Effective Bug Detection with Unused Definitions

Li Zhong
lizhong@ucsd.edu
University of California, San Diego
USA

Chengcheng Xiang
c4xiang@ucsd.edu
University of California, San Diego
USA

Haochen Huang
hhuang@ucsd.edu
University of California, San Diego
USA

Bingyu Shen
byshen@ucsd.edu
University of California, San Diego
USA

Eric Mugnier
emugnier@ucsd.edu
University of California, San Diego
USA

Yuanyuan Zhou
yyzhou@ucsd.edu
University of California, San Diego
USA

Abstract

Unused definitions are values assigned to variables but not used. Since unused definitions are usually considered redundant code causing no severe consequences except for wasting CPU cycles, system developers usually treat them as mild warnings and simply remove them. In this paper, we reevaluate the effect of unused definitions and discover that some unused definitions could indicate non-trivial bugs like security issues or data corruption, which calls for more attention from developers.

Although there are existing techniques to detect unused definitions, it is still challenging to detect critical bugs from unused definitions because only a small proportion of unused definitions are real bugs. In this paper, we present a static analysis framework ValueCheck to address the challenges of detecting bugs from unused definitions. First, we make a unique observation that the unused definitions on the boundary of developers' interactions are prone to be bugs. Second, we summarize syntactic and semantic patterns where unused definitions are intentionally written, which should not be considered bugs. Third, to distill bugs from unused definitions, we adopt the code familiarity metrics from the software engineering field to rank the detected bugs, which enables developers to prioritize their focus.

We evaluate ValueCheck with large system software and libraries including Linux, MySQL, OpenSSL, and NFS-ganesha. ValueCheck helps detect 210 unknown bugs from these applications. 154 bugs are confirmed by developers. Compared to state-of-the-art tools, ValueCheck demonstrates to effectively detect bugs with low false positives.

EuroSys '24, April 22–25, 2024, Athens, Greece
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0437-6/24/04.
https://doi.org/10.1145/3627703.3629576

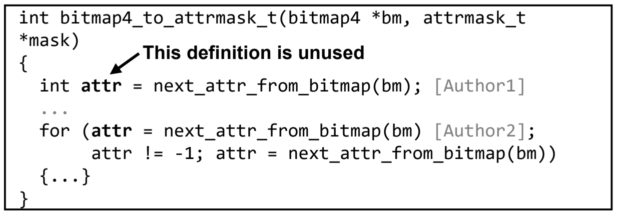
ACM Reference Format:
Li Zhong, Chengcheng Xiang, Haochen Huang, Bingyu Shen, Eric Mugnier, and Yuanyuan Zhou. 2024. Effective Bug Detection with Unused Definitions. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627703.3629576

1 Introduction

Unused definitions have been long regarded as redundant code in C/C++ programs. To be specific, a definition of variable v in programs refers to an occurrence of v on the left-hand side of an assignment statement, and a use indicates an occurrence of v on the right-hand side. If a variable is assigned with a value but the value is not used, that is an unused definition. Since unused definitions do not directly cause severe consequences, they are mostly regarded as bad code practices that do not require much attention. However, unused definitions could indicate deeper problems. When developers write an assignment, intuitively they should use that value from the assignment somewhere afterward. When the value assigned is not used, it violates this intuition, thus reflecting the inconsistency of developers' behavior [11, 27].

Unused definitions could reveal underlying critical bugs. Figure 1 gives two real-world examples from a widely-used open-source software, NFS-ganesha [5]. In the first example from Figure 1a, `attr` is assigned with the return value of `next_attr_from_bitmap`. However, this definition is soon overwritten and not used. This skips the first attribute of the source bitmap list without copying it to the destination bitmap list, causing a severe problem — important file attributes such as ownership are not copied properly to the destination bitmap, which is a security issue that can further invoke privilege escalation. Another example in Figure 1b shows a bug where the function argument `bufsz` in function `logfile_mod_open` is unused and overwritten, which is an implicit unused definition. The unused definition serves as a symptom indicating the code possibly contains a problem — In this example, `bufsz` originates from the configuration value 'logging buffer size'. If developers set the logging buffer size to zero, they would expect logs to be output immediately. However, since the value is subsequently overwritten with 1400 within the function, the configuration has no effect.

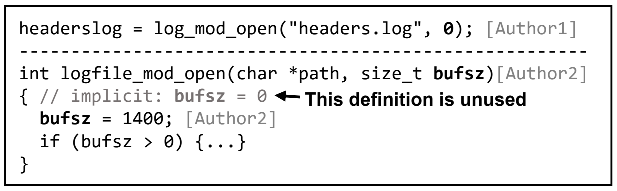
The actual access log buffer size is set to 1400 regardless of how the developer set it, resulting in unexpected memory consumption and wrong buffering behaviors.



```
int bitmap4_to_attrmask_t(bitmap4 *bm, attrmask_t
*mask)
{
  int attr = next_attr_from_bitmap(bm); [Author1]
  ...
  for (attr = next_attr_from_bitmap(bm) [Author2];
       attr != -1; attr = next_attr_from_bitmap(bm))
  {...}
}
```

This definition is unused

(a) **A Severe Bug Underlying Unused Definitions.** Function `bitmap4_to_attrmask_t()` converts NFSv4 attributes mask to the FSAL attribute mask. However, the first attribute returned by `next_attr_from_bitmap()` is unused and overwritten by another definition.Thus the first attribute doesn't get copied to the FSAL attribute mask, which can result in security issues related to file permissions.



```
headerslog = log_mod_open("headers.log", 0); [Author1]
-----------------------------------------------------
int logfile_mod_open(char *path, size_t bufsz)[Author2]
{ // implicit: bufsz = 0
  bufsz = 1400; [Author2]
  if (bufsz > 0) {...}
}
```

This definition is unused

(b) **A Configuration Bug Underlying Unused Definitions.** The developer of `logfile_mod_open` does not use value of `bufsz` but directly overwrite it with 1400, thus the value 0 assigned to `bufsz` is unused. This bug causes unexpected memory consumption and wrong buffering behaviors.

Figure 1. Two real-world bugs underlying unused definitions **detected by ValueCheck** with severe consequences like security issues and configuration bugs. Both bugs cross the author scope.

It is challenging to detect these bugs from unused definitions, although existing tools can detect unused definitions:

(1) Existing techniques only detect which definitions are unused but do not differentiate non-trivial bugs from simply redundant code. This is critical because non-trivial bugs require developers to diagnose more carefully than simply removing the redundant code. Without differentiation, detecting bugs from a large number of unused definitions will be impractical. Besides, existing techniques fail to detect unused definitions precisely and miss unused definitions that could be bugs. (2) Existing techniques do not consider the semantics of unused definitions. Some of the unused definitions could still express special meanings in the programs. They are written by developers intentionally and thus are not bug indications. (3) Existing techniques provide no priority ranking of the detected unused definitions. Since unused definitions are not directly the root cause of a bug but just bug indications, developers could spend more effort but detect fewer bugs without a way to distill bugs. As a result, existing tools report hundreds of unused definitions from a project with high false positives, which requires overwhelming effort from developers to check. We observe that a subset of developers choose to disable unused definition warnings in compilers. In the top 40 GitHub C++ repositories with the most stars, nearly half of them do not use these options in their default compiler configurations[1]. In search results of 'gcc unused' from StackOverflow [9], the largest online community for developers, the top-voted question looks for suggestions on how to silence unused definitions warnings.

In this paper, we design an effective and practical approach to detect, distill, and rank real bugs from a large number of unused definition candidates. Our approach addresses the above challenges based on three insights.

First, we make a unique observation that the unused definitions caused by code written by multiple developers, which we call *cross-scope unused definitions* in this paper, are more bug-prone. We randomly sample 42 unused definitions bugs from the history commits of 4 open source projects from 2019-2021 (c.f. § 3.1) and observe that majority of these bugs involve code written by two developers. The definition is written by one developer but is ignored/overwritten by another developer. We show two examples of such unused definitions. In Figure 1a, `attr` is written by author `Author1` but the definition is overwritten by author `Author2`. In Figure 1b, the function call is written by author `Author1` but the function is implemented by author `Author2`, overwriting the value provided by `Author1`. The data flow from definitions to uses in programs embeds the assumptions of developers on the programs. However, this implicit knowledge sometimes cannot be fully shared due to the lack of documentation and communication. Therefore, the unused definition on the boundary potentially indicates inconsistency between developers and a higher chance of being bugs. We take advantage of this observation to distill bug-prone unused definitions.

Second, we summarize several patterns of unused definitions that are intentionally written by developers in programs. We observe that not all unused definitions are redundant code in programs, some of which have semantics like moving a cursor or are intentionally written to keep compatibility. This kind of unused definition is not a true bug. Also, by mining the use pattern of definitions in similar contexts, which we call *peer definitions* in this paper, we can avoid reporting unused definitions that are not necessarily used. For example, return values of `printf()` usually require no checks and get ignored but do not harm. With these summarized patterns, we prune thousands of irrelevant unused definitions for detecting bugs, requiring no additional input from developers.

