# IMPROVING THE PRECISION OF FLOW-SENSITIVE LIFETIME ANALYSIS

Gábor HORVÁTH, Norbert PATAKI
Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
E-mail: xazax@caesar.elte.hu, patakino@elte.hu

**ABSTRACT**

*Object lifetimes are a common source of bugs in C++ that can cause crashes, unexpected behavior, or even security vulnerabilities. Herb Sutter, the chair of the C++ standard committee proposed a flow-sensitive analysis to catch lifetime errors statically. Sadly, this analysis is prone to false positives unless the author follows some specific guidelines. We developed mitigations to eliminate some classes of false positives to make it easier to write conforming code. The first mitigation fixes a common false positive from a frequently used coding pattern by introducing local path-sensitivity. The second one is a filter based on reaching definitions and dominance algorithms to remove reports that might be the result of analyzing infeasible paths. We tested the effectiveness of the methods on the open source Google Fuchsia project.*

***Keywords:*** *C++, lifetime analysis, Clang*

## 1. INTRODUCTION

Object lifetime, in general, is the time between the construction of an object and the destruction of the same object. The object lifetime rules can vary greatly between the languages. C++ is infamous for lifetime problems as the language gives a lot of control and responsibility to the users. These lifetime problems can cause unexpected results or even security vulnerabilities. Lifetime errors include use after free, double free, and memory leak errors [24].

Microsoft reports that 70% of the security vulnerabilities they fix are memory errors [17]. Interestingly, Google shows a very similar figure about Chromium [30]. Their report states that these vulnerabilities are evenly spread across the code, making it harder to find them. Non-security stability problems have similar root causes. Chromium tried to use sandboxing to mitigate potential errors, but it also has its limitations. Processes are not cheap, and they still share some memory for efficient communication. Mozilla paints a very similar picture of the distribution of the error types [12]. They evaluated how many vulnerabilities could have been prevented using a safer solution like the Rust programming language [3]. According to their findings, it is worth exploring how to write safer C++.

Due to the vast number of lifetime-related issues in software, it is no surprise the C++ Standards Committee is looking for a solution. Herb Sutter, the chairman of the committee, suggested a flow-based analysis to catch lifetime-related errors [27, 28]. His analysis is part of the C++ Core Guidelines [26]. We implemented the analysis based on his proposal and extended it with mitigations to overcome some of the false positives from the analysis.

A typical lifetime error can be seen below in Listing 1.

```
int f() {
  int *p = 0;
  {
    int x = 5;
    p = &x;
  } // Lifetime of 'x' ends here.
  return *p;
}
```

Listing 1: The dereferenced pointer points to already released memory.

In order to catch lifetime errors, it is crucial to know where pointers might point. Without any knowledge about the pointee of `p` we have no chance to find the problem in the code above. Pointer analysis [23] and lifetime analysis are related problems. Unfortunately, as every non-trivial static analysis problem, they are undecidable in the general case. Note that having a sufficiently sophisticated points-to analysis can help us solving the lifetime problems as we can check if all the pointees are alive at the point of a dereference. Pointers that no longer point to valid memory are called *dangling pointers*. Dereferencing a dangling pointer is a memory error.

```
int g() {
  int *p = 0;
  {
    unique_ptr<int> x = make_unique<int>(5);
    p = x.get();
  } // Lifetime of 'x' ends here.
  return *p;
}
```

Listing 2: The dereferenced pointer points to already released memory.

In Listing 2, we have a slightly modified version of the previous code from Listing 1. We are creating a `unique_ptr` that will point to a heap-allocated memory. The `unique_ptr` owns this memory: when its lifetime ends, the heap-allocated memory will be freed. In order to be able to detect the lifetime error in this code snippet, we need to know this ownership relationship. Unfortunately, C++ has no language rules to determine which objects own which memory. The user is free to implement any ownership model making static analysis harder. We cannot observe an allocation directly in Listing 2, but we can assume one due to the presence of the `unique_ptr`, which is considered an Owner.

This paper is organized as follows. Section 2 summarizes the flow-sensitive lifetime analysis that was championed by Herb Sutter. While we contributed significant amount of feedback, this analysis is not the main result of this paper. As that analysis is defined it was not attractive for the industry enough to gain adoption. We suspect that the lack of adoption is the result of the number of false positive findings emitted by the analysis. Next, Section 3 introduces two methods to mitigate some classes of false

positives. These mitigations are not part of the original proposal yet and constitute our contributions described in this paper. Section 4 discusses alternative approaches to find object lifetime errors. Section 5 summarizes our approach how to proceed improving the analysis. Finally, Section 6 concludes the paper recapping our contributions.

