DOI: 10.15514/ISPRAS-2025-37(4)-4

Detection of Dead Function Calls as Source Code Defects through Static Analysis

[1,2] *V.I. Vasileva,* ORCID: 0009-0008-8542-6276 <*veravasilieva14@ispras.ru*>
[1] *A.E. Borodin,* ORCID: 0000-0003-3183-9821 <*alexey.borodin@ispras.ru*>
[1] *A.E. Volkov,* ORCID: 0000-0002-6043-5095 <*volkov@ispras.ru*>

[1] *Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. Detection of dead code (i.e. the code which is executed, but does not affect an observable program behavior) is commonly used by compilers as a part of optimization techniques for redundant code elimination. At the same time dead function calls might be seen as a kind of program source code defects, which may point to serious program logic faults. We describe a new detector for this kind of issues developed as a part of SVACE static defect detection tool, as well as the specific cases, which should be filtered out for practical detection of dead functions calls as program errors in contrast to their formal definition.

Keywords: static analysis; software defects; dead code; useless code; side effects; SVACE; C/C++ programming languages; Go programming language; symbolic execution; data-flow analysis; live variable analysis; interprocedural analysis.

For citation: Vasileva V.I., Borodin A.E., Volkov A.E. Detection of Dead Function Calls as Source Code Defects through Static Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 1, 2025. pp. 65-78. DOI: 10.15514/ISPRAS-2025-37(4)-4.

Поиск мёртвых вызовов функций как дефектов исходного кода методами статического анализа

[1,2] *В.И. Васильева*, ORCID: 0009-0008-8542-6276 <*veravasilieva14@ispras.ru*>
[1] *А.Е. Бородин*, ORCID: 0000-0003-3183-9821 <*alexey.borodin@ispras.ru*>
[1] *А.Е. Волков,* ORCID: 0000-0002-6043-5095 <*volkov@ispras.ru*>

[1] *Институт системного программирования РАН,*
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

[2] *Московский государственный университет имени М.В. Ломоносова,*
Россия, 119991, г. Москва, Ленинские горы, д. 1.

Аннотация. Обнаружение мёртвого кода традиционно используется в компиляторах как один из видов оптимизации с целью его удаления. Мёртвые вызовы функций могут рассматриваться и как вид программных дефектов, которые могут указывать на серьёзные ошибки в реализации логики программы. В статье описывается реализация нового детектора, разработанного как часть инструмента статического анализа SVACE, а также исследуются специфические случаи мёртвых вызовов, демонстрирующие особенности задачи поиска таких ситуаций в качестве программных ошибок по сравнению с формальным определением мёртвого кода.

Ключевые слова: статический анализ; дефекты программного обеспечения; мёртвый код; бесполезный код; побочные эффекты; анализатор SVACE; языки программирования C/C++; язык программирования Go; символьное выполнение; анализ потока данных; анализ живых переменных; межпроцедурный анализ.

Для цитирования: Васильева В.И., Бородин А.Е., Волков А.Е. Поиск мёртвых вызовов функций как дефектов исходного кода методами статического анализа. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 65–78 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(4)–4.

1. Introduction

In imperative programs, functions can be used both in a functional-like paradigm to calculate a return value, and like subroutines (procedures) to perform actions that will lead to some observable effects of a function call, i.e., aiming the use of their side effects.

If a function has no side effects, but is called in a way that the result of its calculations is not used, such a call is useless or has no effect. Since it wastes computational resources, a useless call itself is a sort of a source code defects, but what is more crucial, this code issue may indicate a serious fault in the program logic implementation.

Let's consider an example in Listing 1 which is based on a case we observed on an open-source projects (binutils).

As one can suppose, macro name TO_NEXT_ID might have misled the programmer, so the macro was used in the way like it would change the value of its argument, but in fact this code does not change it, despite the possible expectations of its author.

Dead function call is one of the possible kinds of *dead code*, and the detection of it for the purpose of its elimination is one of the common compiler optimization techniques [1, 2].

In order to avoid any terminological ambiguity, we would like to point out that use of the term *dead code* is often not quite consistent between the area of program static analysis algorithms (as well as compiler techniques) and source code defects descriptions. In this paper we use the term *dead code* to refer to the code that has no impact on the observable program behavior (which corresponds to its use for program static analysis). This meaning of dead code differs from the one for *unreachable code* and is a part of more general classes of *useless* or *redundant code*. In contrast, CWE[1] provides

[1] CWE™ – Common Weakness Enumeration

the following description in the related entry 'CWE-561: Dead Code' [3], quote: "Dead code is code that can never be executed in a running program. The surrounding code makes it impossible for a section of code to ever be executed".

