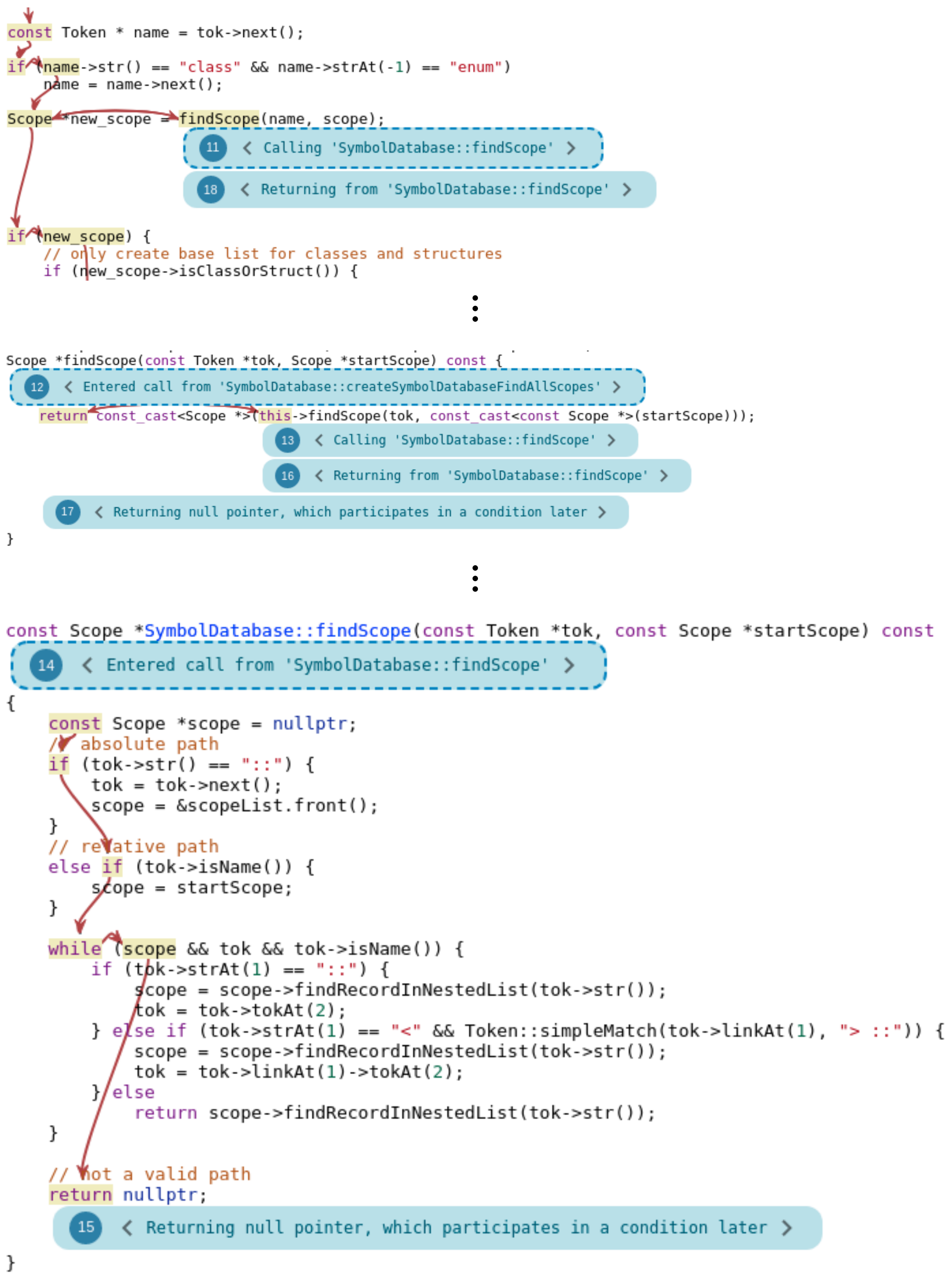
<numerous lines of code>

⋮

```
} 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 6: 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);
                    11  < Calling 'SymbolDatabase::findScope' >
                    18  < Returning from 'SymbolDatabase::findScope' >

if (new_scope) {
    // only create base list for classes and structures
    if (new_scope->isClassOrStruct()) {

                                 ⋮

Scope *findScope(const Token *tok, Scope *startScope) const {
 12  < Entered call from 'SymbolDatabase::createSymbolDatabaseFindAllScopes' >
    return const_cast<Scope *>(this->findScope(tok, const_cast<const Scope *>(startScope)));
                           13  < Calling 'SymbolDatabase::findScope' >
                           16  < Returning from 'SymbolDatabase::findScope' >
     17  < Returning null pointer, which participates in a condition later >