Third, we make the first attempt to apply code familiarity models in bug detection to prioritize the unused definitions

[1]We collect this data on April 9, 2022. We regard compilation configuration with '-Wall', '-Wunused-but-set-variable', '-Wunused-value' but without '-Wno-unused-*' options as detecting unused definitions.

that have a higher chance of being bugs. Since unused definitions are not directly linked to bug root causes but just indications of potential bugs, the false positive rate would be an internal drawback when applying them to detect bugs. Therefore, we hope to figure out a way to maximize the output with low developer efforts. The intuition behind the ranking algorithm is that unused definitions reflect the inconsistency in developers' coordination. A developer with lower expertise/familiarity with code may easily ignore the assumption and intercept the original data flow [22]. Research on code familiarity and expertise are applied to guide defect prediction and expert recommendation, etc. [26, 49, 63]. Existing code familiarity models take authorship [49], commit history [63], and other code editing activities [26] as metrics, which are obtainable from version control systems [35, 60]. However, few bug detection tools ever take advantage of these explorations from the software engineering field. We propose to prioritize code review on unused definitions that have a higher chance of bugs [18] with code familiarity models, instead of requiring the developers to check all of them under time pressure.

We implement a static analysis framework ValueCheck based on these insights, which detects cross-scope unused definitions, prunes false positives considering the semantics and developers' intention, then applies the code familiarity model to rank and prioritize detected unused definitions. The analysis is flow-sensitive, field-sensitive, alias-aware, and involves inter-procedural authorship analysis. We evaluate ValueCheck with four systems and libraries: Linux, MySQL, OpenSSL, and NFS-ganesha. ValueCheck helps detect 210 new bugs, among which **154 are confirmed and fixed by developers**, with a false positive rate of 26%. Compared to existing tools, ValueCheck demonstrates to practically help developers in detecting real bugs from unused definitions.

2 Background

2.1 Liveness Analysis

In this paper, we adopt the terminology defined in [12] to describe our algorithm.

Definition and Use. A definition of variable v in programs refers to an occurrence of v on the left-hand-side of an assignment statement[2]. A use indicates an occurrence of v on the right-hand-side.

Live Variable. A live variable is a variable that is assigned a value that is used in the future.

Liveness Analysis. Liveness analysis is a type of def-use analysis that identify whether a variable is live at certain points. It is a backward data flow analysis. It computes a live variable set from successors of this point and checks whether the variable has a use in this live variable set. If not, the variable is an unused definition. To formalize it as a dataflow problem, for each statement node n, $gen[n]$ is the set of variables used in n. $kill[n]$ is the variable defined in n. $in[n]$ and $out[n]$ are the live variable set before and after n. It has:

$$in[n] = out[n] - kill[n] \cup gen[n]$$
$$out[n] = \bigcup_{s \in succ(n)} in[s]$$

By computing the $in[n]$ and $out[n]$ iteratively based on the worklist algorithm, these sets will converge to a fixed point.

2.2 Unused Definitions

Detecting unused definitions has been regarded as compiler optimization [20, 36, 37, 62] for a long time and is extensively discussed in topics of dead variables [19, 29, 40, 73] and liveness analysis [51, 56]. This kind of optimization is even merged into modern compilers [41, 54]. The technique of detecting unused definitions is fully developed and merged into mainstream compilers [1, 7] to eliminate redundant computation in the generated assembly code and release the allocated registers. Besides, compilers also provide warning options at the source code level like '-Wunused-value', '-Wunused-variable', '-Wunused-but-set-parameter', etc. In this paper, we do not regard them as useless code that needs to be removed but as 'useful symptoms' that indicate bugs.

3 Design Overview

3.1 Detection Scope

Detecting critical bugs from a large number of unused definitions is non-trivial. To achieve this, we make a unique observation that unused definitions involving multiple developers are likely to be bugs. Therefore, we define the concept *cross-scope unused definitions* to help us pinpoint bugs, which includes the following scenarios: (1) Ignored/unused return value, where the developer that implements this function is different from the developer call this function. (2) Overwritten function argument value, where inside the function the argument value is overwritten but the function call is invoked by another developer. (3) Overwritten definitions, where the code of the old definition is written by one developer, and the code overwrites the old definition is implemented by other developers on all successor paths of the overwritten definition. The common characteristic of them is that the value of the definition is generated by one developer, but ignored/overwritten by another developer, causing an unused definition in the code. In all these scenarios, we only consider local variables in a function. Therefore, we do not consider concurrent access to shared variables since shared variables are global variables.

Our design is inspired by a preliminary experiment on collecting existing bugs related to cross-function definitions: We implement original liveness analysis and apply it to the snapshots of MySQL, NFS-ganesha, OpenSSL, and Linux on

[2]This is different from C++ concept of 'definition' where a definition provides a unique description of an entity.

the first commit of 2019 and 2021 separately. We collect unused definitions that were present in the 2019 version but were subsequently removed in the 2021 version. If a bug fix commit removes this unused definition, we investigate how this bug relates to the unused definitions. We identified a total of 325 unused definitions through differential comparison. To investigate their impact on bugs, we randomly selected 60 of these unused definitions by assigning them serial numbers and generating random numbers between 1-325. We manually checked the commit messages to ensure that the developers had addressed these unused definitions as part of the bug fixes, and we found 42 bug-related unused definitions. Notably, 39 out of the 42 instances crossed author scopes, indicating that these defects were located at the boundaries of developers' interaction. Building on this insight, we focused our attention on detecting cross-scope unused definitions in ValueCheck and explored their effectiveness in detecting bugs.

3.2 Framework Overview

ValueCheck detects cross-scope unused definitions in source code, prunes the false positives, and then ranks them by code familiarity to help detect bugs. Its workflow is shown in Figure 2. The generated ranked unused definitions can be checked by developers with given ranking as priority. ValueCheck faces three unique challenges to achieve its goal:

(1) How to completely and precisely detect cross-scope unused definitions in all scenarios? (Section 4) To overcome this challenge, we performed a more precise analysis to achieve better coverage than the state of art compilers [1, 7]). First, we conduct flow-sensitive liveness analysis to achieve a higher precision; Second, we propose a inter-procedural authorship analysis to identify cross-scope unused definitions which have constraints on the authorship of relevant code snippets within and across different functions; Third, we extend the detection scope to unused definitions of field variables (a field in a struct or class) by extending the detection algorithm of local unused definitions to be field-sensitive. Besides, we take advantage of the existing pointer analysis and def-use analysis framework to precisely obtain value flow information in the analysis.

(2) How to prune false positives from the large number of detected unused definitions? (Section 5) The main issue with existing solutions is the high false positive rate. This places a heavy burden on developers to manually check them to detect underlying non-trivial bugs. Therefore, ValueCheck needs effective approaches to prune false positives that are misreported by the analysis. We propose pruning approaches which trims false positives that have special meanings in the program, or are intentionally written by developers. The pruning requires no additional annotations from developers.

(3) How to rank up unused definitions that are more likely to be bugs? (Section 6) To reduce the effort from developers, ValueCheck adopts code familiarity models to rank the detected unused definitions. Our intuition is that for the cross-scope unused definitions, one of the developers introduces unused definitions into the code because they are not fully aware of the data flow in the program they touch. Therefore, for the developers with low familiarity, we rank the unused definitions introduced by them with high priority.

4 Detecting Cross-Scope Unused Definitions

4.1 Detect Local Unused Definitions

Liveness Analysis. Existing detection of unused definitions in compilers mostly relies on AST walking, which only reports a definition as unused when the variable is not referenced at all. However, the order of defines and uses could decide whether a definition is unused, which is flow-sensitive. Therefore, we detect unused definitions with a flow-sensitive liveness analysis in ValueCheck. It conducts analysis on the control flow graph of the function, starts from the end of the function, traverses each basic block backward and updates the live variable set based on the memory operations (load and store) on variables. To deal with loops in the function, we iterate the liveness analysis for several times until it reaches the fix point. The live variable set only records the existence of a use on the variable and cleans all the uses when traversing a definition. After the live variable set converges, we check each definitions in this function to see whether a use of this variable is in the live variable set. If not, we detect an unused definition. In this way, we can also check at the entry of a function whether a parameter is in the use set. If not, this parameter value is not used in this function, thus also an unused definition.

Indirect Function Call. In cases where the unused definition is generated by a function call, ValueCheck extracts the source code location of the called function to enable querying in the authorship lookup phase. When handling function pointers, ValueCheck checks the points-to set of the pointer to look up the corresponding functions. The pointee functions are treated as direct function calls in authorship lookup phase. Since ValueCheck focuses only on local variables, it does not need to delve into the callees in the analysis phase. Thus, there is no need for us to handle the recursive calls differently.