In this paper we do not follow the latter definition and treat the issue described in CWE-561 as *unreachable code*.

```
int getNextId(int id) {
    if (id < 0 || id >= 256)
        return -1;
    return id + 1;
}

#define TO_NEXT_ID(x) getNextId(x)

void processId(int id,
               void (*acceptId)(int)) {
    TO_NEXT_ID(id);  // dead call to getNextId
    acceptId(id);
}
```

Listing 1. Example of a dead call pointing to an erroneous program logic.

The key point of our work is to consider dead function calls as potential source code defects and we describe an implementation of a detector to search such defect issues using static analysis methods.

Our experience with the detector we implemented for finding such cases demonstrates that not all the dead calls should be considered valuable as source code defects: very often the calls which are dead in formal sense and could be eliminated by the compiler during optimization, might appear due to idiomatic usage of certain programming language constructs (such as C++ templates) or specific architectural decisions (different function body implementations depending on compilation settings). Moreover, software developers might rely on the fact that these constructions will be optimized by the compiler, so an analysis that would provide warnings relevant to the programmer's perspective needs to be more specific in contrast to dead calls detection in general.

Thus, the essential feature of a detector for dead calls reported as errors is its ability to filter out the calls, which are dead formally, but have a questionable interest for the programmer (we refer to these issues as the issues, which are *true formally only*).

For this purpose, we will use an interprocedural analysis based on summaries, in which the determination of side effects occurs flow-insensitively. We will check that the return value of a function is not used using liveness analysis.

A detector for finding such erroneous issues we implemented as a new analyzer within the framework of SVACE [4, 5] static analysis tool. Our detector supports analysis of the programs written in C/C++ and Go.

2. Analysis Input Language (Internal Representation)

The low-level language we use as an internal representation for the input programs is the one of SSA-form[2]. Its instruction kinds cover different needs, such are, for instance, some programming languages built-in intrinsics. For clarity purposes we list below only the instructions kinds, which are relevant to our paper:

[2] SSA – Static Single-Assignment Form: a program representation where each variable is assigned exactly once [6].

- r := **alloca**() – allocate memory on program's call stack, write the pointer to the allocated memory to r;
- r := **load** p – read a value from the memory pointed by p, store this value to r;
- **store** v, p – write the value v to the memory pointed by p;
- r := v_1 op v_2 – apply operation op to arguments v_1 and v_2, write its result to r, where op is one of the following: +, -, *, /, ==, !=, >, <, >=, <=;
- r := **call** $func$(v_1, …, v_n) – call function $func$ and pass the values of the arguments $v_1...v_n$, where $func$ is either a function name, or a variable (pointer-to-function);
- **goto** L – unconditional jump to label L;
- **if** (v_1 op v_2) { $code_{true}$ } **else** { $code_{false}$ } – if *bool_expr* yields true, then execute $code_{true}$, else execute $code_{false}$, where op is one of the operations: ==, !=, >, <, >=, <=, and $code_{true}$ and $code_{false}$ are instruction lists;
- **return** v – terminate the current function execution and return the value of v to the caller's context.

Listing 2 demonstrates the representation of the program from the example in Listing 1.

```
def getNextId(id) {
    t1 := alloca()
    store id, t1
    t2 := load t1
    if (0 > t2) {
        return -1
    } else {
        if (256 <= t2) {
            return -1
        } else {
            t3 := t2 + 1
            return t3
        }
    }
}

def processId(id, acceptId) {
    t1 := alloca()
    t2 := alloca()
    store id, t1
    store acceptId, t2
    t3 := load t1
    t4 := load t2
    call getNextId(t3)
    call t4(t3)
    return
}
```

Listing 2. Example of program representation used for analysis.

3. Analysis for Dead Call Issues

3.1 Analysis Algorithm Overview

If a function execution modifies its local environment only (i.e., has no side effects), the only impact it may produce on the program execution is its return value, if it has one.

A dead function call (potentially useless from the perspective of a detector user) is a function call that has no side effects and whose result is not used further.