## 2. LIFETIME ANALYSIS

In this section, we introduce the flow-sensitive analysis [27] we built upon that helps finding lifetime errors. This analysis is very similar to a points-to analysis and is based on the lifetime annotations and type categories we discussed earlier. This analysis is an abstract interpretation method [6]. We implement the analysis in the Clang compiler [1] that is based on the LLVM compiler infrastructure [15].

### 2.1. Type categories

Herb Sutter defined four type categories in his analysis. To automatically categorize a type, we inspect its public interface. The public interface of a class written in modern C++ carries rich information about the class' behavior. In fact, we would prefer to utilize the *concept* feature of C++20, but it has not been widespread at the time of writing this paper [25]. We mainly use these categories to reason about the ownership relationships between objects. Misclassifications can result in false positive or false negative findings, but the alternative to automatic classification would be to require the user to annotate every class. That requirement would hinder the adoption of the analysis.

There are a few main observations behind the idea of the four type categories. *Owner*s own the memory they refer to. They will do all the necessary allocations, deallocations, and copies. Thus, properly implemented Owners will never dangle. Whenever an Owner goes out of scope, the owned memory will be released. Note that none of the proposed analyses tries to validate the safety of the owners' implementation. Other state of the art solutions like Rust also tend to use unchecked implementations for owners. Modeling the implementation of owners requires a sophisticated analysis of the heap, which an open problem with a large number of potential solutions [13, 20], none of which are sufficient for our use case. *Pointer*s do not own the memory they refer to. In order to distinguish between the type category Pointer and the primitive pointer type, we will start the former with a capital letter. Raw pointers, references, and some user-defined types like iterators belong to this category. *Aggregate*s are plain old data types that are handled member-wise. *Value*s are the types that did not fit into any of the former categories. For more details on categories see our paper about a statement-local analysis variant [11].

### 2.2. Annotation language

To model function calls, the lifetime analysis defines an annotation language to describe lifetime related contracts. These contracts are summaries. Note that the user does not need to specify the lifetime contracts for each functions as we have inference rules that can infer a sensible summary for most functions based on their declarations.

The annotations describe some aspects of the function and they are independent of the actual analysis. There are multiple analyses capable of consuming these annotations. Our statement-local analysis [11] based on this work was able to find bugs in many open source projects including Chromium, LLVM, and OpenCV.

We use `gsl::pre`[1] and `gsl::post` annotations to describe lifetime contracts. Both annotations expect an expression of form `lifetime(source, {targets})`. This makes it clear that these contracts describe lifetime requirements. This contract describes that the pointee of `source` has the same lifetime as the smallest lifetime among the pointees in the `targets` set. The EBNF grammar of the annotation language is shown in Fig. 1.

⟨*annotation*⟩ ::= '`[[`' ⟨*pre*⟩ '`]]`'
| '`[[`' ⟨*post*⟩ '`]]`'

⟨*pre*⟩ ::= '`gsl`' '`::`' '`pre`' '`(`' ⟨*contract*⟩ '`)`'

⟨*post*⟩ ::= '`gsl`' '`::`' '`pre`' '`(`' ⟨*contract*⟩ '`)`'

⟨*contract*⟩ ::= ['`gsl`' '`::`'] '`lifetime`' '`(`' ⟨*contract_expr*⟩
'`,`'
'`{`' { ⟨*ext_contract_expr*⟩ '`,`' } ⟨*ext_contract_expr*⟩
'`}`' '`)`'

⟨*ext_contract_expr*⟩ ::= ['`gsl`' '`::`'] '`null`'
| ['`gsl`' '`::`'] '`global`'
| ['`gsl`' '`::`'] '`invalid`'
| ⟨*contract_expr*⟩

⟨*contract_expr*⟩ ::= '`this`'
| '`Return`'
| ? ident ?
| ['`gsl`' '`::`'] '`deref`' '`(`' ⟨*contract_expr*⟩ '`)`'
| ⟨*contract_expr*⟩ '`.`' ? ident ?
| ⟨*contract_expr*⟩ '`->`' ? ident ?