Pointer and Alias. To detect indirect access via pointers, we utilize pointer analysis and examine the point-to graph to determine whether the definition variable is included in the pointer-to sets of other variables. If the definition is referenced by pointers, it is considered possibly used through indirect reference and is therefore not marked

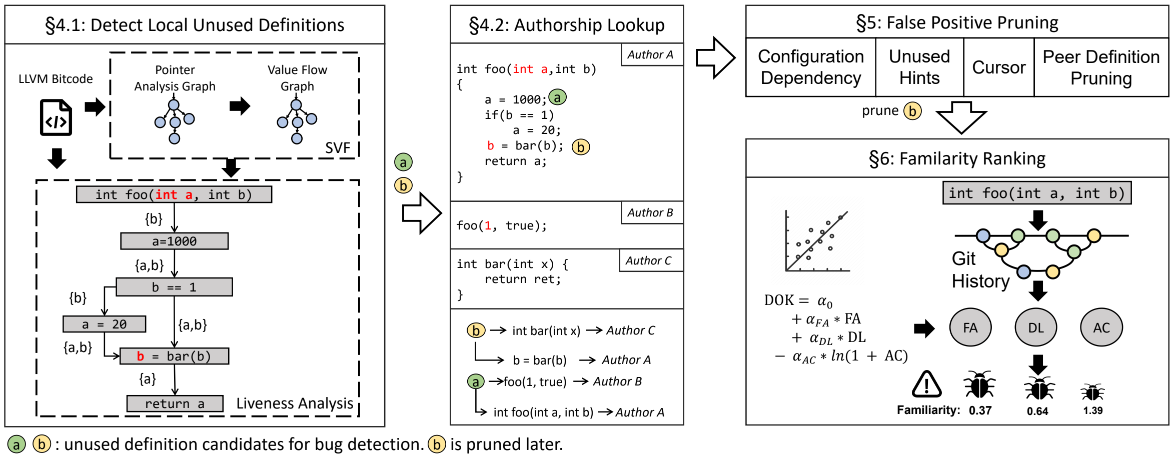


Figure 2. Overview of ValueCheck. ValueCheck consists of cross-scope unused definition detection, false positive pruning, and familiarity ranking. It conducts static analysis based on control flow graphs to detect unused definitions. Then with pruning, it rigorously prunes a large number of false positives. Next, ValueCheck computes the code familiarity of each cross-scope unused definition and ranks the candidates.

as an unused definition. We conduct field-sensitive Andersen's analysis [13] because its better scalability compared to flow-sensitive pointer analysis, while providing a small difference in help detecting unused definitions according to previous work [31]. To handle aliases of variables, we check the value-flow graph generated based on the point-to graph to see whether this definition is used somewhere else. If it has other use, this definition is not an unused definition.

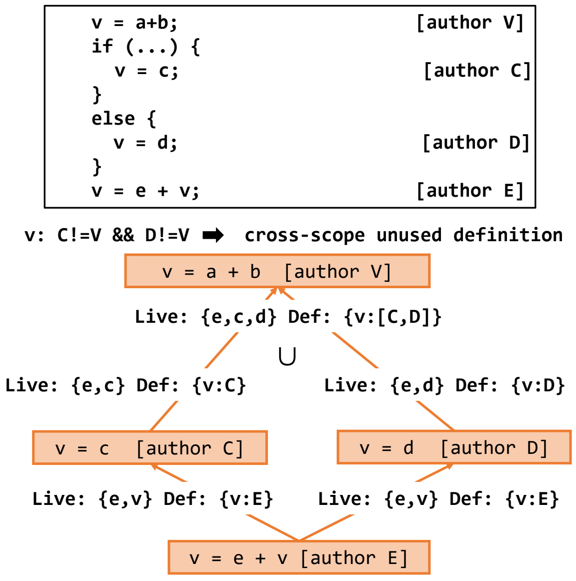

```
v = a+b;                  [author V]
if (...) {
  v = c;                  [author C]
}
else {
  v = d;                  [author D]
}
v = e + v;                [author E]
```

Figure 3. An Example of Define Set. The first definition of v is overwritten by other developers on all the successor paths.

4.2 Authorship Lookup

To decide whether an unused definition is a cross-scope unused definition, ValueCheck looks up the authorship information of the unused definitions it detects based on the three scenarios we discussed in Section 3.1:

(1) For the unused return value, we get the author of the call site first, assuming it is author D. Then we search the source code file of the callee and look up the authorship of the line that returns this value. However, a function may have multiple locations of returning a value, of which the authors are B_1, B_2, B_3... In this case, we got all these locations. If author B_1, B_2, B_3... are all different from D, we regard this as a cross-scope unused definition. If the callee is a library call not included in this project, we regard the author is different from D.

(2) For the unused function argument, we get the author of the call site, which we assume is author C, then look up the author B that defines the parameters of function F. If C and B are different, it is a cross-scope unused definition. If the parameter is overwritten inside the function F by developer D, we compare D to author C.

(3) To decide whether an unused definition is overwritten by other developers, ValueCheck needs to record additional information on whether another definition overwrites it in the successors of this definition. Therefore, we extend the original liveness analysis to maintain another set `define` aside from the live variable set. It records the last definition of variables we traverse and the corresponding authors. The define set follows the same update rule as the live variable set. We show an example in Figure 3. In this example, ValueCheck reversely traverses blocks in this function. Whenever there is a definition of variable v, it updates the author of v in the define set. For every block, it unions the define

Notation Table	
`getVar(v)`	Get variable names of a value
`getRetAuthor(F)`	Get authors of all return statements in function `F`
`checkAuthor(A, List, LiveSet)`	Check whether `A` is different from all items from `List` before adding the unused definition to `LiveSet`.
`updateDef(v, A, DefSet)`	Update the author of variable `v` to `A` in `DefSet`
`getCallSite(F)`	Get all call sites of function `F`
`getLine(A)`	Get the source code line of `A`
`reverse()`	Traverse the basic blocks or instructions reversely
`checkAlias()`	Check the point-to graph and the value flow graph for the variable aliases

function Load::Update(v1 = load v2, LiveSet, DefSet)

```
1: LiveSet.In.add(getVar(v2))
```

function Store::Update(I = store v1 v2, LiveSet, DefSet)

```
1: author1 = getAuthor(getLine(I))
2: if LiveSet.In.find(getVar(v2)) then
3:     LiveSet.In.remove(getVar(v2))
4: else
5:     checkAuthor(author1, DefSet[v2].author, LiveSet.Unused)
6:     if isRetVal(v1) then                ▷ Check unused return value
7:         author2 = getRetAuthor(getCall(v1))
8:         checkAuthor(author1, author2, LiveSet.Unused)
9: updateDef(getVar(v2), CurrAuthor, DefSet)
```

function computeCrossDef(F)

```
1: while Change do Change = False
2:     for BB in F.reverse() do                ▷ Traverse basic blocks
3:         OrigLiveSet = LiveSet[BB]
4:         LiveSet[BB].In = $\bigcup_{s \in BB.getSucc()}$ LiveSet[s].Out;
5:         DefSet[BB] = $\bigcup_{s \in BB.getSucc()}$ DefSet[s];
6:         for I in BB.reverse() do
7:             I.Update(I, LiveSet[BB], DefSet[BB]);
8:         if OrigLiveSet != LiveSet[BB] then
9:             Change = True                ▷ Iterate to handle loops
10: for A in F.args() do  ▷ Check unused definition of parameters
11:     EntryBB = F.getEntryBB()
12:     if !LiveSet[EntryBB].In.find(A) then
13:         for CS in getCallSite(F) do
14:             author1 = getAuthor(getLine(A))
15:             author2 = getAuthor(getLine(CS))
16:             checkAuthor(author1, author2, LiveSet.Unused)
17: if !LiveSet.Unused.empty() then
18:     checkAlias(LiveSet.Unused)
19: reportCrossUnused(F, LiveSet.Unused)
```

Figure 4. Cross-Scope Unused Definition Detection.

set of its successors. When an unused definition is detected, it checks its author against the authors in the define set to decide whether this is a cross-scope unused definition. Therefore, in this example, the first definition of `v` written by author V is a cross-scope unused definition.

Name	Code/IR Pattern

Configuration Dependency	/* Variable host is used when the config USE_ICMP is enabled. */
	 char **host**[10] = "127.0.0.1"
	 #if USE_ICMP
	 n = netdbLookupHost(**host**);
	 #endif

Cursor	/* The definition expresses the semantic of moving cursor. */ (**buf**++) = 'a';
Unused Hints	/* The unused definition is marked by developers with aware.*/ int do_flush_info(const bool **force** [[maybe_unused]]) {...}
Peer Definition Pruning	/* The unused definition is intentionally ignored by developers. */ printf("%d\n", num); // An implicit definition **[tmp]** = printf()

Table 1. Summary of pruning patterns in ValueCheck.

4.2.1 Algorithm Details.

Figure 4 shows our unused definition detection algorithm. To handle field-sensitive analysis, we check the value inside `getVar()`. If this value is loaded from a field of a struct variable v with offset n, we create a new variable name as the v_n to refer to this field. In this way, we can treat the field definitions similarly to other definitions. In this unified framework, `computeCrossDef()` computes the cross-scope unused definitions for each function. It traverses each basic block reversely. For each load instruction, it adds a use to the live variable set. For each store instruction, it removes all the use of this variable and if there is none, it detects an unused definition. In this case, it checks with `checkAuthor()` to see whether this is a definition overwritten by other developers or a return value written by other developers by `getRetAuthor()`. Besides, it will update `DefSet` by `updateDef()`. By iterating repeatedly on this function, the live variable set and the define set are guaranteed to converge. The cross-scope unused definitions in this function are reported.