We will use a summary-based approach for interprocedural analysis implementation. It analyzes the functions behavior in the order of a *call graph* (CG) traversal and starts from its leaf nodes, i.e., the functions that do not call other functions. As a result of each function analysis, it creates its summary, which reflects relevant properties of the function's behavior. These summaries are used subsequently while processing calls to the analyzed functions.

During the CG traversal the engine runs side effect analysis, live variable analysis, and then the dead call issues detector for each processed function. The latter uses the results of the first two listed analyses to identify and report redundant calls.

3.2 Function Side Effects vs Function Call Side Effects

If a function has side effects, a particular call to it may have or (in contrast) may have no side effects as well. The latter can occur as a result of the conditions combination in the callee and the caller function context. We illustrate it in Listing 3.

A call to xlog function may have (line 12) or may have no (line 13) side effects. This fact depends on the value passed as verbosity argument and the actual definition of LOG_LEVEL macro (which is defined in this example explicitly, but may come from the compiler options in projects). In addition, this example demonstrates a case, where a true dead call issue might not be treated like a valuable defect issue. That is why in the current implementation we build our analysis to consider just the called function side effects for any call to it in a context-insensitive way.

```
 1  enum { NONE=0, ERROR=1, WARNING=2, INFO=3 };
 2
 3  #define LOG_LEVEL WARNING
 4
 5  void xlog(int verbosity, int eventCode) {
 6      if (verbosity <= LOG_LEVEL) {
 7          printf("LOG: %d\n", eventCode);
 8      }
 9  }
10
11  void demo() {
12      xlog(ERROR, 123);    // live call
13      xlog(INFO, 456);     // dead call
14  }
```

Listing 3. Calls with and without side effects to the same function.

3.3 Dead Call Analysis

The following cases may occur while analyzing a call to a function without side effects:

- The function does not have a return value. In this case the call is dead.
- The function has a return value, but it is not assigned to any variable. In this case the call is dead.
- The return value is assigned to an SSA variable. In this case use of live variable analysis results allows to check, if the variable is used anywhere, and if not, the call is dead.

The implemented detector identifies dead call as above, which are subject to report after additional checks to filter out issues that are irrelevant from the perspective of source code defects.

3.4 Side Effect Analysis

To determine if a function has no side effects we use a flow-insensitive analysis. A function does not have side effects, if all its instructions do not have side effects. The following instructions may have side effects:

- Function call (non-virtual). To determine if a function has side effects, the analysis uses the called function summary. If the summary does not specify the absence of side effects, the call is considered a one with side effects.
- Function call-by-pointer and virtual function calls. SVACE resolves these calls where it is possible to do [7]. In such cases, SVACE changes call instruction to the one with the resolved callee, thus the algorithm designed for direct function calls will be used then. In other cases, analysis conservatively considers these calls as the ones with side effects.
- Use of `store` instruction to write to memory. The instruction has a side effect, if the write occurs not to the function's local memory, i.e., memory that is not allocated by the `alloca` instruction. SVACE tracks variable aliases [8], this information is used to determine that variable is local. For cases, where it is not clear, variables are considered as non-locals.

4. Implementation

We implemented the algorithms described in the previous sections as a part of SVACE static analysis tool. SVACE supports summary-based analysis and provides a data-flow analysis engine [9], which is used to implements live variable analysis.

SVACE uses SVACE IR – an internal representation (IR) which is the same for all the supported input source code languages, the key features of SVACE IR were described in Section 2.

An important analysis feature is trace building, especially in the case of interprocedural analysis, otherwise the defects reported as its result might be unclear to users, and treated as false reports. In the case of the implemented detector, it might be not obvious, why a function does not have side effects, if it calls other functions.

The Listing 4 and the Listing 5 illustrate this and demonstrate a dead call issue reported by the implemented detector on BINUTILS (v2.22) project.

The implemented detector generates a warning as shown in Listing 6, which refers not only to the reported dead call itself at line 5, but contains information about the called functions.

```
1  static void
2  create_obj_attrs_section (void)
3  {
4    ...
5    frag_now_fix ();     // dead call
6    ...
7  }
```

Listing 4. Dead call at `as.c` from `binutils`.

```
#define obstack_next_free(h)  ((h)->next_free)

addressT
frag_now_fix_octets (void)
{
  if (now_seg == absolute_section)
    return abs_section_offset;
  return ((char *) obstack_next_free (&frchain_now->frch_obstack)
          - frag_now->fr_literal);
}

addressT
frag_now_fix (void)
{
  return frag_now_fix_octets () / OCTETS_PER_BYTE;
}
```

