Acta Cybernetica — *online–first paper version* — pages 1–18.

Detecting Uninitialized Variables in C++ with the Clang Static Analyzer*

Kristóf Umann and Zoltán Porkoláb[a]

Abstract

Uninitialized variables have been a source of errors since the beginning of software engineering. Some programming languages (e.g. Java and Python) will automatically zero-initialize such variables, but others, like C and C++, leave their state undefined. While laying aside initialization in C and C++ might be a performance advantage if an initial value cannot be supplied, working with variables is an undefined behaviour, and is a common source of instabilities and crashes. To avoid such errors, whenever meaningful initialization is possible, it should be applied. Tools for detecting these errors run time have existed for decades, but those require the problematic code to be executed. Since in many cases, the number of possible execution paths is combinatoric, static analysis techniques emerged as an alternative to achieve greater code coverage. In this paper, we overview the technique for detecting uninitialized C++ variables using the Clang Static Analyzer, and describe various heuristics to guess whether a specific variable was left in an undefined state intentionally. We implemented and published a prototype tool based on our idea and successfully tested it on large open-source projects. This so-called "checker" has been a part of LLVM/Clang releases since 9.0.0 under the name optin.cplusplus.UninitializedObject.

Keywords: C++, static analysis, uninitialized variables

1 Introduction

When declaring a variable in program code, we might not be able to come up with a meaningful default value thus leaving the variable uninitialized. This is not an issue if one later assigns said variable before reading it, but such errors can be introduced through, for example, code maintenance. Different languages approach this problem in different ways: Java, Python (and many others) zero- initialize variables by

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

[a]Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary. E-mail: `{szelethus, gsd}@inf.elte.hu`. ORCID: `https://orcid.org/{0000-0002-6679-5614, 0000-0001-6819-0224}`.

DOI: 10.14232/actacyb.282900

default, while others, like C and C++, leave their values in an undefined state, and working with such variables leads to undefined behaviour at runtime.

Undefined behaviour in C/C++ is any behaviour the standard does not specify. It may occur, among many other sources, when a null pointer is dereferenced, the division is made by zero, or an array is indexed out of its bounds. The real danger of undefined behaviour is that in some cases, the program might behave seemingly correctly, but other times run into runtime errors, e.g. crashing, corrupting opened files or memory regions allocated by other programs. Also, even the same kind of undefined behaviours might manifest in different runtime errors on different executions.

This makes catching undefined behaviour often very hard – in the case of uninitialized variables, zero initialization might occur with a particular compiler, on a particular platform, in a particular build mode, but might also leave the variable hold whatever value that was stored in the memory region to which the variable was assigned (so-called "garbage value"). However, the use of this behaviour could result in some performance enhancement, tempting programmers to not initialize variables, even though this is often a bad approach.

This paper investigates how such variables can be detected using static analysis. Unlike many other available tools, we will focus on non-idiomatic C++ initialization rather then uninitialized value misuse. In Section 2, we will discuss related initialization rules in C++. In Section 3, we overview runtime and static tools that are available for the same problem and sample several techniques of the latter in Section 4 with a highlight on symbolic execution. We will detail the actual implementation of our prototype in Section 5 along with the heuristics we use to emit only reports that will most likely result in incorrect program behaviour, and what other heuristics could be implemented. We evaluate our prototype solution on various large open-source codebases in Section 6 and discuss future works in Section 7. Our paper concludes in Section 8.

2 C++ Initialization Rules

The way variables are initialized in C/C++ depends on the variables' type. They can be categorized by whether they are records, arrays, or else. For this paper, we will refer to the latter category as *primitive*. The C++ standard specifies two types of initialization that may result in an indeterministic value [13, p. 221][25], but zero initialization is also relevant in this context. When creating an uninitialized object of type `T`

- default initialization occurs
  - if `T` is a class: default constructor is called, or
  - if `T` is an array, each element is default initialized, or
  - otherwise, no initialization occurs resulting in an indeterminate value,

Variable declaration	`i`'s initialization
`T i;`	default initialization
`T i{};`	value initialization (C++11)
`T i = T();`	value initialization
`T i = T{};`	value initialization (C++11)
`T i();`	function declaration (no variable initialization occurs)

Figure 1: Initialization rules for instatiation of local variables

`A::A()`'s definition	`A::t`'s initialization
`struct A { T t; A() : t() {} };`	value initialization
`struct A { T t; A() : t{} {} };`	value initialization (C++11)
`struct A { T t; A() {}; };`	default initialization
`struct A { T t; A() = default; };`	value initialization

Figure 2: Initialization rules for data member `A::t` upon instantiating a local `A` typed variable with the default constructor.