5 Pruning

Not all cross-scope unused definitions are bugs. We observed that there are many cases where an unused definition is intentionally left in programs. Reporting them as bugs can introduce a high false positive rate. Based on our observation, we summarize four patterns for pruning as shown in Table 1.

5.1 Configuration Dependency

The use statements of some definitions could be controlled by preprocessor directives (e.g., `#if`), which may be disabled by the compilation configurations. In this case, static analysis may regard these definitions as unused because their uses are not compiled into IR. Therefore, ValueCheck looks into the corresponding source code of each definition and checks if there is any use of this definition enclosed by `#if`, `#ifdef`, `#ifndef` and `#endif` directives in the same function. If so, we prune this definition.

5.2 Cursor

As shown in Figure 5, after assigning a value to the memory region that the cursor `o` points to in Line 259, the code

```
static void dashes_to_underscores(...)
{
    char *o = output;
    if (c == '-')
        *o++ = '_';
    *o++ = '\0';
}
```

This unused definition is a cursor.

Figure 5. Example of Cursors.

increments o. This definition serves as program semantics "moving cursor" intentionally. Therefore, we regard these unused definitions are not bugs and prune them to reduce false positives based on the uses of a variable in the value flow graph. If a variable is incremented repeatedly by the same constant, ValueCheck considers it as a cursor and prunes it.

5.3 Unused Hints

In some cases, developers keep a definition unused for intended reasons. To hint these definitions are unused, the developers could add an unused attributes to them. We exclude them by matching the keyword 'unused' in the source code of these unused definitions.

5.4 Peer Definition Pruning

Sometimes, function calls are guaranteed to be successful or developers just don't care whether the call succeeds or not, resulting in the return value being unused. To quantify how much the developers 'care about' using the definition, we look at peer definitions of this definition. We define *peer definitions* as (1) For the definition of a function $ret = F()$ return value, peer definitions of ret are return values of other call sites of F. (2) For n_{th} parameter of function F, its peer definitions are the n_{th} parameter from functions with the same signatures. If the occurrences are over ten and over half of the peer definitions are not used, we will not report it.

6 Ranking based on Code Familiarity

We adopt the code familiarity model to help developers prioritize their effort in checking unused definitions that are more likely to be bugs. Even with rigorous pruning strategies, there are still unused definitions that require checks by developers, which could impose a workload on developers.

To deal with this, we propose to integrate the code familiarity model into ValueCheck, which helps to rank the unused definitions that are more likely to be bugs. Our intuition is that if the developer is not familiar with a certain snippet of code, he/she is more likely to cause some inconsistent behaviors in code. To measure the developers' code familiarity, the software engineering area has explored the code familiarity models for years. These models extract the familiarity metrics from the code contribution history to measure the developers' expertise. ValueCheck selects one representative work from the code familiarity area, the degree-of-knowledge (DOK) model [26], to measure code familiarity. The DOK model used in ValueCheck is:

$$\mathbf{DOK} = \alpha_0 + \alpha_{FA} * \mathbf{FA} + \alpha_{DL} * \mathbf{DL} - \alpha_{AC} * ln(1 + \mathbf{AC})$$

With this model, we select the author of each code line and compute this author's familiarity with the current file. **FA**, **DL**, and **AC** respectively represent first authorship, the number of deliveries from a developer, and deliveries to this file that are not authored by this developer. We count the commit numbers instead of committed lines because it is less resource-intensive and time-consuming. Based on prior literature [50], there is a strong correlation between commit numbers and commit line numbers. To obtain the weight in this model, we follow the steps of the original paper [26] to sample 40 source code lines from each application and ask the developers to self-rate their code familiarity (from 1-5) on these lines. Then we fit the linear model and get the weights, which are $\alpha_0 = 3.1$, $\alpha_{FA} = 1.2$, $\alpha_{DL} = 0.2$, $\alpha_{AC} = 0.5$. We apply this linear model in ValueCheck to compute the code familiarity.

The DOK model is chosen for two reasons. First, the DOK model is the most recent and representative model of code familiarity. It considers common factors that are mostly accessible in real-world software development. Second, the DOK model has generality to different applications. The three factors in DOK model, which are first authorship (FA), deliveries from a developer (DL), and deliveries to a code element that are not authored by this developer (AC), are language-independent and obtainable from most open source projects.

7 Implementation

Based on the design and techniques presented in Section 3, Section 4, Section 5 and Section 6, we implement the framework ValueCheck with LLVM-13.0.0 [6], SVF-2.6 [69] and python. It consists of four components:

Code analysis. The clang compiler compiles the source code into LLVM [41] bitcode. Then ValueCheck obtains point-to graph and sparse value flow graph based on SVF [69]. Since SVF takes a long time to analyze a whole large scale program, we apply ValueCheck on separate bitcode files generated by each single program file and only call SVF APIs to generates value flow graphs and the point-to graphs when we identify unused definitions within this bitcode file and want to further check aliases and indirect calls. It conducts analysis based on control flow graphs of each function, and accesses the def-use information of variables based on the sparse value flow graph and the point-to graph.

Authorship Lookup. This part is implemented in Python. It reads the meta information such as file names, function names, and line numbers of each unused definition. Then it reads the git meta files to look up authorship of unused definitions based on `GitPython` [2]. Then it compares the authorship and outputs cross-scope unused definitions.

False Positive Pruning. This component consists of two parts in LLVM and Python respectively. It checks the corresponding source code for each definition to see whether it should be pruned based on the pruning strategies. To match the unused hints and configuration dependency, we use `re` [8] library to do regex matching on code. For cursor and peer definition pruning, we collect all uses of variables and functions by LLVM API `getNumUses()`.

Familiarity Ranking. After pruning, ValueCheck computes FA, DL, and AC values for each unused definition by traversing the file commit log in git repositories with `GitPython` [2]. Lastly, ValueCheck outputs the unused definition report ranked by code familiarity.

8 Evaluation

In the evaluation, we answer the following questions:

1. How effective is ValueCheck in applying cross-scope unused definitions to help detect bugs?
2. What is the accuracy of detecting bugs in ValueCheck?
3. How does ValueCheck compare with existing tools?
4. What is the contribution of each component in ValueCheck?
5. How scalable is ValueCheck on a large code base?

8.1 Experiment Setup

8.1.1 Evaluated Applications. We mainly evaluate ValueCheck with four widely-used open-source system software and libraries, Linux-5.19, MySQL-8.0.21, OpenSSL-3.0.0, NFS-ganesha-4.46. These applications are selected with three criteria. First, they are popular real-world system projects of various types, which reflect how generally ValueCheck can be applied to real-world applications. Second, the source code of these applications is well-maintained and tested. The detected bugs are not from an immature program. Third, these applications have abundant version histories.

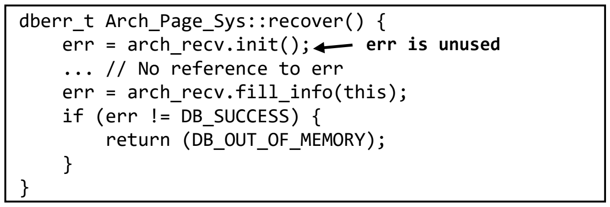
8.1.2 Evaluation Environment. All the experiments are conducted on a machine with 3GHz 6-core Intel i5-9500 CPU, 9216 KB cache, 16GB memory, and a 480GB SSD, which runs Ubuntu 18.04 with kernel 4.15.0. All applications are compiled with `-O0` and `-fno-inline` by clang-12 to retain source-level information. We compile each source object into separate bitcode files then perform analysis on these individual bitcodes. This helps reduce overhead of SVF from the inter-procedural analyses but does not affect the detection results since our detection target is local unused definitions.

Application	#Detected Bugs	#Confirmed Bugs
Linux	63	44
NFS-ganesha	22	18
MySQL	99	74
OpenSSL	26	18
Total	**210**	**154**

Table 2. The number of bugs newly detected by ValueCheck. Among the 210 bugs detected, 154 bugs are confirmed by developers.