Listing 5. Interprocedural side effects absence at `frags.c` from `binutils`.

```
NO_EFFECT.CALL:
Call to function: 'frag_now_fix' has no effect at as.c:5.
    Function 'frag_now_fix' has no side effects at frags.c:16
    Call to 'frag_now_fix_octets' at frags.c:15
    Function 'frag_now_fix_octets' has no side effects at frags.c:10
```

Listing 6. An example of warning report

Examining the initial detector's results on open-source projects showed us that a big amount of the detected dead function calls are just the calls to functions with empty and trivial bodies – i.e., the functions which has no instructions in the body at all or only a single return instruction that returns a constant or value of one of the formal arguments. Very often these functions look like intended stubs; their execution obviously consumes almost no computational resources and, above all, the programmers most likely expect that these calls will be eliminated by the compiler optimizations. In the case of C/C++ these functions quite often are the result of conditional preprocessing (see Section 4.2). Thus, reporting these cases as program defects is of a questionable practical interest.

Consequently, we developed additional checks in the detector to filter out the cases mentioned above in order not to report them. We devote two subsections below to the specifics of the languages supported in the current implementation of the detector. The specific we described demonstrates that though the basic detection algorithms are the same, but some language-specific constructions and widely used idioms of their use, require some additional support, that is why the current implementations does not yet cover all the languages supported by Svace.

4.1 Go-specific Support

Go language has *channel* data type as a built-in language feature. It simplifies implementations related to *goroutines*[3] communications.

Go provides *read* and *write* operations (<-) for channels and select statement, which allows a goroutine to wait for multiple channel read/write operations. For Go-specific support we extend the Exemplary Language described in Section 2 with the corresponding instructions go_send, go_receive, and chan_select.

Consider the representation of these instructions in an exemplary language in Listing 7 and Listing 8.

```go
func receive(c chan int) int {
    return <-c
}

func send(c chan int, value int) {
    c <- value
}

func answer(in, out chan int) int {
    select {
    case t := <-in:
        return t
    case <-out:
        return 0
    default:
        return -1
    }
}
```

Listing 7. Example of channels use in Go programming language.

[3] Goroutine – a lightweight execution thread

```
def receive(c) {
    t1 := go_receive(c)
    return t1
}

def send(c, value) {
    go_send(c, value)
    return;
}

def answer(in, out) {
    t0 := chan_select(in, out)
    t1 := extract(t0, 0)
    if (0 != t1) {
        if (1 == t1) {
            t2 := extract(t0, 2)
            return 0
        } else {
            return -1
        }
    } else {
        t3 := extract(t0, 1)
        return t3
    }
}
```

Listing 8. IR for the example of Go program with channels use.

Any interaction with channels obviously produces a side effect, so our detector considers the related instructions of the internal representation (go_send, go_receive, and chan_select) as the ones with side effects.

Also Go allows to define anonymous functions inside another functions and allows the inner function to refer to local variables of its enclosing function, which forms a (*closure*). The captured variables used in these inner functions are considered as non-local.

4.2 C/C++-specific Support

C/C++ macros and certain idioms macros use introduce their own specifics to dead call issues detection.

Svace analyzes the IR built after the source code preprocessing, therefore only a part of the actual source code might be analyzed because of conditional compilation. So, the calls detected as dead since the called function has no side effects might be not dead under other compilation conditions, when the same function produces side effects (see Listing 9). Reporting these dead calls as defect issues is undesirable. A partial solution for this problem is filtering out dummy functions with empty bodies, as we mentioned earlier in this paper.

In addition, the C++ language has templates which can be instantiated with different data types. It may happen that not all of the instantiations of a function body, which has a call to another function, but only a part of them result to a formally dead call. The analysis, which analyses the IR produced for each function body instantiation, detects only that a call in a particular instantiation is dead. However, when a call in a particular template function instantiation to another template function is dead, some other instantiation may result for the same call invocation of another instantiation of the called function, which in turn has side effects. So, a dead call in a particular instantiation does not mean a true defect issue in terms of source code in this case. Therefore, warnings about unnecessary function calls are not generated by the detector inside templates.