- value initialization occurs
  - if T is a class, the object is default initialized after zero initialization if T's default constructor is not user-defined/deleted, or
  - if `T` is an array, each element is value initialized, or
  - otherwise the object is zero initialized,
- zero-initialization occurs, before any other initialization
  - if the object is static or thread-local, or
  - if `T` scalar (number, pointer, enum) set to 0, or
  - if `T` is a class, all subobjects are zero-initialized.

We summarize these rules in Figure 1. and 2. While we did not discuss initialization in great depth in this section, it shows that this issue is not only context-sensitive, it is also confusing, since it can be intertwined with other C++ rules, such as those related to inheritance. Even telling when the compiler will generate a constructor can be difficult [30].

3 Related work

When approaching the issue of uninitialized variables, we can sort the already existing solutions into two categories: runtime and static. An ideal solution is both *correct* and *complete*, where none of our reports is incorrect (*false positive*), and we

identify all uninitialized objects. Different techniques tend to emphasize different parts of the requirements mentioned above – Runtime techniques are inherently less prone to report false positives, but lack completeness [14]. Static analysis tools cover far more of the code and offer a more complete but less precise solution [18].

In the context of this paper, we are looking for a solution to detect error-prone lack of initialization and not uninitialized value misuses. Despite this, both due to the scarcity of tools that focus on the former rather than the latter, and the strong relevance of the two problems, we feel it is important to take a short survey of analyzers and techniques that implement rules for either.

3.1 Runtime analysis tools

Several papers survey the detection of uninitialized value misuses in runtime analyzers [14, 20, 6]. A common characteristic of said tools is to inspect the program during execution, with a given input. This means that in any given analysis, these tools will only observe a single path of execution. However, this simplification makes these tools far more precise, improving, on that particular path of execution, both the true positive and the true negative findings.

Valgrind is a general dynamic binary instrumentation (DBI) framework [23]. It inspects the executable binary, rather than the source code of a program, which might not even be available. Valgrind offers several tools (also referred to as *plugins*) that can find bugs. It is implemented by its core disassembling a given code block from the binary into an intermediate representation, which is instrumented with analysis code by the plugin, and then converted back into machine code. The resulting *translation* is stored in a code cache to be rerun as necessary. One of these plugins, MemCheck [27], can find uninitialized value misuses using shadow bits [22] for every byte in the application memory: one a bit indicates whether it is addressable, and a bit indicates whether it has a defined value.

Dr. Memory [4] works similarly to MemCheck but is more modern, better optimized and multi-threaded. Its two times faster then MemCheck, but due to concurrent updates of adjacent shadow bits, is more prone to emit false positive and false negative reports [28].

MemorySanitizer [28] offers a different approach to runtime analysis by only solving the problem of uninitialized variable misuses. It generates a modified binary during compilation, skipping disassembly and reassembly entirely. Running the generated binary is far faster and consumes less memory then the solution Valgrind with MemCheck or Dr. Memory offers, but this requires the source code to be available.

3.2 Static analysis tools

Static analyzers do not execute the program under analysis, but rather inspect either the source code or the generated binary code. This results in far greater code coverage as they are not restricted to a single path of execution. However, this generality comes at the cost of the analyzer tool having little knowledge about input

values. In parts of the code where such information is crucial, the given analyzer might have to conservatively suppress its reports, or simulate the execution of parts of the code, making assumptions on values. This, considering the complexity of some of the C++ language features (as discussed in Section 2.) can result in a higher number of false positives and false negatives. We will discuss some of the more popular techniques in Section 4. For the remainder of this section, we will sample some of the widely used C/C++ static analyzers.

CppCheck [21] is among the earliest open-source tools with support for C++. It uses AST matching and dataflow analysis[1] to find bugs and code smells. Contrary to many open-source tools for C++ analysis, CppCheck implements its own preprocessor, parser and abstract syntax tree (*AST*). It defines two rules on incorrect initialization in constructors, separately for private and non-private fields.

Infer [5] is a relatively new tool, focusing on scalability and fast execution. It has a unique approach to static analysis, using bi-abduction to perform interprocedural analysis. Infer also runs with cross translation unit analysis enabled by default, and scales significantly better with the number of translation units to analyze compared to other tools such as the Clang Static Analyzer [10]. While it has several checkers to detect uninitialized value misuse, it does not have any that focuses on non-idiomatic C++ object initialization.

The Clang Static Analyzer [18], similarly to CppCheck, is among the more mature static analyzers for C++. Having the benefit of being implemented directly in the Clang compiler and transitively LLVM itself, it can take advantage of several well-tested algorithms and data structures. The Clang Static Analyzer (or *analyzer* for short) was ultimately our choice of project to implement our prototype in and will be discussed in greater detail in Section 5.1.