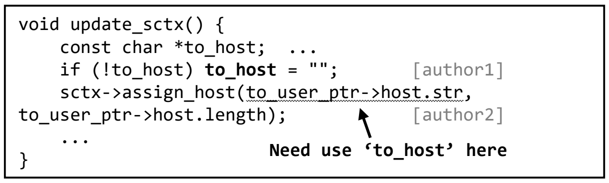
Bug Type	App.	Bug Description
Missing Check (134)	NFS-g	Unhandled ACL error
	MySQL	Missing sanity check
	MySQL	Unhandled error code
	OpenSSL	Malloc a negative size
	Linux	Fail to check device status
Semantic Bugs (20)	NFS-g	Ignore first bitmap attribute
	OpenSSL	Use the wrong master secret in TLS

Table 3. Bug examples detected by ValueCheck. Generally, ValueCheck detects two categories of bugs: missing check bugs and semantic bugs. Of 154 bugs confirmed, 134 are missing check bugs, 20 are semantic bugs.



```
dberr_t Arch_Page_Sys::recover() {
    err = arch_recv.init();  ←— err is unused
    ... // No reference to err
    err = arch_recv.fill_info(this);
    if (err != DB_SUCCESS) {
        return (DB_OUT_OF_MEMORY);
    }
}
```

(a) A Missing Check Bug Detected by ValueCheck. The error code returned from `init()` function is unchecked, which could result in the crash of page archiver recovery. It is a latent error that will corrupt data and cause failure in future execution, which is missed by the test suite.



```
void update_sctx() {
    const char *to_host;  ...
    if (!to_host) to_host = "";        [author1]
    sctx->assign_host(to_user_ptr->host.str,
to_user_ptr->host.length);             [author2]
    ...
                       Need use 'to_host' here
}
```

(b) A Semantic Bug Detected by ValueCheck. `to_host` is assigned a value but not used. It should have been used as the first parameter of `assign_host()`. Otherwise, the incorrect host address could corrupt the security context `sctx`.

Figure 6. Examples of New Bugs Detected by ValueCheck.

8.2 Detect New Bugs

8.2.1 Overall Results. ValueCheck detects 210 new bugs from cross-scope unused definitions in applications, among which 154 are confirmed by developers. We apply ValueCheck to the recent versions of the evaluated applications listed in

Section 8.1.1. We report bugs detected by ValueCheck to developers and the result is shown in Table 2.

8.2.2 Ethics and Responsible Disclosure. We take ethics in the highest standard regarding the new bugs we detect. We report all the bugs we detect to the developers though their official bug mailing list, clarify the potential impact of the bugs and help with the patches. We do not reveal the details of the bugs to any unofficial channels unless they are already fixed. In this paper, we anonymize all developers' names and identifiers and hide irrelevant details of the bug code that could be used to trace the authors.

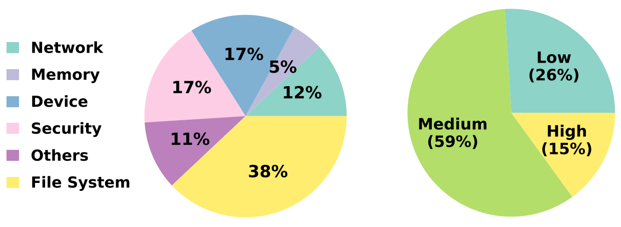
8.2.3 Bug Case Study. Table 3 shows several bugs we detected with ValueCheck. These bugs vary in terms of types and root causes. We categorize them into two types: 1) Missing check bugs — bugs that fail to check on function return values, parameters, or other variables, as the example shown in Figure 6a. This will make the following execution take the wrong assumption on completeness of certain operations and even cause corrupted data to be used by the program silently; 2) Semantic bugs — bugs that break specific program semantics. This will cause no runtime crash but the logic of the programs is wrong, as shown in Figure 6b. Some of them are hard to detect with existing solutions. For example, for the bug in Figure 6a, ValueCheck detects latent errors which could cause invisible symptoms and affect further execution but do not crash the programs immediately, which is hard for developers to detect pre-release by testing. The error in initializing the recovery mechanism of the page archiver could cause failure in future execution, which demonstrates the effectiveness of ValueCheck in detecting real-world non-trivial bugs.

8.2.4 Bug Categorization. To investigate when the 154 new bugs detected by ValueCheck arise and how they affect the applications, we classify them based on their distribution across software components, security severity, and the number of days it took to detect them. The results are illustrated in Figure 7.

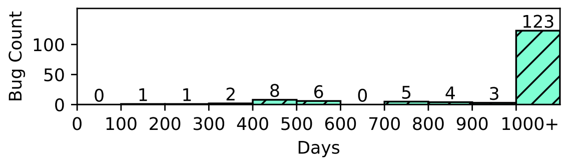
(1) Distribution. 38% of the cross-scope unused definition bugs we detected are related to file system, and 17% of the bugs are located in security modules such as authentication modules (Figure 7a).

(2) Security Severity. We categorize the severity levels assigned by developers to the bug reports. In cases that the severity level is not provided, we refer to the corresponding CWE. As shown in Figure 7b, 15% of bugs are of high severity and 59% are of medium severity, indicating that the bugs detected by ValueCheck can point to severe security issues like broken access control, data leak, etc.

(3) Days before Detected. From Figure 7c, more than 80% of the bugs had persisted in the code base for over 1000 days before we reported them and get confirmed, indicating a significant challenge in diagnosing and detecting these bugs. This suggests that ValueCheck is effective in detecting long-standing bugs that have gone undetected in the code base.



(a) Distribution of Bugs **(b)** Security Severity



(c) Days Before a Bug is Detected

Figure 7. Bug Categorized by Component Distribution, Security Severity and Days before Detected. All evaluated software undergo thorough testing. Despite this, ValueCheck uncovers high-severity bugs in critical components that have previously gone undetected for a long time.

8.3 Accuracy of ValueCheck

8.3.1 False Positives. ValueCheck has a low false positive rate (18%-30%) for detecting bugs if we only consider the bugs already confirmed by developers. These false positives come from three sources: (1) 51 false positives are still admitted by developers as minor defects in programs but not serious bugs. Sometimes the developers do not mark a definition as unused even though they know clearly it is no longer used. When we report them, some developers add the unused markers to avoid confusing other developers and the markers will help improve code readability. However, some developers just ignore them. For example, some return error codes are unused because the developers know the function call will not fail in this context. (2) 5 false positives are from debugging code and deprecated code but are not included in the release version. We detect them and report them as bugs since we compile all applications in debug mode and conduct the analysis on all functions. However, the developers do not put a high priority on these unused definitions.

8.3.2 False Negatives. We apply ValueCheck to detect the 39 existing bugs we collect in the preliminary experiments. ValueCheck successfully detects 37 existing bugs from them, namely 92.3% recall. 2 bugs are missed from the detection due to the peer definition pruning, which prunes the bugs when most of their peer definitions are unchecked.

App.	#Original	#Pruned (%Prune Rate)					#Detected After Pruned	% Prune False Negative (sampled)
		Config Dependency	Cursor	Unused Hints	Peer Definition	Total		
Linux	259	1 (0.39%)	22 (8.49%)	46 (17.76%)	127(49.03%)	196 (75.68%)	63	2%
NFS-g	898	7 (0.78%)	7 (0.78%)	839 (93.43%)	23 (2.56%)	876 (97.55%)	22	1%
MySQL	7743	37 (0.48%)	83 (1.07%)	3031 (39.15%)	4493 (58.03%)	7644 (98.72%)	99	3%
OpenSSL	642	18 (2.82%)	74 (11.60%)	322 (50.47%)	202 (31.66%)	616 (96.55%)	26	1%

Table 4. Prune rate breakdown and sampled false negative rate in ValueCheck. The false negative rate of pruning is less than 10% based on sampling with 95% confidence.

8.3.3 Prune Rate. To help understand the effectiveness of our pruning strategies, we present the breakdown of each pruning strategy in Table 4. Overall, ValueCheck's pruning strategies largely reduce the number of candidates for detecting bugs. For all the evaluated software, ValueCheck prunes between 75.68% to 98.72% of the cases, significantly reducing the burden of developers in reviewing all the potential bugs. Specifically, the unused hints and peer definition pruning strategies are found to be the most effective in reducing the number of candidates. For example, in MySQL, pruning eliminates over 7000 cases, with 98% due to these two pruning strategies. It's worth noting that the prune numbers are obtained from the pipeline of pruning as we showed in Figure 2, which means some false positives may match multiple patterns in our pruning but are pruned by the pruning strategies in the earlier stage. With powerful and aggressive pruning strategies, ValueCheck ensures that most of the detected cases are truly unused.

8.3.4 False Negatives of Pruning. To further evaluate the precision of the pruning strategy, we sample 100 cases from the unused definitions that get pruned from each application and compute the sampled false negatives, as shown in Table 4. We sampled pruned cases by assigning serial numbers to the cases in the order they were detected by ValueCheck. We generated random numbers within the serial number range and selected the pruned cases corresponding to these numbers. For each application, the false negative rate of pruning is less than 10%, which is statistically significant with 95% confidence. This suggests that the vast majority of cases that were pruned are indeed false positives. Among 7 false negative cases, 2 are due to the configuration dependency pruning, for which developers mark the variables with `(void)` to silent the unused warnings. However, this is not good practice of dealing with unused definitions. 5 are due to peer definitions pruning. Despite the potential false negatives, we still regard these pruning strategies as necessary because they effectively reduce the false positive rate to an acceptable level. According to experience from [17], developers tolerates false negatives better than false positives.

8.4 Comparison with Existing Tools

In this section, we compare ValueCheck to Clang, fb-infer, Smatch and Coverity on detecting bugs from unused definitions, as shown in Table 5.