We illustrate the case like described above in Listing 10.

```
PRStatus
nssArena_Shutdown(void)
{
    PRStatus rv = PR_SUCCESS;
#ifdef DEBUG
    rv = nssPointerTracker_finalize(&arena_pointer_tracker);
#endif
    return rv;
}

. . .
SECStatus
nss_Shutdown(void)
{
    . . .
    nssArena_Shutdown(); // questionable dead call defect
    . . .
}
```

Listing 9. Questionable dead call defect because of conditional compilation in `nss` *project.*

```
 1  class A {
 2      static int field;
 3  public:
 4      int foo() {
 5          return ++field;
 6      }
 7  };
 8
 9  class B {
10      static int field;
11  public:
12      int foo() {
13          return field + 1;
14      }
15  };
16
17  template <typename T>
18  void bar(T t) {
19      t.foo();   // dead call for T=B instantiation only
20  }
21
22  void demo() {
23      A a;
24      B b;
25      bar(a);    // live call
26      bar(b);    // dead call
27  }
```

Listing 10. Dead calls to a template function.

Method `foo()` of class A has side effects, since it changes variable `A::field`, in contrast to method `foo()` of class B. Svace traverses both template function `bar` instantiations: `bar<A>` instantiated for `T=A` and `bar<B>` for `T=B`. The call `t.foo()` is dead inside the `bar<A>` instantiation, but not inside `bar<B>` instantiation. So a warning for the call `t.foo()` at line 19 though true for `T=B`, but false for `T=A`, thus false in general as a defect report.

5. Similar Works

Most of the static analysis tools (as well as SVACE) provide detectors for a similar from the user perspective, but different problem: unused (or ignored) return values for a specified set of functions. The list of the functions to consider is either predefined in the detector, or a tool may provide a mechanism to specify them: configuration files to describe functions behavior or source code annotations (through syntax-supported attributes, comments, etc.)

For example, KLOCWORK (proprietary) checks (SV.RVT.RETVAL_NOTTESTED issue kind [10]) for ignored return value of socket, recv and a subset of pthread-family functions.

Another tool SONARSOURCE [11] (open source) has, for example, rule RSPEC-5308 for C: “Return value of “setuid” family of functions should always be checked” and rule RSPEC-5277 for C++: “Return value of “nodiscard” functions should not be ignored”.

An open-source SPLINT [12] covers C/C++ cases, documented as: “8.4.1 Statements with No Effects” and “8.4.2 Ignored Return Values” (in “8.4 Suspicious Statements”). Their approach is based on source code attributes – ‘pure’ attribute in particular.

Note, that a tool may cover this kind of issues not for all the supported languages. For example, SONARSOURCE (already mentioned above) has a rule very close to the logic of the detector we implemented: RSPEC-2201: “Methods without side effects should not have their return values ignored”, but for C# only.

All this makes any noteworthy tools comparison quite complicated.

The closest analog for the detector described in this paper is most likely the one implemented in COVERITYSCAN. Though it is a proprietary static analysis tool and COVERITY does not provide a free access even to the list of the defect kinds, which their tool is able to detect, one of the success stories they published online [13] gives evidence of a USELESS_CALL issue detected by COVERITYSCAN on ScummVM project (file tattoo_journal.cpp), we quote a tiny part of it in Listing 11.

Commit 95884c3 in project ScummVM [14] with the message “SHERLOCK RT: Actually, clear screen instead of a useless call. CID 1308097”, which fixes this problem, is shown in Listing 12.

```
<<< CID 1308097: Incorrect expression USELESS_CALL
<<< Calling "screen->empty()" is only useful for its return value,
    which is ignored.

        screen.empty();
```

Listing 11. USELESS_CALL reported by COVERITY.

It is one more example from the open-source projects, which demonstrates that a dead code issue may indicate more serious logic faults of the analyzed program.

6. Results and Discussion

We tested the detector we have developed on the source code of open-source projects. Our measurements showed that the contribution of the new detector almost does not affect the total SVACE analysis time (it is just comparable to measurements precision).

6.1 Results on Go Projects

For the testing on Go projects we used a collection of relatively small- to medium-size projects. The detector reported 14 issues, 71% are true issues and are potential candidates to fix, while the rest is most likely of low interest for a programmer though still indicate true dead calls.

The detailed results for Go projects are displayed in Table 1. The size listed in the table includes the lines of code of analyzed dependencies (dependency libraries/projects) and for technical reasons is calculated by SVACE as a total size only.