**Fig. 1.** The grammar of the lifetime contract annotation language

```
void basic(int *a, int *b)
  [[gsl::pre(lifetime(b, {a}))]];

void f() {
  int c, d;
  basic(&c, &d);
}

void g() {
  int array[20];
  basic(array, array+20);
}
```

Listing 3: Simple annotation example.

In Listing 3, the function `basic` has a precondition that both arguments' pointees should have the same lifetime. In our model the same lifetime means that they either need to be the same object or belong to the same Owner or Aggregate (like container or parent array/struct). The function

[1] `gsl` is the namespace used by the C++ Core Guidelines Support Library

call to `basic` in `f` does not fulfill this precondition. Without knowing this precondition, it is not possible to detect the problem. However, the function call in `g` is safe, as both pointers point to something with the same lifetime. One could argue that the individual elements of the array have different lifetime as the array members are created/destructed in a sequential order. However, for our analyses, each aggregate is considered one atomic entity lifetime-wise. Our contracts are following a provenance-based model, meaning that the lifetime of an aggregate member or the pointee of an owner is considered the same as their parents. On the other hand, the variables `c` and `d` in function `f` are considered separate entities in the provenance-based model. They do not belong to the same Owner or Aggregate. This choice might seem arbitrary at this point, but it proved to be a powerful tool to detect iterator misuses where the iterators originated from different containers (see Listing 4), and we have yet to experience any downside to the approach [27].

Note that `array+20` will point past the last element of the array in Listing 3. This is a common pattern in C family languages. Range of elements are represented as follows, the beginning of the range is included but the end of the range is not. Thus, the pointer or iterator referring to the end of the range is never dereferenced. This representation makes it easy to check for empty ranges, we just need to equality compare the beginning and the end of the range. While this specific example is correct, our analysis would not be able to detect out of bounds errors. We do not reason about the indices.

```
template <typename It, typename T>
It find(It begin, It end, const T &val)
    [[gsl::pre(lifetime(end, {begin}))]]
    [[gsl::post(lifetime(Return, {begin}))]];

struct [[gsl::Owner(int)]] MyOwner {
  int *begin()
      [[gsl::post(lifetime(Return, {this}))]];
  int *end()
      [[gsl::post(lifetime(Return, {this}))]];
};

void f() {
  int *res = find(MyOwner{}.begin(),
                  MyOwner{}.end(), 5);
}
```

Listing 4: A real-world example of using lifetime contracts.

Let us look at a more real-world example. The code in Listing 4 has multiple object lifetime problems. First of all, the `find` function expects two iterators that belong to the same container, so they should have the same lifetime. The returned iterator will also have this lifetime. While this contract is quite natural to a seasoned developer, it can be hard to derive this automatically using static analysis. Making all these contracts explicit in the source code opens up new possibilities. Now that the tools can be aware of the lifetime contracts, they can detect that the iterators passed to `find` in function `f` have pointees with separate lifetime. Therefore, we can diagnose the problem. Moreover, the tools can detect that the returned Pointer's pointee has the same lifetime as the first argument's pointee, which is a temporary object in this case, it will be destroyed at the end of the full expression. As a result, the Pointer returned by `find` will dangle.

### 2.3. Abstract Interpretation

The analysis implementation works on the *control flow graph* (CFG) provided by Clang. The nodes of this graph are *basic blocks*, which are sequences of instructions always executed sequentially, while edges are the possible jumps between basic blocks. Note that this representation usually does not encode concurrency related information. An example CFG can be seen below in Fig. 2.

**Analysis domain** The analysis is similar to a points-to analysis. We have a set of abstract memory locations. For each variable with a type from the Pointer category, we maintain a points-to set that is a subset of all the possible memory locations for a given function. These sets are overestimating the possible destinations. For Owners, we will not maintain a points-to set as they will always point to the same abstract memory location in our system.

**Abstract memory locations** Our model defines one abstract memory location for each variable and allocation site. Allocation sites (e.g., new expressions) are not always visible inside the function we analyze. Thus, we have assumed allocation sites for the pointees of owners and the pointees of the function arguments. Each of these abstract allocation sites can correspond to multiple objects at run time. We also have special memory locations to represent uninitialized values, null Pointers, and Pointers we do not reason about. The number of memory locations is finite. We also track the relationship between these memory locations, i.e., we know that what are the parents of a field. Our method is *field-sensitive*. Each structure consists of the memory locations of its fields and each field can have its separate points-to information during the analysis.