}

                                 ⋮

const Scope *SymbolDatabase::findScope(const Token *tok, const Scope *startScope) const
 14  < Entered call from 'SymbolDatabase::findScope' >
{
    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;
     15  < Returning null pointer, which participates in a condition later >
}
```

Figure 7: New notes after our improvement in the report from CppCheck

6.2 Example 2

This example is from the project Xerces in the file `TraverseSchema.cpp`. Reading the report in Figure 8 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?

In Figure 9, 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.

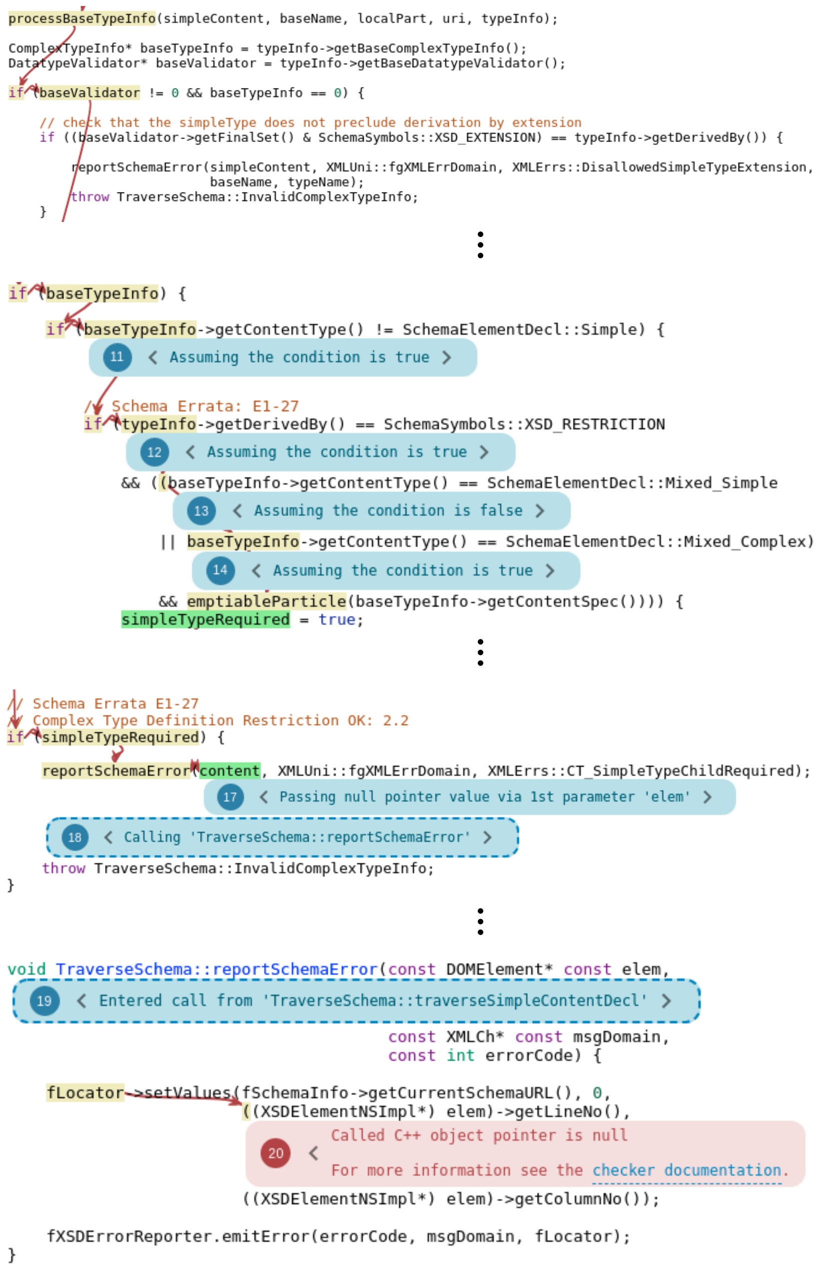
Upon reviewing the definition of `baseTypeInfo` and `baseValidator`, we can see that the getter functions they are initialized with pair with these setter functions, explaining how the analyzer knew their precise value.