There are also several commercial static analyzers such as CodeSodar [8], Coverity [29], Klocwork [15], but due to licencing issues we will not compare our results to them.

4 An introduction to symbolic execution

Several static analysis techniques may be considered for finding uninitialized variables, each having different strengths and weaknesses in terms of analysis speed, memory or persistent storage consumption. In this section, we introduce symbolic execution through two other approaches, and demonstrate why it is more appropriate for our purpose.

4.1 Text-based matching

A possible, though a primitive approach would be to use textual pattern matchers. Let us see through a couple of examples whether we can tackle the problem initialization with regular expressions:

[1]The authors of CppCheck refer to this technique as “valueflow”, rather then dataflow.

```
int i;
```

With the regular expression rule `int [A-Za-z]+[A-Za-z0-9]*;` we can catch this error. We can even enhance this regular expression by handling other fundamental types, ignoring whitespaces, C-style comments and the like. By inspecting the preprocessed code, rather than the original source code, we can also handle cases where the preprocessor would generate parts of the expression. However, in Figure 3, we demonstrate that non-trivial cases require a context-sensitive grammar. On that code snippet, `a.i` will be initialized by the of the constructor call, but `a.j` will not be. Having multiple constructors, potentially *after* instantiating the class, inheritance, virtual inheritance, constructor delegation, aggregate initialization make solving even smaller parts of this problem practically impossible with text-based pattern matching.

```
1  struct A {
2    int i;
3    int j;
4    A() : i(0) {}
5  };
6
7  A a;
```

Figure 3: Text-based pattern matching is unable to identify `a.j` as uninitialized.

4.2 AST matching

A more sophisticated approach is to utilize the *abstract syntax tree* (AST), which provides far more C++ specific information, especially when coupled with semantic information, such as type information and an identifier table.

For the code snippet on Figure 4a, according to the rules detailed in Section 2, `b1`, `b2` are value initialized while `b3` is default initialized, leaving `b3.data` in an indeterministic state. As discussed earlier, we cannot tell this with regular expressions.

On Figure 4c, we can inspect how Clang constructs the AST for Figure 4a. Using Clang's AST matcher library, we can match the line on which `b3` is defined with the following matcher:

```
declStmt(unless(hasDescendant(
    stmt(anyOf(cxxConstructExpr(requiresZeroInitialization()),
               implicitValueInitExpr())))));
```

This approach is clearly superior to text-based matching because it is context-sensitive and allows us to express C++ specific properties easily. AST matching can be complemented with the retrieval of the matched expression, enabling us to do additional compile-time analysis, such as inspecting the inheritance tree of a

```
1  struct RealSqrt {
2    int data;
3
4
5
6
7  };
8
9  int main() {
10   RealSqrt b1 = RealSqrt();
11   RealSqrt b2{};
12   RealSqrt b3;
13 }
```

(a)

```
1  struct RealSqrt {
2    int data;
3    RealSqrt(int i) {
4      if (i >= 0)
5        data = i;
6    }
7  };
8
9  int main() {
10   int i = rand();
11   if (i >= 0)
12     RealSqrt b(i);
13 }
```

(b)

```
CXXRecordDecl line:1:8 referenced struct A definition
|-DefinitionData
| |-DefaultConstructor trivial
| |-CopyConstructor trivial
| |-MoveConstructor trivial
| |-CopyAssignment trivial
| |-MoveAssignment trivial
| `-Destructor irrelevant trivial
|-FieldDecl col:7 referenced a 'int'
FunctionDecl line:5:5 main 'int ()'
`-CompoundStmt
  |-DeclStmt
  | `-VarDecl col:5 b1 'A' cinit
  |   `-ExprWithCleanups 'A'
  |     `-CXXConstructExpr 'A' 'void (A &&) noexcept'
  |       `-MaterializeTemporaryExpr 'A' xvalue
  |         `-CXXTemporaryObjectExpr 'A' 'void () noexcept' zeroing
  |-DeclStmt
  | `-VarDecl col:5 b2 'A' listinit
  |   `-InitListExpr 'A'
  |     `-ImplicitValueInitExpr 'int'
  |-DeclStmt
  | `-VarDecl col:5 b3 'A' callinit
  |   `-CXXConstructExpr 'A' 'void () noexcept'
```

(c) Simplified AST generated by Clang for Figure Figure 4a.

Figure 4: Code snippets demonstrating the internal representation and analysis techniques used by Clang.

type, gathering all direct (inherited and in-class) fields and checking whether the called constructor is compiler-generated.