An example of an actual dead call defect found by the detector in xgb project is shown in Listing 13. The call to Pad() method at line 12 is dead.

```
engines/sherlock/tattoo/tattoo_journal.cpp
. . .
@@ -66,7 +66,7 @@ void TattooJournal::show() {

        // Set screen to black, and set background
        screen._backBuffer1.SHblitFrom((*_journalImages)[0], Common::
            Point(0, 0));
-       screen.empty();
+       screen.clear();
        screen.setPalette(palette);

        if (_journal.empty()) {
```

Listing 12. Fix of USELESS_CALL in ScummVM project.

```
1  func Pad(n int) int {
2     return (n + 3) & ^3
3  }
4
5  func changeGCRequest(c *xgb.Conn, Gc Gcontext, ValueMask uint32,
6                       ValueList []uint32) []byte {
7     size := xgb.Pad((8 + (4 + xgb.Pad((4
8                     * xgb.PopCount(int(ValueMask)))))))
9     b := 0
10    buf := make([]byte, size)
11    ...
12    b = xgb.Pad(b) // detected dead call
13    return buf
14 }
```

Listing 13. Dead call detected in xgb project.

Table 1. Reports on Go project.

project	true formally only	true fix candidates	false	size (KLOC)
oakmound/oak	1	0	0	
tikv/tikv	0	2	0	
pingcap/tidb	1	0	0	
xgb	0	8	0	
jaeger-client-go	1	0	0	
go-redis	1	0	0	
Total:	4	10	0	>3929

6.2 Results on C/C++ Projects

The detector emitted 125 warnings on the analyzed C/C++ projects. 13 of them are true, 6 are false positives. Others though true in formal sense, but are of questionable interest for the users.

The warnings that are true formally only, relate to calls to functions, which definitions use macros and depend on conditional compilation.

False issue reports are the result of the current internal representation specifics for some built-in functions that lacks some details for them. These details are of no importance for other analysises, but are crucial in this case for ours. Thus, 6 false reports are caused by live variable analysis which does not get information to determine that the variables passed to __builtin_alloca are live. We plan to extend the representation used in SVACE in order to fix it.

The detailed results for C/C++ projects are shown in “Table 2”.

An example of an actual dead call defect found by the detector in binutils (v2.22) project is shown in Listing 14. The call to _bfd_mips_elf_sign_extend() at line 19 is dead.

One more example was shown in Listing 4 in Section 4.

Table 2. Reports on C/C++ projects.

project	true formally only	true fix candidates	false	size (KLOC)
binutils	8	2	0	1078.6
gnupg	3	0	0	157.5
gst-plugins-good	32	0	0	566.2
openssl	0	1	0	366.7
xorg-server	2	1	0	602.4
libxml	60	0	0	303.4
nss	2	0	0	617.6
gtk+	0	1	0	915.9
gcc	18	8	6	5390.2
Total:	125	13	6	9998.5

7. Conclusion and Future Work

As the result of our work, we developed a new detector which is a part of SVACE static analysis tool. The detector is able to find dead call issues relevant as potential source code defects. The introduced analysis has no observable impact on the overall SVACE analysis time.

We studied the results of the initial implementation versions of our detector and discovered the cases which are true in the sense of dead call definition, but are of questionable interest for software developers. Subsequently, as a significant part of the current version of the detector we implemented the algorithms to exclude the most rubbish cases from the detector output. An important part of these cases is language-specific and covers now C/C++ and Go languages. The rest of the discovered dead call cases, which are detected by the current implementation of the detector, but are of low interest as source code defect issues are the subject for the related future work and the detector improvements. In addition to that we plan to adjust the detector for other languages supported by SVACE, since their particular features and use of certain language idioms require specific support as our preliminary experiments have demonstrated to us.

```
bfd_vma
_bfd_mips_elf_sign_extend (bfd_vma value, int bits)
{
  if (value & ((bfd_vma) 1 << (bits - 1)))
    /* VALUE is negative.  */
    value |= ((bfd_vma) - 1) << bits;
  return value;
}

bfd_reloc_status_type
_bfd_mips_elf_gprel16_with_gp (bfd *abfd, asymbol *symbol,
    arelent *reloc_entry, asection *input_section,
    bfd_boolean relocatable, void *data, bfd_vma gp)
{
  bfd_vma relocation;
  bfd_signed_vma val;

  val = reloc_entry->addend;
  _bfd_mips_elf_sign_extend (val, 16);   // detected dead call

  . . .

  return bfd_reloc_ok;
}
```

Listing 14. Dead call detected in binutils.