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.

6.3 Example 3

This example is from LLVM, in the file `clang/lib/CodeGen/CGObjCGNU.cpp`. In Figure 11, 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.

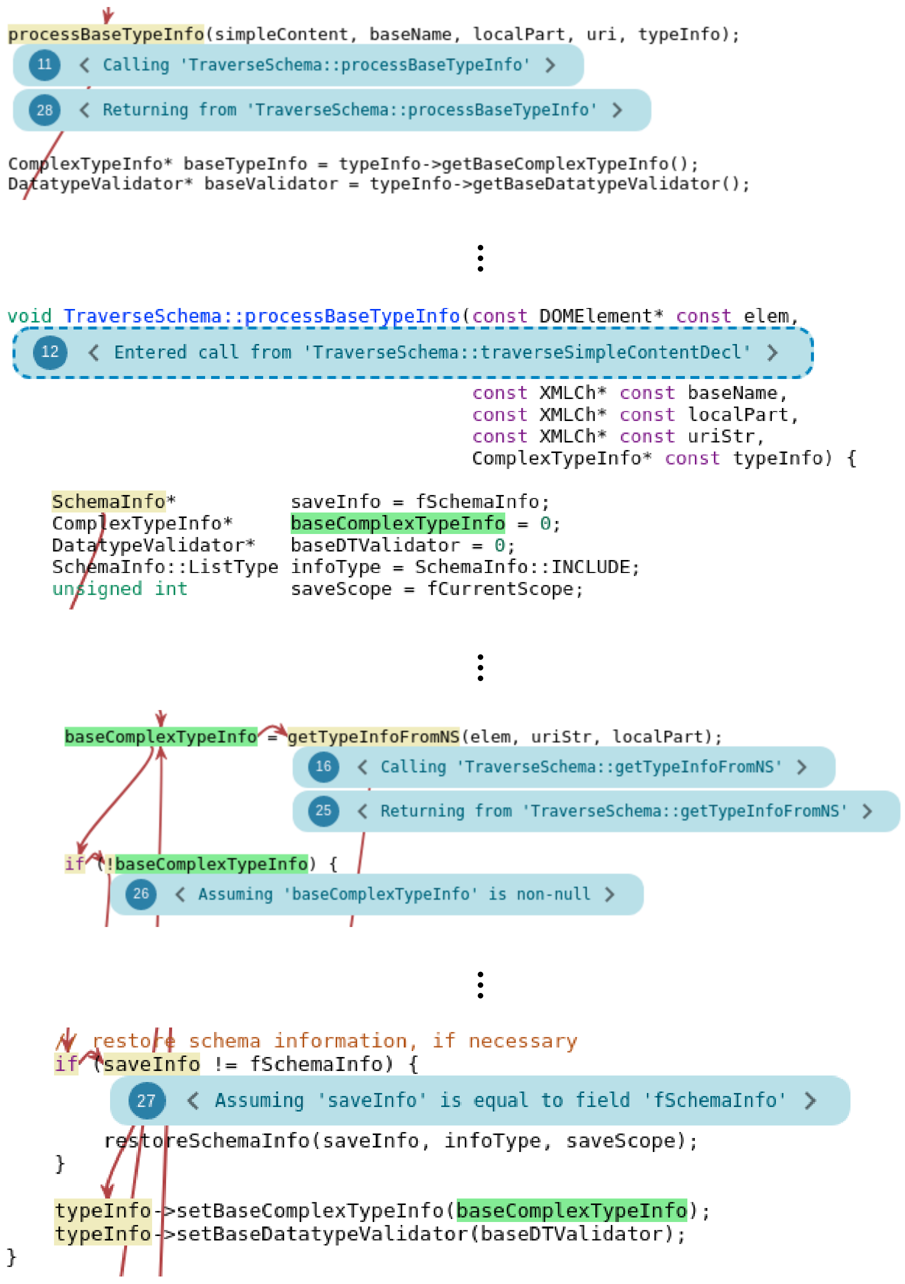
BEFORE:



```
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, XMLErrs::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 >
                && emptiableParticle(baseTypeInfo->getContentSpec()))) {
            simpleTypeRequired = true;
⋮
// Schema Errata E1-27
// Complex Type Definition Restriction OK: 2.2
if (simpleTypeRequired) {

    reportSchemaError(content, XMLUni::fgXMLErrDomain, XMLErrs::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(),
                          Called C++ object pointer is null
                        20 <
                          For more information see the checker documentation.
                        ((XSDElementNSImpl*) elem)->getColumnNo());

    fXSDErrorReporter.emitError(errorCode, msgDomain, fLocator);
}
```