We set the pointee to `global` to represent Pointers we do not reason about. We can think of `global` as a special top element in the lattice. The name of this abstract location can be misleading as we use this symbol both to represent that the pointee has provably longer lifetime as the Pointer and that we do not want to reason about the lifetime of the pointee. The main reason for having such a symbol is to reduce the number of false positives for certain code patterns that are really challenging to support.

**Monotonicity** We have a powerset lattice at the heart of the analysis domain, and the join operator is based on the set union. Each set contains abstract memory locations that represents the possible pointees for a pointer. All of our transfer functions can only grow the size of the sets. Since the number of abstract memory locations is finite for each function, and the transfer functions are monotonic since we use the set union as the join operator, the analysis will always terminate. Moreover, since the size of the sets will never decrease, whenever we see a problematic element in the powerset, we can report the error immediately without requiring an extra pass to traverse the CFG and the analysis state to collect the warnings.

**Context-sensitivity** The analysis is not context-sensitive. This is a direct consequence of the annotation language we

```
int collatz(int x) {
  int num = 0;
  while(x > 1) {
    if (x % 2)
        x = 3 * x + 1;
    else
        x = x / 2;
    ++num;
  }
  return num;
}
```

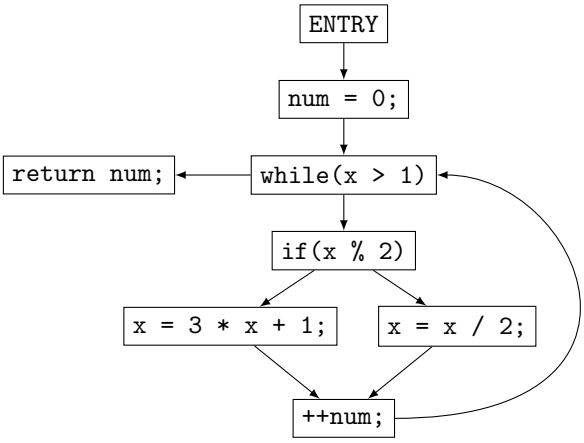


**Fig. 2.** A C function and its simplified control flow graph

have as (just like in Rust) context-dependent lifetime contracts cannot be expressed with them. The primary purpose of the analysis is to find lifetime contract violations; thus, it is not possible to make it meaningfully context-dependent without extending the annotation language. To achieve interprocedural analysis, we check whether the preconditions hold at the call site, and assume the postconditions after the call to model the effect of the call. In the callee, we assume the preconditions and check whether the postconditions are met before returning from the function. See Fig. 3 for details. We do not enforce the postconditions on exceptional paths from throw expressions and assertion failures.

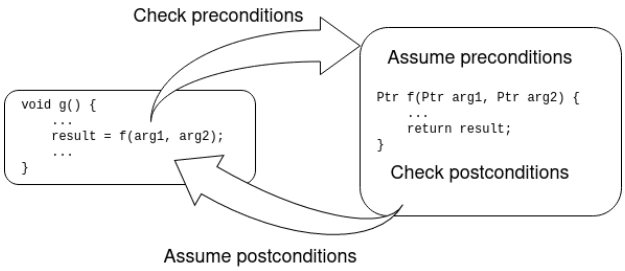


**Fig. 3.** Interprocedural analysis with lifetime contracts

**Performance** The goal of the analysis is to keep the performance good enough so users can turn it on for each compilation. According to our measurements the analysis has about 5% overhead over a full compilation and 10% overhead on a syntax- and typecheck (no code generation). While we did not have time yet to fine-tune the performance we did use some clever tricks to avoid redundant work and improve cache locality. For example, we skip checking of the C++ Standard Template Library (STL) implementation [16]. There still is some room for improvement, for example, we do not use immutable data structures yet to represent the state.

## 3. ELIMINATING FALSE POSITIVES

Unfortunately, paraphrasing Rice's theorem [21]: all non-trivial properties of a program are undecidable at compile time. Thus, static analysis tools often over-approximate or under-approximate the behavior of a program. Consequently, such tools may report false errors (called *false positives*), or they might miss some real problems (called *false negatives*). While verification tools aim to catch all errors at the cost of having a large number of false positives, industrial bug-finding tools aim to have a low false positive rate at the cost of missing some true errors [8]. The reason is economincs, the developer time is valuable, the time spent on evaluating false positives could be spent on fixing known bugs or developing new features.