While the techniques mentioned above can be used to reduce the number of false positives drastically, not even AST based matching can effectively deal problems requiring path sensitive information.

It is clear that on Figure 4b, `b.data` will not be uninitialized, but we cannot reliably detect that with AST matchers. Even if we do additional compile-time analysis, we cannot reason about runtime values, branches, loops and the like without path sensitive analysis. However, AST-based pattern matching is relatively fast and can be used as a supplement to a better approach.

4.3 Symbolic Execution

Symbolic execution [9] is a powerful static analysis technique: it essentially simulates the execution of the program. The implementing tool follows the control flow graph (*CFG*), and evaluates statements within a basic block. Each expression is represented with a symbolic expression that is assigned a value (e.g. assigning `i + 1` with the value `11`), which itself could be symbolic if the value is unknown (e.g. supplied from a file or the command line, randomly generated or received from a translation unit we cannot analyze).

Upon encountering branch statements (when a basic block has more than one outgoing edge), the tool will use a constraint solver to evaluate the condition [16]. If the condition can be proven to be false or true in the given program state, the tool can ignore all but one of the outgoing edges. Otherwise, the analyzer will explore both paths, one on which the condition of it is true, and one where it is false. This introduces the possibility to impose constraints upon symbolic values, such as a pointer check could tell that the pointer value is non-null on a path, even if the precise value is still unknown, and null on the other.

With this technique, we can theoretically explore all execution paths and reason about the values of variables. For Figure 4b, we will note that `A::A()` will only be called if the supplied parameter has a value of 5, and there is no uninitialized value problem made in the program.

Since we can inspect the values in any given program state, handling complex C++ code, such as inheritances, virtual inheritances, constructor delegations come naturally: we could inspect an object after the end of a constructor call.

Clearly, symbolic execution is a far more powerful tool than text- or AST- based pattern matching, hence our decision to use it in our implementation. It is worth noting though that such an analysis is several times slower than compilation.

5 Implementation

In this section, we will detail how we implemented our prototype in the Clang Static Analyzer, and some of the heuristics used to guess whether the discovered uninitialized objects were left as such intentionally.

5.1 Clang Static Analyzer

Clang is the C/C++ frontend of the LLVM optimizer and code generator and houses the Clang Static Analyzer (*analyzer*). The Clang Static Analyzer implements all three analyses detailed in Section 4. In this project, symbolic execution is divided into two main components: a core that explores paths of execution and evaluates statements, and so-called *checkers* that define bugs or code smells. Our prototype is also a *checker*.

During symbolic execution, the core will simultaneously build an *exploded graph* [12]. This graph describes the entire analysis – nodes could represent new information the analyzer learned while evaluating a statement, such as a new constraint on a symbolic expression, the end of a code block, call to a function, end of a function call, or even checker-specific information (e.g. our prototype creates a new node to note already reported fields). Control statements like `if` and `while` could correlate to several exploded nodes, depending on how many times the condition was evaluated. We call the event when the analyzer constructs multiple child nodes *state splitting*, and it may be introduced by checkers as well. For instance, a dynamic memory modelling checker may split the state when modelling a call to the `malloc` function, creating a path of execution where the result of the function call is `NULL`, and one where the allocation was successful. This implies that the exploded graph grows exponentially, hence the naming.

The exploded graph is not isomorphic with the CFG, but it does have a natural projection to it: each exploded node can be mapped to a statement or an edge in the CFG.

The analyzer categorizes many modelling steps into events. Checkers can subscribe to one or several of these, and are notified by the core should they occur. InnerPointerChecker [17], for instance, is subscribed to the "end of a function call" event to mark the return values of method calls like `std::string::c_str()` as pointers to internal data, and to the "dead symbols" event to mark such data as released after going out of scope.

5.2 Representing an object

In C++, the proper initialization of objects of a record type is the responsibility of the special member function called the constructor. The constructor could be user-provided when the programmer specifies it or compiler provided if it is automatically generated. The improper implementation of the user-provided constructor is the primary source of uninitialized value misuse. Therefore, in this research, we are concentrating on validating the construction.

Naturally, any representation must respect fields and fields of fields (*subobjects*), which we will call *direct containment*. Inherited fields are also directly contained. However, non-null valid pointer objects refer to objects that may be uninitialized themselves. We will refer to pointees of pointer fields as *indirectly contained* objects.