Figure 8: Bug report before our improvement from Xerces

AFTER:



```
processBaseTypeInfo(simpleContent, baseName, localPart, uri, typeInfo);
  11 < Calling 'TraverseSchema::processBaseTypeInfo' >
  28 < Returning from 'TraverseSchema::processBaseTypeInfo' >

ComplexTypeInfo* baseTypeInfo = typeInfo->getBaseComplexTypeInfo();
DatatypeValidator* baseValidator = typeInfo->getBaseDatatypeValidator();

⋮

void TraverseSchema::processBaseTypeInfo(const DOMElement* const elem,
  12 < Entered call from 'TraverseSchema::traverseSimpleContentDecl' >
                                         const XMLCh* const baseName,
                                         const XMLCh* const localPart,
                                         const XMLCh* const uriStr,
                                         ComplexTypeInfo* const typeInfo) {

    SchemaInfo*          saveInfo = fSchemaInfo;
    ComplexTypeInfo*     baseComplexTypeInfo = 0;
    DatatypeValidator*   baseDTValidator = 0;
    SchemaInfo::ListType infoType = SchemaInfo::INCLUDE;
    unsigned int         saveScope = fCurrentScope;

⋮

     baseComplexTypeInfo = getTypeInfoFromNS(elem, uriStr, localPart);
                          16 < Calling 'TraverseSchema::getTypeInfoFromNS' >
                          25 < Returning from 'TraverseSchema::getTypeInfoFromNS' >

     if (!baseComplexTypeInfo) {
       26 < Assuming 'baseComplexTypeInfo' is non-null >

⋮

    // restore schema information, if necessary
    if (saveInfo != fSchemaInfo) {
      27 < Assuming 'saveInfo' is equal to field 'fSchemaInfo' >
        restoreSchemaInfo(saveInfo, infoType, saveScope);
    }

    typeInfo->setBaseComplexTypeInfo(baseComplexTypeInfo);
    typeInfo->setBaseDatatypeValidator(baseDTValidator);
}
```

Figure 9: New notes after our improvement in the report from Xerces.

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.

In Figure 11, 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.