Traditionally, researchers tend to focus on the formal verification end of the spectrum. Godefroid argues [9] that bug-finding aspect of static analysis is just as important. Peter O'Hearn devised a formal method to argue about the soundness of bug-finding tools [18]. Some authors claim that excess number of false positive findings renders certain checks useless [14]. Ranking static analysis results is an active research area that employs statistics and machine learning [10, 14].

In this paper, we introduce two methods that help fine-tuning an analysis match the rigor that is justified by the requirements of the analyzed project.

### 3.1. Local path-sensitivity

Whenever the analysis evaluates a condition it will update the state of the Pointers. On the branch where the Pointer is evaluated to true we assume that the Pointer is non-null while in the other branch we assume that the Pointer is null. This helps preventing some false positives like in the code below.

```
void f(int *p) {
  if (!p)
    return;
  *p = 5; // no warning
}
```

Listing 5: Dereference guarded by early return.

Initially, the analysis assumes that the value of `p` can be null. After the `if` statement on the then branch we know that the value of the pointer must be null. Similarly, on the implicit fallthrough branch we know that the value of the

pointer cannot be null. Hence, the analysis will not warn on the dereference, it is safe.

This method also happens to work for compound conditions like in the code below:

```
void f(int *p, int *q) {
  if (p && q) {
    *p = 5; // no warning
    *q = 3; // no warning
  }
}
```

Listing 6: Dereference guarded by compound conditional.

We only take the true branch of the conditional statement when both pointers are evaluated to true, so none of them can be null.

It was quite counter intuitive when we discovered that a slightly rewritten version of the code above will not work as expected:

```
void f(int *p, int *q) {
  assert (p && q);
  *p = 5; // warning: possible null dereference
  *q = 3;
}
```

Listing 7: Dereference guarded by compound assertion.

The `assert` above is a macro that is expanded in Fig. 4.

Where the false branch of the ternary operator is a `noreturn` function stopping the analysis on that path. The source of the problem is the explicit cast to boolean. The main reason for this cast is to make sure that types with explicit conversion operators work in assert contexts. This conversion operator creates an unnamed temporary. This temporary is at a join point in the CFG which results in merging the analysis states corresponding to the true and false branches. Later on, we branch on this temporary. At this point the states were already merged and the analysis can no longer figure out the correct nullness of the pointers.

This pattern of assertive programming is widely used. It is not an option to not support this pattern as it would hinder the adoption of the analysis. A programmer would know that the explicit conversion there has the only role to make the type system contented but it does not change the semantics of the program. Path-sensitive analyses have no problem modeling this as they keep multiple program states for each CFG node, one for each paths from the ENTRY to the node. The cost of this precision is exponential memory and runtime complexity.

To avoid the cost of the path explosion while reaping some of the benefits we introduced local path-sensitivity to the analysis. Whenever the tool encounters this pattern, it will not merge the states for the temporary result of the explicit conversion. It will keep two separate states, one for the true condition and one for the false. Later on, it will propagate the true states to the successors on the then branch and the false states to the successors on the else branch. Using this method, we were able to make assertions work as expected with the analysis.

This local path-sensitivity does not cause an exponential explosion of paths as path constraints have at most one temporary at a time. The fact that we could improve the precision of the analysis with this method is a proof of the lack of distributivity, i.e., the transfer functions do not distribute over the join operation. See the equation NON-DISTR, where $F_{B,truth}$ stands for the transfer function of the basic block $B$ for the output edge $truth$, $\sqcup$ is the join operation and $s$ is the initial analysis state. The formula is based on the CFG in Fig. 4.

As a result, the meet-over-all-paths solution is more precise than the maximal-fixed-point algorithm for this analysis.

An alternative would be to prune the explicit conversion from the CFG. This was not an option as the Clang CFG is very closely coupled to the abstract syntax tree by design. Moreover, it could change the result of other analyses as Clang attempts to reuse the CFG as much as possible to keep the run time low.

After implementing this method we tested the analysis on some open-source projects and this pattern of false positive findings disappeared. Unfortunately, we are not aware of any open-source projects that adopted this analysis (with consistent use of `gsl::not_null` to mark non-null pointers and following some other guidelines). This makes it meaningless to evaluate the change in the false positive ratio as the original analysis was not designed to work with arbitrary C++ code.