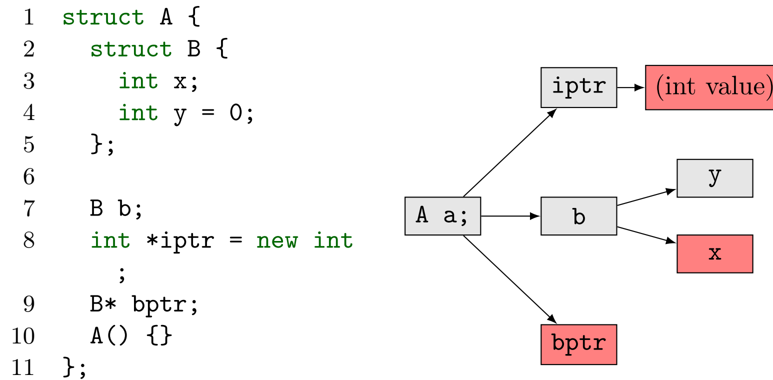

Figure 5: How an object is represented by the checker. Red nodes are uninitialized objects.

With that in mind, our proposed representation is a directed graph, where

- The root of this graph is the object we are analyzing,
- Every other node is also an object that is a union, a non-union record, dereferenceable, an array or of a primitive type,
- The parent of each node is the object that contains it.

It is easy to show that this representation is not always a directed tree: circular linked lists and pointers pointing to themselves are all directed cycles. However, by keeping track of the objects we already analyzed, we can disallow these con- structs, turning this representation into a directed tree.

We can also realize that in this directed tree every leaf is either a null pointer, an undefined pointer, an array, a primitive object, or a pointer that that points to an already analyzed object, meaning that they can be represented as numerical values. This is important, as the analyzer may only mark such objects as uninitialized.

By subscribing our checker to the *end of a function call* event, we can inspect every object after the end of a constructor call. By traversing the above graph, we can detect all directly and indirectly contained objects, and emit reports for each of them.

Our prototype traverses this graph recursively in a depth-first manner, and after each descent to a child node, constructs an informative object to keep track of the path leading to the current node. By the time it reaches a leaf, it has constructed an informative list that contains every information needed to reach it. Should that leaf be in an indeterministic state, a report is emitted, and a helpful warning message is constructed from the informative list. For Figure 5, our prototype would report that `this->b.x`, `this->iptr`'s pointee and `this->bptr` is left uninitialized after the constructor call. However, since the analyzer keeps track of the dynamic type of

the pointers during symbolic execution, the warning message could be incorrect, if we report a derived class' field as uninitialized through its base class' typed pointer. To solve this problem, the checker stores dynamic type information as well in the constructed informative list, so the constructed warning message could contain casts the actual dynamic type.

As constructors are called for fields during construction, we will only run our checker when function call stack does not contain other constructor calls.

5.3 Heuristics

In static analysis, we can classify results as true positive, if the analyzer correctly identified a programming error, true negative if the analyzer correctly identified the lack of a programming error, false positives if the analyzer reported a programming error despite the code being correct, and false negatives when the analyzer did not report incorrect code. These definitions, however, describe uninitialized object related reports rather poorly. While correctly identified and reported uninitialized objects are by definition true positives, these objects may have been left as such intentionally. An essential aspect of this problem is that not initializing a variable is not an error, only the reading of an uninitialized value. In fact, when the programmer cannot supply a meaningful default value (e.g. declaring a variable as a buffer), initialization could result in a performance loss.

For this reason, one of the main goals of our research is to identify which uninitialized variables are most likely to result in misuse and undefined behaviour. This implies that we have to reason about the intention of the programmer and suppress some reports, turning them into false negatives. The following section will detail how we try to find an optimal true positive/false negative ratio.

We made our checker configurable, allowing us to enable, disable or fine-tune some of our heuristics for a particular project.

5.3.1 Arrays

Before C++11, elements of dynamically allocated arrays could not be initialized. Even stack-allocated arrays are often used as buffers, which was consistent with the results of our findings, so our prototype ignores arrays.

5.3.2 No initialized field

Through testing our prototype, we concluded that objects that do not initialize a single one of their fields are often created intentionally. However, this heuristic can result in a higher amount of false negatives than maybe desired, so we made it toggleable.

5.3.3 Pointer chasing

Indirect containment raises a philosophical question: Is an object responsible for leaving its pointee object in a fully deterministic state? One perspective we could

```
1  struct PhysicalProperty {
2    int volume, area;
3    enum Kind { VOL, AREA } kind;
4
5    PhysicalProperty(Kind k) : kind(k)
       {
6      switch(k) {
7      case VOL:
8        volume = 0;
9        break;
10     case AREA:
11       area = 0;
12       break;
13     }
14   }
15
16   int getVolume() const {
17     assert(kind == VOL);
18     return volume;
19   }
20
21   int getArea() const {
22     assert(kind == AREA);
23     return area;
24   }
25 };
```

Figure 6: `PhysicalProperty` doesn't initialize all data members, but „guards" against uninitialized value misuse.

take is to guess whether the objects *owns* the pointee. However, ownership is a conceptually popular, but non-standardized concept within C++ [11]. This, and the current faults in the analyzer lead to us to not analyze pointees (or *chasing pointers*) by default.