7 Related work

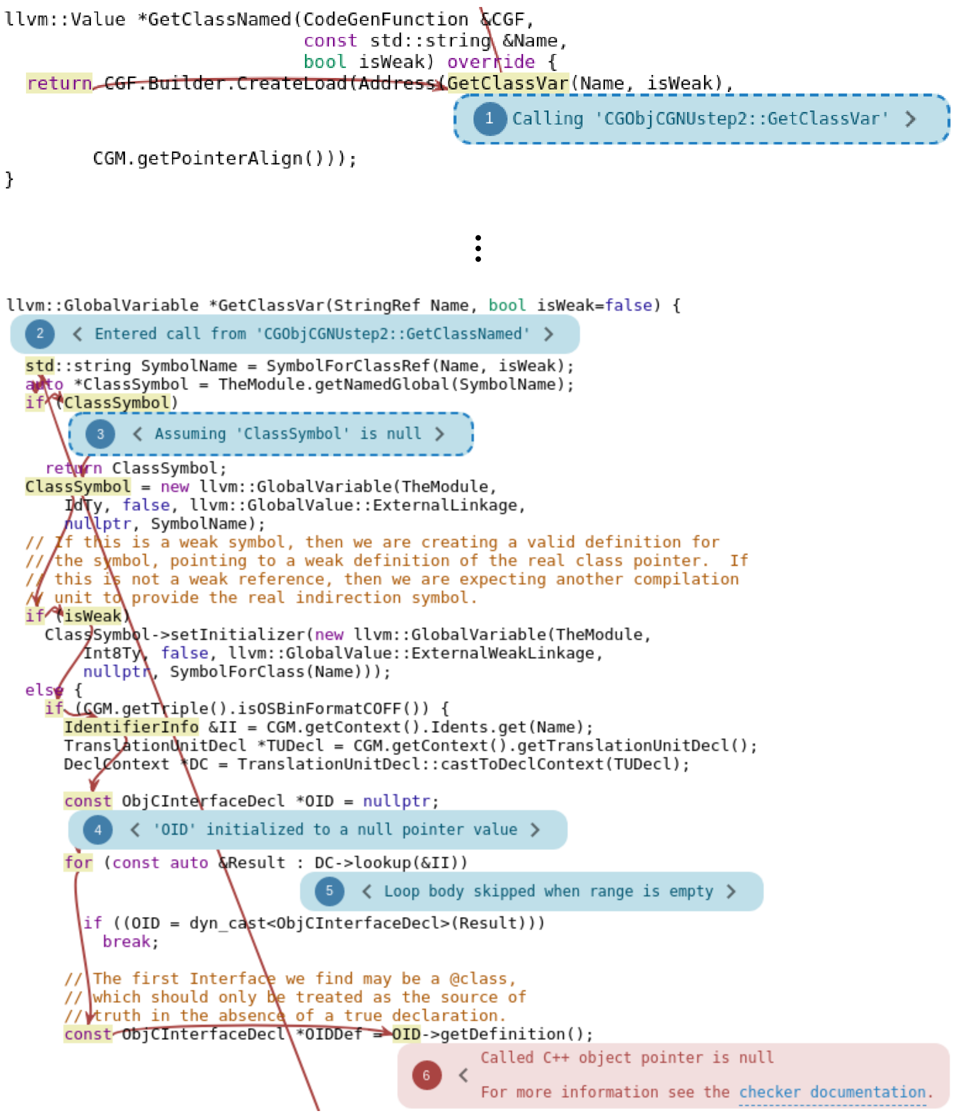
In 2008, the authors of paper [5] reported that while the static analysis methods are frequent research areas in the academy, there are no many usage examples in the industry. Current trends show a growing industrial interest of the static analysis tools [9, 61].

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 [45] 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 [28] 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

BEFORE:

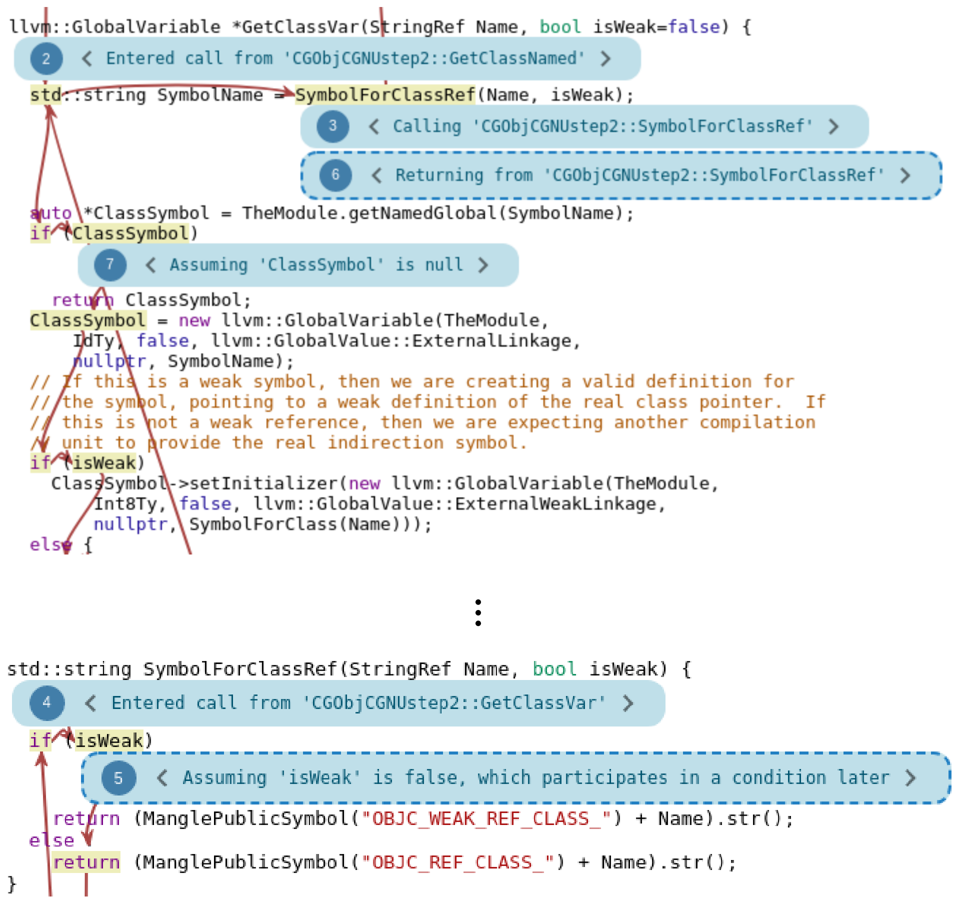


```
llvm::Value *GetClassNamed(CodeGenFunction &CGF,
                           const std::string &Name,
                           bool isWeak) override {
  return CGF.Builder.CreateLoad(Address(GetClassVar(Name, isWeak),
                                  1  Calling 'CGObjCGNUstep2::GetClassVar'  >

        CGM.getPointerAlign()));
}

                                 ⋮

llvm::GlobalVariable *GetClassVar(StringRef Name, bool isWeak=false) {
  2  < Entered call from 'CGObjCGNUstep2::GetClassNamed'  >
  std::string SymbolName = SymbolForClassRef(Name, isWeak);
  auto *ClassSymbol = TheModule.getNamedGlobal(SymbolName);
  if (ClassSymbol)
      3  < 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 {
    if (CGM.getTriple().isOSBinFormatCOFF()) {
      IdentifierInfo &II = CGM.getContext().Idents.get(Name);
      TranslationUnitDecl *TUDecl = CGM.getContext().getTranslationUnitDecl();
      DeclContext *DC = TranslationUnitDecl::castToDeclContext(TUDecl);

      const ObjCInterfaceDecl *OID = nullptr;
      4  < 'OID' initialized to a null pointer value  >
      for (const auto &Result : DC->lookup(&II))
                                5  < Loop body skipped when range is empty  >
        if ((OID = dyn_cast<ObjCInterfaceDecl>(Result)))
          break;

      // The first Interface we find may be a @class,
      // which should only be treated as the source of
      // truth in the absence of a true declaration.
      const ObjCInterfaceDecl *OIDDef = OID->getDefinition();
                             6  <  Called C++ object pointer is null
                                   For more information see the checker documentation.
```

Figure 10: Bug report before our improvement from LLVM.

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.

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

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.

We discussed that in the general case, the more complex a static analysis system is, the harder it is to construct intelligible reports for them. The authors in [46] discuss a static analysis technique with the usage of the preprocessor – a historically difficult concept to write good diagnostics around.

A similar methodology was conducted in the research published in paper [55]. 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.*

8 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 AST 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 large 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.

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

The authors in [13] discuss combining the Clang Static Analyzer with the dynamic symblic analyzer KLEE to refine the analysis. They highlight that traces provided by Clang are not that useful, and that Clang struggles to find non-trivial true positive. Maybe if the communication in between these tools improves (with the help of Clang itself better understanding the intetion of the programmer), research in this area could show new results as well.

9 Conclusion

Static analysis and symbolic execution specifically is a powerful technique to find deeply rooted programming errors. As an interprocedural path sensitive analysis, it gains 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.

In this paper, 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, tools such as the Clang Static Analyzer inspect the sequence of program states leading to the error, and construct 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. Our improved bug report generation facilities, which has been a part of the Clang Static Analyzer stable releases since version 10.0.0., demonstrates how the discovery of such information allows a tool to construct more comprehensive bug reports.

Acknowledgment

We would like to extend our heartfelt gratitude towards Artem Dergachev, who actively participated in every part of this research, including planning, implemeneting, measuring and evaluating.