8.4.1 Comparison with Clang. We compare ValueCheck to the compiler Clang. Maintainers of the evaluated applications periodically clean up code based on Clang warnings as indicated in their commit history. Therefore, no unused definitions are reported when we compile with the option '-Wunused'. Many unused definitions detected by ValueCheck but not by Clang is because Clang does not perform a precise analysis to detect unused definitions but just depends on recursive AST walking. It follows gcc as the specification and only detects a variable as unused when it never gets referred to on the right-hand side. Therefore, no bugs newly detected by ValueCheck are detected by Clang.

8.4.2 Comparison with fb-infer. FB-infer is a static analysis tool from Facebook. It can detect unused definitions that are referred to as "Dead Store" in its report, which we refer as 'Infer-unused'. Table 5 shows fb-infer detects fewer bugs than ValueCheck because they are incomplete in detecting all types of unused definitions in programs like overwritten/ignored arguments and field unused definitions. Also, fb-infer has a much higher false positive rate than ValueCheck. The false positives come from the following reasons: (1) fb-infer reports many unused definitions that are not cross-scope. When developers call the function written by themselves, they usually have the sense of when to use the parameter and the return value and when not. Therefore, they typically do not confirm unused definitions that are not cross-scope as bugs. (2) Cursor assignments, which are not excluded from fb-infer results. In our sampling, all the true bugs detected by fb-infer are also detected by ValueCheck. ValueCheck detects more bugs from unused definitions with a lower false positive rate compared to fb-infer.

8.4.3 Comparison with Smatch. Smatch [4] is a static analysis tool for Linux based on AST. It reports warnings when bug patterns are matched. It helps kernel developers detect thousands of bugs in kernel [3].

Smatch detects fewer bugs with a higher false positive rate from unused definitions compared to ValueCheck. Smatch detects one type of unused definitions: the return value of a function is unused. We refer the unused definition bugs detected by Smatch as Smatch-unused. We run Smatch on the evaluated software. However, Smatch-unused reports compilation error on all applications except Linux. Therefore, we only compare the result of Linux: Smatch-unused detects 28 real bugs compared to 154 real bugs detected by ValueCheck, with a false positive rate of 81%. The bug number is lower and the false positive rate is higher than ValueCheck due to two reasons: (1) It only detects unused return values among unused definitions. Besides, due to inlining, some unused return values are inlined as unused assignments, thus are not detected by Smatch-unused. (2) It conducts analysis based on the AST parser instead of control flow analysis, so the analysis is not precise and has high false positives.

8.4.4 Comparison with Coverity Scan.

Coverity is a static analysis tool that can detect defects in C/C++ projects, which is a commercial tool. We apply for its basic version Coverity Scan and evaluate it on the four projects. Coverity Scan two types of unused definition bugs: unused value and unchecked return value (unused return value is a subset). We call the bugs detected by Coverity Scan from unused definitions as Coverity-unused. Coverity-unused detects 170 bugs with a total false positive rate 62% from four applications. For Linux, though Coverity-unused detects more bugs, it **misses 35 bugs** detected by ValueCheck. For the other applications, their commit history shows developers of some evaluated applications previously utilized Coverity and addressed its warnings, which explains why Coverity detects much less bugs. ValueCheck can detect new bugs with lower false positive rate because: (1) Coverity-unused only detects unused assignment and unused return value, excluding other types of unused definitions (e.g. assigned but unused arguments). Besides, to avoid the huge number of unpruned results, it infers whether function return values need be used based on the percentage of used return values. If the function is only used once, it cannot correctly infer whether the return value should be used. Compared to Coverity-unused, ValueCheck additionally considers authorship when deciding whether an unused definition should have been used, which is not limited by the number of function invocations. (2) Coverity-unused pruning does not consider any authorship information and code semantics, so it does not prune unused definitions that are intentionally left in the code, resulting in higher total false positives.

8.4.5 Case Study: a bug detected by ValueCheck but missed by other tools .

Figure 8 shows a bug example that fb-infer, Smatch-unused and Coverity-unused fail to detect. Since the variable `ret` is referred in `if(ret)`, all definitions of `ret` are regarded as used by fb-infer and Smatch-unused due to their inaccurate analysis. Besides, Coverity-unused does not report it as a bug because it fails to infer that the return value of `get_permset` should be checked since it is only invoked once. However, this actually is a real bug acknowledged by developers with security concerns of broken access control when invalid permission set is read. In fact, the definition in line 237 was previously used. But after author2 committed line 239, it became an unused definition. When an unused definition spans multiple authors, it indicates such bugs, which can be identified using authorship information.

It is worth noting that ValueCheck is not a replacement of other tools but a complement. It focuses on precisely detecting bugs from cross-scope unused definitions. However, it does not detect unused definition bugs introduced by the same developers due to carelessness or other reasons. Existing tools report this type of bugs but with high false positives, which remains as an open problem to explore in the future.

Tool	#Found Bugs/#Real Bugs/%Bug False Positive				
	Linux	NFS-g	MySQL	OpenSSL	Total
Clang	0	0	0	0	**0**
Infer -unused	–*	8/2/75%	45/9/80%	13/3/77%	**66/14/79%**
Smatch -unused	147/28/81%	–*	–*	–*	**147/28/81%**
Coverity -unused	157/56/64%	3/3/0%	4/1/75%	6/4/33%	**170/64/62%**
ValueCheck	63/44/30%	22/18/18%	99/74/25%	26/18/31%	**210/154/26%**

Table 5. Unused Definition Bugs Detected by Clang, Infer, Smatch, Coverity and ValueCheck. ValueCheck in total detects more bugs with lower false positives than other tools. *Report errors during analysis.

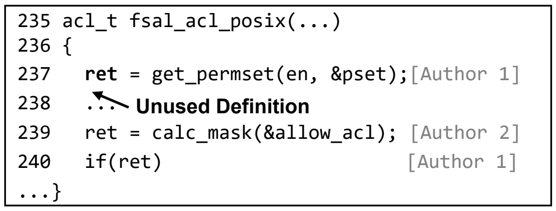

```
235 acl_t fsal_acl_posix(...)
236 {
237   ret = get_permset(en, &pset);[Author 1]
238   ...
239   ret = calc_mask(&allow_acl); [Author 2]
240   if(ret)                      [Author 1]
...}
```

Figure 8. A bug detected by ValueCheck but not detected by fb-infer, Smatch-unused and Coverity-unused. The developer forgot to handle the return error status from `get_permset`, which would cause access control error if the acl entry is invalid. Fb-infer and Smatch-unused fail to detect it due to inaccurate analysis where the variable `ret` is regarded as used, and Coverity-unused fails to infer the return value of `get_permset` need to be checked because it is only called once.

8.5 Authorship and Code Familiarity Effectiveness

8.5.1 Effectiveness of the Cross-Scope Authorship.

To explore how cross-scope authorship can help distill unused definitions that are real bugs from programs, we remove cross-scope filtering from ValueCheck and preserve all other components (w/o Authorship group). Then we report the top 20 bugs detected by the modified tool. The result is

App.	#Detected Bugs from Top 20 Bugs					
	ValueCheck	w/o Authorship	w/o Familiarity	w/o AC	w/o DL	w/o FA
Linux	**20**	14	16	20	19	20
NFS-g	**17**	2	16	17	16	17
MySQL	**20**	10	15	19	18	19
OpenSSL	**17**	2	15	17	16	17
Total	**74**	28 (-62%)	58 (-16%)	73 (-1%)	69 (-7%)	71 (-4%)

Table 6. Effect of authorship and the DOK model in ValueCheck. ValueCheck detects a higher total number of bugs compared to other groups.

presented in the second column of Table 6. Compared to original ValueCheck, the detected real bugs are much fewer, in total 28 bugs. This is because without cross-scope authorship filtering, the number of detected unused definition is much higher (2259 in total), for which pruning and ranking are insufficient to reduce false positives to an acceptable level.

8.5.2 Effectiveness of the DOK Model. To explore how effective the DOK model prioritizes bugs from detected unused definitions, we set up four groups: 1) (w/o Familarity) Remove the ranking from ValueCheck. Select the first 20 cross-scope unused definitions detected by ValueCheck from each application. 2) (w/o AC, w/o DL, w/o FA) Individually removing each factor from the code familiarity model and applying ValueCheck to get 20 cross-scope unused definitions with the lowest code familiarity. Table 6 shows the number of bugs detected by the four groups. In total, ValueCheck detects 74 existing bugs, 16% more than detecting without the familiarity model. Removing AC and DL factor decreases the total number of detected bugs and the precision of bug detection in the evaluated applications.

8.5.3 Bug Detection Precision of Different Cutoffs. We evaluate the precision of bugs detected by ValueCheck with different cutoffs on report numbers in Figure 9. When ValueCheck only reports the top 10 unused definitions with the lowest code familiarity from each applications, the precision of confirmed bug is the highest at 97.5%. With the increasing of the reported bug number, the precision of bug detection decreases, which indicates the relevance of code familiarity and the possibility of detecting bugs. From the result, it shows that the code familiarity model is effective in prioritizing real bugs.