5.3.4 Guarded field analysis

Consider the code snippet in Figure 6. Although `PhysicalProperty` will leave one of its fields uninitialized on every instantiation, we cannot encounter an uninitialized object related error runtime. Similar constructs within the LLVM codebase is very popular and will trigger a report from our checker, despite the lack of a programmer error.

We will call any statement that could prevent the execution from reaching and reading an uninitialized field a *guard*. We call a field *guarded* if every read of it control depends on a guard.

Unfortunately, it is hard to guess compile-time whether a field is guarded, as the argument of if statements might not be correlated to whether the field is initialized. We implemented a primitive heuristic to solve this problem using AST matchers on the object's record definition, analyzing whether the field is public, and is read before a guard in the code. Due to the reasons mentioned above, this is a very rough estimate, and this analysis is disabled by default.

5.3.5 Known to be safely uninitialized fields

In some instances, we might want to ignore particular objects of a particular type intentionally, or if they have a specific variable name. For this reason, our prototype is configurable with a regular expression, and if it matches a variable's name or it's type, we ignore it.

6 Evaluation

We evaluated our prototype on several large, open-source C++ projects, such as Rtags [3], LLVM [19], Xerces [1], CppCheck [21], Bitcoin [31] with a variety of configurations. We used the open-source program csa-testbench [24], which helped us compare the results of different configurations of our checker with ease. We used CodeChecker [7] to visualize our results. We also ran CppCheck on said projects and compared its results with the one our solution generated.

We purposely chose projects with diverse design patterns. For example, the LLVM project relies almost purely on its own libraries, and being a compiler, it is very performance-critical. The C++ indexer Rtags uses several third-party libraries and houses a variety of coding styles, some more performance-critical then others. We found that the more performance-critical a project is, the more likely it is that the code takes advantage of not initializing every variable.

As mentioned in Section 5.3, it can be challenging to find good metrics on the quality of the reports. While the authors were researching an algorithm which enforces the idiom of initializing every variable, there are several reports (especially in the code generation libraries of LLVM) that we feel justifies going against it. This implies that judging whether a report is meaningful or not is debatable. For this reason, we categorized the results into three categories: we say a report *useful* if the lack of initialization had probably little to no effect on performance and is error-prone, *questionable* if judging from the code context the lack of initialization is appropriate, and *false positive* if the report was incorrect. We summarized our results in Table 1.

According to the C++ initialization rules, we found only a single false positive where the analyzer disregarded an in-class initialization, though this is a fault of the analyzer's core. We found that the reports from our prototype with the

Table 1: Reports from our prototype in the first 4 columns, and from CppCheck in the last on large, open source projects. Reports are shown in the format "all/**useful**/false positive/questionable".

	Default	Pointer chasing	Pedantic	Guarded fields ignored	CppCheck
Rtags	1/**1**/0/0	6/**1**/0/5	1/**1**/0/0	1/**1**/0/0	5/**1**/0/4
LLVM	46/**18**/1/27	88/**18**/1/69	48/**20**/1/27	43/**18**/1/24	4/**1**/0/3
Xerces	1/**1**/0/0	1/**1**/0/0	1/**1**/0/0	1/**1**/0/0	7/**4**/0/3
CppCheck	0/**0**/0/0	0/**0**/0/0	0/**0**/0/0	0/**0**/0/0	0/**0**/0/0
Bitcoin	5/**5**/0/0	5/**5**/0/0	5/**5**/0/0	5/**5**/0/0	12/**5**/0/7

default configuration were overwhelmingly useful except for LLVM. Despite most uninitialized variable finds in that project were of little value, even there we were able to find and patch error-prone code.

We found code pattens in nearly all projects that clearly go against idiomatic C++ code. Such code patterns include not letting the compiler generate constructors by defaulting them with `= default`, not initializing variables that are removed in non-debug builds, leaving uninitialized fields of non-POD classes public, or storing auxiliary data such as counters that would be more appropriate as function-local variables.

Contrary to our expectations, both reports that are present with the Pedantic option enabled but not with default configurations were useful, showing constructors that should have been defaulted. Though the number of reports increased significantly after enabling pointer chasing, it mostly lead us to find pointers to buffers that were handled correctly in the class, making them uninteresting in all reports found in LLVM and Rtags. LLVM uses guarded fields extensively, but we were only able to suppress reports based on this information in 3 cases.