References

[1]. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: Principles, techniques, and tools (2 ed.). Addison Wesley, Boston, 2007, p. 535.

[2]. [gcc.git] / gcc / tree-call-cdce.cc: Conditional Dead Call Elimination pass for the GNU compiler. Available at: https://gcc.gnu.org/git/?p=gcc.git;f=gcc/tree-call-cdce.cc;hb=refs/heads/master, accessed 19.02.2024.

[3]. Common Weakness Enumeration (CWE™) – CWE-561: Dead Code. Available at: https://cwe.mitre.org/data/definitions/561.html, accessed 19.02.2024.

[4]. Belevantsev A., Borodin A., Dudina I., Ignatiev V., Izbyshev A., Polyakov S., Zhurikhin D. Design and Development of Svace Static Analyzers. In 2018 Ivannikov Memorial Workshop (IVMEM), 2018, pp. 3- 9.

[5]. Borodin A.E., Belevancev A.A. A static analysis tool Svace as a collection of analyzers with various complexity levels. Proceedings of the Institute for System Programming of the RAS, 2015, vol. 27, issue 6, pp. 111-134.

[6]. Cytron R., Ferrante J., Rosen B. K., Wegman M. N., Zadeck F. K. An efficient method of computing static single assignment form. In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1989, pp. 25-35.

[7]. Galustov A., Borodin A., Belevantsev A. Devirtualization for static analysis with low level intermediate representation. In 2022 Ivannikov Ispras Open Conference (ISPRAS), 2022, pp. 18-23. DOI: 10.1109/ISPRAS57371.2022.10076859.

[8]. Borodin, A.E., Dudina, I.A. Intraprocedural Analysis Based on Symbolic Execution for Bug Detection. Programming and Computer Software, 2021, vol. 47, issue 8, pp. 858-865.

[9]. Mulyukov, R.R., Borodin, A.E. Using unreachable code analysis in static analysis tool for finding defects in source code. Proceedings of the Institute for System Programming of the RAS, 2016, vol. 28, issue 5, pp. 145-158.

[10]. Klocwork Documentation – C and C++ checker reference – SV.RVT.RETVAL_NOTTESTED: Ignored return value. Available at: https://help.klocwork.com/current/en-us/reference/sv.rvt.retval_nottested.htm, accessed 19.02.2024.

[11]. SonarSource: Static Analysis Rules. Available at: https://rules.sonarsource.com, accessed 19.02.2024.

[12]. Splint Manual. 8.4 Suspicious Statements. 8.4.1 Statements with No Effects. 8.4.2 Ignored Return Values. Available at: https://splint.org/manual/manual.html#control, accessed 19.02.2024.

[13]. Coverity Scan – Success Stories: Sample of Defects found and fixed – ScummVM: USELESS_CAL. Available at: https://scan.coverity.com/o/oss_success_stories/86, accessed 19.02.2024.

[14]. GitHub: ScummVM repository – commit 95884c3. Available at: https://github.com/scummvm/scummvm/commit/95884c396b667e048af933292f40fc18da2cefd1, accessed 19.02.2024.

Информация об авторах / Information about authors

Вера Игоревна ВАСИЛЬЕВА учится на факультете вычислительной математики и кибернетики МГУ имени М. В. Ломоносова, одновременно работая в Институте системного программирования РАН. Сфера научных интересов: компиляторные технологии, статический анализ исходного кода программ, обнаружение ошибок программного обеспечения.

Vera Igorevna VASILEVA – student at the Faculty of Computational Mathematics and Cybernetics of Moscow State University (MSU), employee of Institute for System Programming of the RAS. Her research interests: compiler technologies, program source code static analysis, software defects detection.

Александр Ефимович ВОЛКОВ – старший научный сотрудник Института системного программирования РАН. Сфера научных интересов: компиляторные технологии, статический анализ исходного кода программ, обнаружение ошибок программного обеспечения.

Alexander Efimovich VOLKOV – researcher of Institute for System Programming of the RAS since 2007. Research interests: compiler technologies, program source code static analysis, software defects detection.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник Института системного программирования РАН. Сфера научных интересов: компиляторные технологии, статический анализ исходного кода программ, обнаружение ошибок программного обеспечения.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.), researcher of Institute for System Programming of the RAS since 2007. Research interests: compiler technologies, program source code static analysis, software defects detection.