8.6 Scalability of ValueCheck

As shown in Table 7, for each application, the execution time of ValueCheck on the whole application code base is under 30 min even for Linux with 27.8M LOC (we turn on `allmodconfig` compilation flag). Further, when integrating ValueCheck into the code testing and analysis process, this overhead could be reduced by running the analysis incrementally, i.e., only on the changed functions and the affected files in a commit. We do the incremental analysis on the

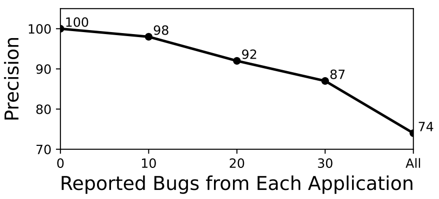


Figure 9. Precision of bug detection with different cutoffs after familarity ranking. ValueCheck has a precision of 97.5% when reporting the top 10 detected unused definitions with the lowest familiarity from each applications.

first 20 commits after 2022 on each application. The average execution time on each commit is under 5s for all the applications we evaluate. It empirically demonstrates that ValueCheck can be integrated into the code development with an acceptable time cost for a large-scale code base.

Application	#LOC	Time	Incremental Time
Linux	27.8M	28m12s	4.6s
NFS-ganesha	315K	2m13s	2.2s
MySQL	1.7M	16m32s	2.6s
OpenSSL	1.5M	3m54s	1.9s
Total	**31.3M**	**50m51s**	**11.3s**

Table 7. Scalability of ValueCheck.

9 Limitations and Discussion

9.1 Limitation

ValueCheck demonstrates to have better accuracy in detecting bugs from unused definitions compared to the state-of-the-art. However, ValueCheck still has limitations on having false positives. For example, some unused definitions are just legacy code or debugging, which could be further pruned by analyze the commit history and comments. But this will incur much more overhead so we do not prune this type of false positive. Besides, our exploration of cross-scope unused definitions is not an assertion that unused definitions which do not cross author scopes are not bugs. The assumption we make in the design of ValueCheck is based on our preliminary experiments (Section 3.1). Whether there are other types of unused definitions that are prone to be bugs is another problem to be explored in the future.

9.2 Alternatives of the DOK model

We use the DOK model because it is one of the state-of-the-art model and considers accessible factors from public code repositories. However, some alternative models could be considered, which may be less accurate but do not require the original developers to participate. The EA model [49] models the type of commits made by a developer, such as bug fixes, refactoring, and new functionality and assigns familiarity score to them differently. [52] can automatically infer developer expertise through their time to fix detects in commit histories. Another model [14] considers activities like comment and review which may also increase familiarity with the code. It is possible to replace with alternative familiarity models in ValueCheck.

10 Related Work

Bug Detection. A range of research has been proposed to detect bugs and vulnerabilities [65, 79] with formal method [21, 42, 67], static analysis [15, 33, 43, 44, 75, 80, 81] and automatic testing [39, 59, 66]. Some work directly infer rules from source code and picks up deviant outliers, which could be potential bugs [23, 34, 45, 47, 70, 76, 78]. Compared to them, our work detects a potential bug pattern, cross-scope unused definitions, which previously is regarded as redundant code, and demonstrates that it is possible to detect bugs from the unused definitions with low effort.

Code Familiarity. Previous work measure how familiar a developer is with a code snippet in a project by metrics include change history [32, 46, 49], file dependency [48], authorship [49], and interaction information [71]. [26] proposes the Degree-of-Knowledge (DOK) model to achieve a better measurement. In [74], the authors reveal the relationship between bug fixes and developer familiarity. Our work borrows the existing literature in this field to help reduce developers' efforts under time pressure. Some work rank the reports from static analysis tools with other methods like AdaBoost [58], which is orthogonal to our ranking method.

Inconsistent Code. Researchers studied the inconsistency in code in some past literature [16, 61, 72]. They focus on detecting unreachable code and removing them to reduce static analysis and coverage analysis effort, but not for bug detection while our work focuses on bug detection. Some previous work illustrates that certain code structures can indicate deeper problems in software design [24, 25, 28, 64], etc. Code smells are also related to the code quality and defect rate [30, 38, 55, 68, 77] of programs, which inspire our work to detect bugs from bad code patterns (unused definitions).

Dead Code Elimination As we already discussed in Section 2, eliminating unused definitions of registers has been regarded as a low-level code optimization [19, 20, 29, 36, 37, 40, 53, 57, 62, 73]. Unlike these previous works that simply remove all the redundant code, we propose to treat cross-scope unused definitions as potential bugs.

11 Conclusions

In this paper, we propose a practical method to detect cross-scope unused definitions, prune false bugs, and prioritize potential bugs in programs. We implement ValueCheck and evaluate it on open-source system software. By rigorously pruning as well as combining with code familiarity models, ValueCheck boosts the effectiveness of detecting bugs from unused definitions. ValueCheck detects 210 real-world new bugs, and 154 bugs confirmed by the developers. ValueCheck demonstrates how to take advantage of unused definitions to detect bugs with low effort. The artifact is available in [10].

References

[1] Diagnostic flags in Clang. https://clang.llvm.org/docs/DiagnosticsReference.html#wunused, 2007.

[2] GitPython Documentation — GitPython 3.1.12 documentation. https://gitpython.readthedocs.io/en/stable/index.html, 2015.

[3] Smatch: pluggable static analysis for C. https://lwn.net/Articles/691882/, 2016.

[4] Smatch the Source Matcher. https://smatch.sourceforge.net/, 2020.

[5] NFS-ganesha : User Space NFS and 9P File Server. https://nfs-ganesha.github.io/, 2021.

[6] The LLVM Compiler Infrastructure. https://llvm.org/, 2021.

[7] Warning Options (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#Warning-Options, 2021.

[8] re - Regular expression operations. https://docs.python.org/3/library/re.html, 2022.

[9] StackOverflow. https://stackoverflow.com/, 2022.

[10] FloridSleeves/ValueCheck. https://github.com/FloridSleeves/ValueCheck, 2023.

[11] Ahmadi, M., Farkhani, R. M., Williams, R., and Lu, L. Finding bugs using your own code: detecting functionally-similar yet inconsistent code. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2025–2040.

[12] Aho, A. V., Sethi, R., and Ullman, J. D. Compilers, principles, techniques. *Addison wesley 7*, 8 (1986), 9.

[13] Andersen, L. O. *Program analysis and specialization for the C programming language.* PhD thesis, Citeseer, 1994.

[14] Anvik, J., and Murphy, G. C. Determining implementation expertise from bug reports. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)* (2007), IEEE, pp. 2–2.

[15] Bai, J.-J., Li, T., and Hu, S.-M. {DLOS}: Effective static detection of deadlocks in {OS} kernels. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (2022), pp. 367–382.

[16] Bertolini, C., Schäf, M., and Schweitzer, P. Infeasible code detection. In *International Conference on Verified Software: Tools, Theories, Experiments* (2012), Springer, pp. 310–325.

[17] 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 (2010), 66–75.

[18] Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011), pp. 4–14.

[19] Briggs, P., and Cooper, K. D. Effective partial redundancy elimination. *ACM SIGPLAN Notices 29*, 6 (1994), 159–170.

[20] Chaitin, G. J. Register allocation & spilling via graph coloring. *ACM Sigplan Notices 17*, 6 (1982), 98–101.

[21] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 18–37.

[22] Claytor, L., and Servant, F. Understanding and leveraging developer inexpertise. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (2018), pp. 404–405.

[23] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 57–72.

[24] Fowler, M. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland* (1997).

[25] Fowler, M. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[26] Fritz, T., Ou, J., Murphy, G. C., and Murphy-Hill, E. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1* (2010), pp. 385–394.

[27] Gabel, M., Yang, J., Yu, Y., Goldszmidt, M., and Su, Z. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (2010), pp. 175–190.

[28] Giger, E., and Gall, H. Object-oriented design heuristics.

[29] Gupta, R., Benson, D., and Fang, J. Z. Path profile guided partial dead code elimination using predication. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques* (1997), IEEE, pp. 102–113.

[30] Hall, T., Zhang, M., Bowes, D., and Sun, Y. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM) 23*, 4 (2014), 1–39.

[31] Hind, M., and Pioli, A. Which pointer analysis should i use? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis* (2000), pp. 113–123.

[32] Hu, H., Zhang, H., Xuan, J., and Sun, W. Effective bug triage based on historical bug-fix information. In *2014 IEEE 25th International Symposium on Software Reliability Engineering* (2014), IEEE, pp. 122–132.

[33] Huang, H., Shen, B., Zhong, L., and Zhou, Y. Protecting data integrity of web applications with database constraints inferred from application code. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2023), pp. 632–645.

[34] Huang, H., Xiang, C., Zhong, L., and Zhou, Y. {PYLIVE}:{On-the-Fly} code change for python-based online services. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 349–363.

[35] Huang, Y., Zheng, Q., Chen, X., Xiong, Y., Liu, Z., and Luo, X. Mining version control system for automatically generating commit comment. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2017), IEEE, pp. 414–423.

[36] Hwang, Y.-S., and Saltz, J. Identifying def/use information of statements that construct and traverse dynamic recursive data structures. In *International Workshop on Languages and Compilers for Parallel Computing* (1997), Springer, pp. 131–145.

[37] Johnson, R., and Pingali, K. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation* (1993), pp. 78–89.