References

[1] Aho, A., Sethi, R., and Ullman, J. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[2] Amazon Web Services. S2n, 2022. URL: https://github.com/awslabs/s2n/.

[3] Anders, M. and Michael, I. Static program analysis, 2012. URL: https://users-cs.au.dk/amoeller/spa/spa.pdf.

[4] Apache Software Foundation. Apache Xerces, 2022. URL: https://xerces.apache.org/.

[5] Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J., and Penix, J. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008. DOI: 10.1109/ms.2008.130.

[6] Bakken, A. Rtags, 2022. URL: http://www.rtags.net.

[7] Baldoni, R., Coppa, E., D'Elia, D., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):1–39, 2018. DOI: 10.1145/3182657.

[8] Bambini, M. Gravity, 2022. URL: https://github.com/marcobambini/gravity.

[9] Beller, M., Bholanath, R., McIntosh, S., and Zaidman, A. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016. DOI: 10.1109/saner.2016.105.

[10] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010. DOI: 10.1145/1646353.1646374.

[11] Binkley, D. and Harman, M. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings of the International Conference on Software Maintenance*, pages 44–53. IEEE, 2003. DOI: 10.1109/ICSM.2003.1235405.

[12] Boehm, B. and Basili, V. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001. DOI: 10.1109/2.962984.

[13] Busse, F., Gharat, P., Cadar, C., and Donaldson, A. Combining static analysis error traces with dynamic symbolic execution (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 568–579. ACM, 2022. DOI: 10.1145/3533767.3534384.

[14] Checker Developer Manual. Clang Static Analyzer: Checker Developer Manual, 2019. URL: https://clang-analyzer.llvm.org/checker_dev_manual.html (last accessed: 24-04-2023).

[15] Clang Static Analyzer, 2019. URL: https://clang-analyzer.llvm.org/.

[16] Clang-Tidy, 2019. URL: https://clang.llvm.org/extra/clang-tidy/ (last accessed: 24-04-2023).

[17] CodeSecure. CodeSonar, 2019. URL: https://codesecure.com/our-products/codesonar/ (last accessed: 15-02-2024).

[18] Cooper, K. and Torczon, L. *Engineering a compiler*. Elsevier, 2011. ISBN: 9780120884780.

[19] Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, F. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. DOI: 10.1145/115372.115320.

[20] Dergachev, A. Clang Static Analyzer: A checker developer's guide, 2016. URL: https://github.com/haoNoQ/clang-analyzer-guide (last accessed: 24-04-2023).

[21] Ericsson. CodeChecker, 2022. URL: https://github.com/Ericsson/codechecker.

[22] Google. Protobuf, 2022. URL: https://github.com/protocolbuffers/protobuf.

[23] gRPC Authors. grpc, 2022. URL: https://grpc.io/.

[24] Horváth, G. and Gehre, M. 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.

[25] Horváth, G. and Pataki, N. Categorization of C++ classes for static lifetime analysis. In *Proceedings of the 9th Balkan Conference on Informatics*, pages 1–7. ACM, 2019. DOI: 10.1145/3351556.3351559.

[26] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990. DOI: 10.1145/77606.77608.

[27] Hutchins, D., Ballman, A., and Sutherland, D. C/C++ thread safety analysis. In *Proceedings of the IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014. DOI: 10.1109/scam.2014.34.

[28] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering*, pages 672–681. IEEE, 2013. DOI: 10.1109/ICSE.2013.6606613.

[29] Khoo, Y., Foster, J., Hicks, M., and Sazawal, V. Path projection for user-centered static analysis tools. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, pages 57—-63, New York, NY, USA, 2008. Association for Computing Machinery. DOI: 10.1145/1512475.1512488.

[30] King, J. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. DOI: 10.1145/360248.360252.

[31] Kogut, J. TinyVM, 2022. URL: https://github.com/jakogut/tinyvm.

[32] Kovács, R., Horváth, G., and Porkoláb, Z. Detecting C++ lifetime errors with symbolic execution. In *Proceedings of the 9th Balkan Conference on Informatics*, pages 1–6, 2019. DOI: 10.1145/3351556.3351585.

[33] Kremenek, T. Finding software bugs with the Clang Static Analyzer. Apple Inc., 2008. URL: https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf.