Comparatively, CppCheck had the least amount of reports on LLVM except for its own codebase, significantly less than our prototype both in terms of count and useful finds, but had more on Rtags, Xerces and Bitcoin. Shockingly, there was only a single report found by both CppCheck and out prototype in Xerces. This supports the conclusion of other findings on static analyses that to find more bugs, it is better to use multiple tools [2, 33].

It should be noted that unlike our prototype, CppCheck constructs a warning message per uninitialized field, rather than per constructor call, so we regarded multiple warnings originating from the same constructor as one.

This checker, under the name of `optin.cplusplus.UninitializedObject` has been a part of the analyzer since it's 9.0.0 release [32], and has been used by various industrial parties, such as Firefox[2], Apple[3], Google[4] and Ericsson[5].

[2] `https://reviews.llvm.org/D45532\#1145512`
[3] `http://lists.llvm.org/pipermail/cfe-dev/2018-August/058905.html`
[4] `https://reviews.llvm.org/D58573\#1477581`
[5] `https://reviews.llvm.org/D58573\#1425837`

7 Future works

It is hard to find an ideal true positive/false negative ratio to make our reports meaningful enough without suppressing too many of them. More research into new heuristics and improving existing ones is where most of our future projects lay.

Guarded field analysis could benefit from being implemented using dataflow analysis instead of AST matching. While finding a correlation in between the guard statement and whether the field is initialized is theoretically impossible, clever heuristics could help on this matter.

Pointer chasing suffers from some poor modelling techniques within the analyzer's core, which is not directly related to our implementation. Also, heap allocated objects are not yet modelled at all. Improving these within the core and better defining in which cases we want to report uninitialized pointees could eventually enable us to enable pointer chasing by default.

While there are no constructors in C, it is worth investigating whether our C++ prototype could be used for analyzing C code. One approach would be to note when an object of a great enough size is created, and when the function call in which its created ends, analyze that object with our proposed technique.

A popular technique in C++ programming is the *pimpl idiom* [26], where the part of the class' definition is implemented through an opaque pointer. The Clang Static Analyzer, by default, can only analyze a single translation unit at a time, so it may be unable to reason about opaque pointers if the definition of a function or a class lies a translation unit different than what is being analyzed. Cross translation unit (*CTU*) analysis [10] can be used to acquire the definition. It should be investigated whether our prototype is conformant with CTU.

8 Conclusion

Static analysis of C/C++ code can be used to detect uninitialized variables, which are a common source of undefined behaviour. While more prone to false positives, static analysis has a far greater code coverage compared to dynamic analysis. We argued against analyzing only fundamental objects and proposed an accurate representation of record objects in the form of a directed tree. Our prototype, implemented in the Clang Static Analyzer, can traverse this graph to detect uninitialized variables for each object after the end of its constructor call. We proposed a variety of heuristics to reduce the number of reports emitted by this prototype, focusing on uninitialized variables most likely to be read. Evaluation of large open-source projects lead us to discover several records that are likely to leave some fields uninitialized unintentionally and are prone to misuse. While we generally found the results of our prototype meaningful, we plan to add new heuristics and enhance existing ones to reduce the further number of uninteresting reports on performance-critical projects.

This checker, under the name of `optin.cplusplus.UninitializedObject` has been a part of the analyzer since it's 9.0.0 release [32], and has been used by various industrial parties, such as Firefox, Apple, Google and Ericsson.

References

[1] Apache Software Foundation. Apache Xerces. `https://xerces.apache.org/`.

[2] Arusoaie, Andrei, Ciobâca, Stefan, Craciun, Vlad, Gavrilut, Dragos, and Lucanu, Dorel. A comparison of open-source static analysis tools for vulnerability detection in C/C++ code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 161–168. IEEE, 2017. DOI: `10.1109/SYNASC.2017.00035`.

[3] Bakken, Anders. Rtags. `http://www.rtags.net`.

[4] Bruening, Derek and Zhao, Qin. Practical memory checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011. DOI: `10.1109/CGO.2011.5764689`.

[5] Calcagno, Cristiano, Distefano, Dino, Dubreil, Jérémy, Gabi, Dominik, Hooimeijer, Pieter, Luca, Martino, O'Hearn, Peter, Papakonstantinou, Irene, Purbrick, Jim, and Rodriguez, Dulma. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015. DOI: `10.4204/eptcs.188.2`.

[6] Cifuentes, Cristina, Hoermann, Christian, Keynes, Nathan, Li, Lian, Long, Simon, Mealy, Erica, Mounteney, Michael, and Scholz, Bernhard. Begbunch: Benchmarking for c bug detection tools. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 16–20, 2009. DOI: `10.1145/1555860.1555866`.