[38] Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., and Antoniol, G. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering 17*, 3 (2012), 243–275.

[39] Kim, S., Xu, M., Kashyap, S., Yoon, J., Xu, W., and Kim, T. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 147–161.

[40] Knoop, J., Rüthing, O., and Steffen, B. Partial dead code elimination. *ACM SIGPLAN Notices 29*, 6 (1994), 147–158.

[41] Lattner, C. Llvm and clang: Next generation compiler technology. In *The BSD conference* (2008), vol. 5.

[42] Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J. F., and Gunawi, H. S. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (USA, 2014), OSDI'14, USENIX Association, p. 399–414.

[43] Li, T., Bai, J.-J., Sui, Y., and Hu, S.-M. Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022), pp. 859–872.

[44] Lu, K., Pakki, A., and Wu, Q. Automatically identifying security

checks for detecting kernel semantic bugs. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part II 24* (2019), Springer, pp. 3–25.

[45] Lu, K. L., Pakki, A., and Wu, Q. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Conference on Security Symposium* (2019).

[46] McDonald, D. W., and Ackerman, M. S. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work* (2000), pp. 231–240.

[47] Min, C., Kashyap, S., Lee, B., Song, C., and Kim, T. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 361–377.

[48] Minto, S., and Murphy, G. C. Recommending emergent teams. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)* (2007), IEEE, pp. 5–5.

[49] Mockus, A., and Herbsleb, J. D. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002* (2002), IEEE, pp. 503–512.

[50] Munson, J. C., and Elbaum, S. G. Code churn: A measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (1998), IEEE, pp. 24–31.

[51] Muth, R. Register liveness analysis of executable code. *Manuscript, Dept. of Computer Science, The University of Arizona, Dec* (1998).

[52] Nguyen, T. T., Nguyen, T. N., Duesterwald, E., Klinger, T., and Santhanam, P. Inferring developer expertise through defect analysis. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 1297–1300.

[53] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis*. Springer Science & Business Media, 2004.

[54] Novillo, D. Gcc an architectural overview, current status, and future directions. In *Proceedings of the Linux Symposium* (2006), vol. 2, p. 185.

[55] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., and De Lucia, A. Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution* (2014), IEEE, pp. 101–110.

[56] Probst, M., Krall, A., and Scholz, B. Register liveness analysis for optimizing dynamic binary translation. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* (2002), IEEE, pp. 35–44.

[57] 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* (1995), pp. 49–61.

[58] Ribeiro, A., Meirelles, P., Lago, N., and Kon, F. Ranking warnings from multiple source code static analyzers via ensemble learning. In *Proceedings of the 15th International Symposium on Open Collaboration* (2019), pp. 1–10.

[59] Ridge, T., Sheets, D., Tuerk, T., Giugliano, A., Madhavapeddy, A., and Sewell, P. Sibylfs: formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 38–53.

[60] Ruparelia, N. B. The history of version control. *ACM SIGSOFT Software Engineering Notes 35*, 1 (2010), 5–9.

[61] Schäf, M., Schwartz-Narbonne, D., and Wies, T. Explaining inconsistent code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), pp. 521–531.

[62] Schneck, P. B. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference* (1973), pp. 106–113.

[63] Schuler, D., and Zimmermann, T. Mining usage expertise from version archives. In *Proceedings of the 2008 international working conference on Mining software repositories* (2008), pp. 121–124.

[64] Shatnawi, R., and Li, W. An investigation of bad smells in object-oriented design. In *Third International Conference on Information Technology: New Generations (ITNG'06)* (2006), IEEE, pp. 161–165.

[65] Shen, B. *Automatic Methods to Enhance Server Systems in Access Control Diagnosis*. University of California, San Diego, 2022.

[66] Shen, B., Shan, T., and Zhou, Y. Multiview: Finding blind spots in access-deny issues diagnosis. In *USENIX Security Symposium* (2023).

[67] Sigurbjarnarson, H., Bornholt, J., Torlak, E., and Wang, X. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (USA, 2016), OSDI'16, USENIX Association, p. 1–16.

[68] Sjøberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., and Dybå, T. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering 39*, 8 (2012), 1144–1156.

[69] Sui, Y., and Xue, J. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction* (2016), ACM, pp. 265–266.

[70] Tan, L., Zhang, X., Ma, X., Xiong, W., and Zhou, Y. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium* (2008), pp. 379–394.

[71] Thongtanunam, P., McIntosh, S., Hassan, A. E., and Iida, H. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering* (2016), pp. 1039–1050.

[72] Tomb, A., and Flanagan, C. Detecting inconsistencies via universal reachability analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (2012), pp. 287–297.

[73] Wand, M., and Siveroni, I. Constraint systems for useless variable elimination. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1999), pp. 291–302.

[74] Wang, C., Li, Y., Chen, L., Huang, W., Zhou, Y., and Xu, B. Examining the effects of developer familiarity on bug fixing. *Journal of Systems and Software 169* (2020), 110667.

[75] Wang, Y.-J., Yin, L.-Z., and Dong, W. Amchex: Accurate analysis of missing-check bugs for linux kernel. *Journal of Computer Science and Technology 36* (2021), 1325–1341.

[76] Xiang, C., Wu, Y., Shen, B., Shen, M., Huang, H., Xu, T., Zhou, Y., Moore, C., Jin, X., and Sheng, T. Towards continuous access control validation and forensics. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security* (2019), pp. 113–129.

[77] Yamashita, A., and Moonen, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 682–691.

[78] Zhang, T., Shen, W., Lee, D., Jung, C., Azab, A. M., and Wang, R. Pex: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Conference on Security Symposium* (USA, 2019), SEC'19, USENIX Association, p. 1205–1220.

[79] Zhong, L. A survey of prevent and detect access control vulnerabilities. *arXiv preprint arXiv:2304.10600* (2023).

[80] Zhong, L., and Wang, Z. Can chatgpt replace stackoverflow? a study on robustness and reliability of large language model code generation, 2023.

[81] Zhong, L., and Wang, Z. A study on robustness and reliability of large language model code generation. *arXiv preprint arXiv:2308.10335* (2023).

Appendix A ARTIFACT APPENDIX

A.1 Abstract

In the paper we propose to use cross-author unused definitions to detect bugs and prioritize bugs by its familarity.

Its workflow contains following steps:

- Using the compiled bitcode as input, ValueCheck applies liveness analysis to identify unused definitions and prune false positives.
- From the application code, ValueCheck extracts authorship information and capture cross-authorship relationship.
- By calculating the code familiarity, ValueCheck prioritizes the detected unused definitions.

We provide the source code, scripts, and other artifacts for the framework we presented in the paper. The artifact is available on GitHub at github.com/floridsleeves/ValueCheck. You can set up and use `git lfs clone` to download the repo to your local machine. In some new version of gits, `git clone` has been updated in upstream Git to have comparable speeds to `git lfs clone`.

A.2 Dependencies

- Linux (evaluated on Ubuntu 20.04)
- Python >=3.8
- SVF >= 2.7
- LLVM >= 12.0

A.3 Datasets

The artifact evaluates four open-source web applications. The scripts will automatically download their source code from GitHub and checkout the corresponding versions. The artifact includes the pre-compiled bitcode from each application by `wllvm` with flag `-fno-inline -O0` and `-g`. The bitcodes are broken into different separated modules to reduce the inter-procedural value analysis time of SVF.

A.4 Setup

We provide a script `install.sh` in the artifact to automatically install the dependencies and build the software. The details of setup is as follows:

- Install `cmake`, `gcc`, `g++`, `libtinfo5`, `libz-dev`, `zip`, `wget`.
- Setup SVF by script `setup.sh`. It will download `llvm` and build `SVF`. The SVF in the artifact is a modified version, in which we modify `CMakeLists.txt` to generate shared libraries.
- Install Python libraries `cxxfilt`, `gitpython`, `numpy`, `matplotlib`.
- Download git repositories of the evaluated applications and checkout to the corresponding versions.
- Compile ValueCheck. The generated libraries are under `build` directory.

A.5 Evaluation workflow

We provide a script `run.sh` to automatically perform the evaluation. It runs the framework on the four evaluated applications and generate bug reports with familarity ranking in the directory `result`. We also provide scripts to automate the evaluation and generate the Tables and numbers in Section 4. All the output will be in the `result` folder and contain the following key results:

- **result/table_2_detected_bugs.csv**: Total number of detected bugs from each application. (Table 2)
- **result/table_6_dok_effect.csv**: The number of detected bugs within top 20 bugs under different DOK settings. (Table 6)
- **result/figure_7_dist.pdf**: The category of bugs based on distribution, security, and days before detected. (Figure 7)
- **result/figure_9_detected_bug_dok.pdf**: The figure of reported bugs when increasing DOK rank. (Figure 9)
- **result/table_7_time_analysis.csv**: Time (seconds) to run the analysis. (First column of Table 7)
- In the **result/APP_NAME/** directory, `detected.csv` contains all the detected bugs.

Note that due to the differences in hardware environments, the performance results in Table 7 can be different from the numbers reported in the paper.