[34] Lattner, C. LLVM and Clang: Next generation compiler technology, 2008. Lecture at BSD Conference. URL: https://llvm.org/pubs/2008-05-17-BSDCan-LLVMIntro.html.

[35] Lattner, C. and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 75–86. IEEE Computer Society, 2004. DOI: 10.1109/CGO.2004.1281665.

[36] Layman, L., Williams, L., and Amant, R. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 176–185. IEEE Computer Society, 2007. DOI: 10.1109/ESEM.2007.82.

[37] Marjamäki, D. CppCheck: A tool for static C/C++ code analysis, 2013. URL: http://cppcheck.sourceforge.net/.

[38] Nethercote, N. and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007. DOI: 10.1145/1273442.1250746.

[39] OpenSSL Software Foundation. OpenSSL, 2022. URL: https://openssl.org/.

[40] Ottenstein, K. and Ottenstein, L. 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*, pages 177–184. ACM Press, 1984. DOI: 10.1145/800020.808263.

[41] Park, J., Lim, I., and Ryu, S. Battles with false positives in static analysis of Javascript web applications in the wild. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering Companion*, pages 61–70. IEEE, 2016. URL: https://ieeexplore.ieee.org/document/7883289.

[42] Perforce. Klocwork, 2024. URL: https://www.perforce.com/products/klocwork (last accessed: 15-02-2024).

[43] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995. DOI: 10.1145/199448.199462.

[44] Rice, H. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953. DOI: 10.2307/1990888.

[45] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. Lessons from building static analysis tools at Google. *Communications of the ACM*, 61(4):58–66, 2018. DOI: 10.1145/3188720.

[46] Schubert, P., Gazzillo, P., Patterson, Z., Braha, J., Schiebel, F., Hermann, B., Wei, S., and Bodden, E. Static data-flow analysis for software product lines in C. *Automated Software Engineering*, 29(1), 2022. DOI: 10.1007/s10515-022-00333-1.

[47] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, Berkeley, CA, USA, 2012. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2342821.2342849.

[48] Sreedhar, V. and Gao, G. A linear time algorithm for placing φ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73. ACM Press, 1995. DOI: 10.1145/199448.199464.

[49] Sutter, H. Lifetime safety: Preventing common dangling. Technical report, Microsoft Corporation, 2018. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1179r1.pdf.

[50] Synopsys. Coverity, 2019. URL: https://scan.coverity.com/ (last accessed: 24-04-2023).

[51] The Bitcoin Core. Bitcoin Core, 2022. URL: https://bitcoincore.org/.

[52] Umann, K. The penultimate challange: Constructing bug reports in the Clang Static Analyzer. LLVM Developers' Meeting, San Jose, CA, 2019. URL: https://llvm.org/devmtg/2019-10/talk-abstracts.html#tech17.

[53] Umann, K. Enhancing bug reports in the Clang Static Analyzer, 2019. URL: https://szelethus.github.io/gsoc2019/ (last accessed: 24-04-2023).

[54] Umann, K. A survey of dataflow analyses in Clang, 2020. URL: https://lists.llvm.org/pipermail/cfe-dev/2020-October/066937.html (last accessed: 24-04-2023).

[55] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., and Gall, H. Context is king: The developer perspective on the usage of static analysis tools. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, pages 38–49. IEEE, 2018. DOI: 10.1109/SANER.2018.8330195.

[56] Vidács, L., Beszédes, ., and Gyimóthy, T. Combining preprocessor slicing with C/C++ language slicing. *Science of Computer Programming*, 74(7):399–413, 2009. DOI: 10.1016/j.scico.2009.02.003.

[57] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984. DOI: 10.1109/tse.1984.5010248.

[58] XGBoost Contributors. XGBoost, 2022. URL: https://xgboost.ai/.

[59] Xu, Z., Kremenek, T., and Zhang, J. A memory model for static analysis of C programs. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation — Volume Part I*, ISoLA'10, pages 535–548, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1939281.1939332.

[60] Zaks, A. and Rose, J. Building a checker in 24 hours, 2012. URL: https://www.youtube.com/watch?v=kdxlsP5QVPw.

[61] Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., and Di Penta, M. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. IEEE, 2017. DOI: 10.1109/msr.2017.2.