[7] Ericsson. CodeChecker. `https://github.com/Ericsson/codechecker`.

[8] GrammaTech. Codesonar. `https://www.grammatech.com/products/codesonar`.

[9] Hampapuram, Hari, Yang, Yue, and Das, Manuvir. Symbolic path simulation in path-sensitive dataflow analysis. *ACM SIGSOFT Software Engineering Notes*, 31(1):52–58, 2005. DOI: `10.1145/1108768.1108808`.

[10] Horváth, Gábor, Gera, Zoltán, Krupp, Dániel, Porkoláb, Zoltán, and Szécsi, Péter. Translational unit analysis in clang static analyzer: Prototype and measurements. European LLVM Developers Meeting, Saarbrücken, 2017.

[11] Horváth, Gábor and Pataki, Norbert. Categorization of C++ classes for static lifetime analysis. In Eleftherakis, George, Lazarova, Milena, Aleksieva-Petrova, Adelina, and Tasheva, Antoniya, editors, *Proceedings of the 9th Balkan Conference on Informatics, BCI 2019, Sofia, Bulgaria, September 26-28, 2019*, pages 21:1–21:7. ACM, 2019. DOI: `10.1145/3351556.3351559`.

[12] Horváth, Gábor, Szécsi, Péter, Gera, Zoltán, Krupp, Dániel, and Pataki, Norbert. Implementation and evaluation of cross translation unit symbolic execution for c family languages. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 428–428. ACM, 2018. DOI: `10.1145/3183440.3195041`.

[13] Programming languages – C++. Standard, International Organization for Standardization, Geneva, Switzerland, December 2017.

[14] Jana, Anushri and Naik, Ravindra. Precise detection of uninitialized variables using dynamic analysis-extending to aggregate and vector types. In *2012 19th Working Conference on Reverse Engineering*, pages 197–201. IEEE, 2012. DOI: `10.1109/WCRE.2012.29`.

[15] Klocwork. Klocwork. `https://www.roguewave.com/products-services/klocwork`.

[16] Kovács, Réka and Horváth, Gábor. An initial prototype of tiered constraint solving in the clang static analyzer. *Studia Universitatis Babes-Bolyai, Informatica*, 63(2), 2018. DOI: `10.24193/subbi.2018.2.06`.

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

[18] Kremenek, Ted. Finding software bugs with the Clang static analyzer. LLVM Developers' Meeting.

[19] Lattner, Chris and Adve, Vikram. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004. DOI: `10.1109/CGO.2004.1281665`.

[20] Lu, Shan, Li, Zhenmin, Qin, Feng, Tan, Lin, Zhou, Pin, and Zhou, Yuanyuan. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

[21] Marjamäki, Daniel. Cppcheck: a tool for static C/C++ code analysis, 2013.

[22] Márton, Gábor and Porkoláb, Zoltán. Compile-time function call interception for testing in C/C++. *Studia Universitatis Babes-Bolyai, Informatica*, 63(1), 2018. DOI: `10.24193/subbi.2018.1.02`.

[23] Nethercote, Nicholas and Seward, Julian. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007. DOI: `10.1145/1250734.1250746`.

[24] Nikolett Kovács, Réka, Horváth, Gábor, and Szécsi, Péter. Towards proper differential analysis of static analysis engine changes. In *The 11th Conference of PhD Students in Computer Science*, 2018.

[25] Porkoláb, Zoltán. Multiparadigm programming: Constructors, destructors, operators, 2019. `http://gsd.web.elte.hu/lectures/multi/slides/constructor.pdf`.

[26] Reddy, Martin. *API Design for C++*. Elsevier, 2011. DOI: `10.1016/c2010-0-65832-9`.

[27] Seward, Julian and Nethercote, Nicholas. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[28] Stepanov, Evgeniy and Serebryany, Konstantin. Memorysanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 46–55. IEEE Computer Society, 2015. DOI: `10.1109/CGO.2015.7054186`.

[29] Synopsys, Inc. Coverity. `https://scan.coverity.com/`.

[30] Szalay, Richárd and Porkoláb, Zoltán. Visualising compiler-generated special member functions of C++ types. In *Proceedings of Collaboration, Software and Services in Information Society*, pages 55–58, 2018.

[31] The Bitcoin Core. Bitcoin Core. `https://bitcoincore.org/`.

[32] The LLVM Foundation. Clang 9.0.0 release notes, 2019. `https://releases.llvm.org/9.0.0/tools/clang/docs/ReleaseNotes.html#static-analyzer`.

[33] Zitser, Misha, Lippmann, Richard, and Leek, Tim. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106, 2004. DOI: `10.1145/1029894.1029911`.