### 3.2. Dominance- and reaching-definitions-based filtering

Different projects have different safety requirements. For some projects, the authors are willing to evaluate large number of false positive results to find a few true positives. Sometimes, they are even willing to rewrite correct code to suppress a false positive result. Most projects, however, are not safety critical. Authors of those projects do not want to be bothered with false positive findings. They are interested in bug-finding tools as a productivity boost. A tool can save time if it can discover hard-to find or debug problems without wasting too much time with false positives.

According to our experience, bugs on infeasible paths are the most common sources of false positives for our flow-sensitive lifetime analysis. To mitigate this problem and make it useful for a wider audience, we introduced a filtered version of the analysis. This filtered analysis will only report bugs that are guaranteed to be on a feasible path at the cost of additional false negatives, i.e., missed bugs. This filtered analysis can transform a tool that is closer to verification and useful for safety critical software into a bug-finding tool that is useful for most industrial projects.

This section introduces this filtered analysis that is based on dominance relationships and reaching definition analysis. We first introduce these definitions and showcase some examples how can they prevent false positive findings. Note that this is only a filter on the top of the existing flow-sensitive lifetime analysis and it could be reused for other analyses as well.

Firstly let us define dominance, which will help us reason about some code examples.

**Definition 3.1. (Dominance)** *Node $v \in CFG$ dominates node $n \in CFG$ iff all paths from the entrance node to $n$ go through $v$.*

```
void f(int *p, int *q)
{
  (bool)(p && q) ?
    0 :
    __assert(...);
  *p = 5; // warning
  *q = 3;
}
```

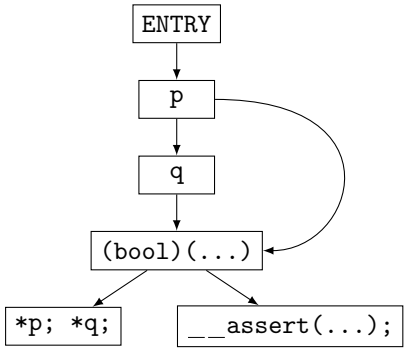


**Fig. 4.** Expansion of the assert macro and its control flow graph

$$F_{cast,true}((F_{q,true}(F_{p,true}(s)) \sqcup F_{q,false}(F_{p,true}(s))) \sqcup F_{p,false}(s)) \neq$$
$$F_{cast,true}(F_{q,true}(F_{p,true}(s))) \sqcup F_{cast,true}(F_{q,false}(F_{p,true}(s))) \sqcup F_{cast,true}(F_{p,false}(s)) \quad \text{(NON-DISTR)}$$

**Fig. 5.** Counter example to prove that the lifetime analysis is not distributive

**Definition 3.2. (Post-dominance)** *Node $v \in CFG$ post-dominates node $n \in CFG$ iff all paths from $n$ to the exit node go through $v$.*

**Definition 3.3. ((Post-)dominators)** *For node $v \in CFG$, $v$'s (post-)dominator set contains all nodes $n \in CFG$ that (post-)dominate $v$.*

```
void f(...) {
  p = getNonNullPtr();
  if (cond())
    p = 0;
  if (cond2())
    *p = 5;
}
```

Listing 8: Possible null pointer dereference without dominance relationship.

Listing 8 has a possible null pointer dereference error. We assume that all the branches can be taken. As a result, pointer p can be null. The dereference of the pointer is not guarded by a null check. Hence, the analysis will report a warning. Note that this might be a false positive. In case the two conditions `cond` and `cond2` are not independent, the error might be on an infeasible path. I.e., the program might never take both of the branches. In that case, the finding is a false positive.

```
void f(...) {
  p = 0;
  if (cond2())
    *p = 5;
}
```

Listing 9: Possible null pointer dereference with dominance relationship.

```
void f(...) {
  p = getNonNullPtr();
  if (cond())
    p = 0;
  *p = 5;
}
```

Listing 10: Possible null pointer dereference with post-dominance relationship.

Listing 9 has a similar code snippet. Here, however, we only have one branch. The assignment that sets the pointer p dominates the dereference. Thus, every path that reaches the dereference has to set the value of the pointer to null. Such code either has a null pointer dereference or it has dead code. Either way, it is always worth report this to the user. Similarly, Listing 10 is also a true positive finding because the dereference post-dominates the null assignment.

```
void f(...) {
  p = 0;
  if (cond())
    p = getNonNullPtr();
  if (cond2())
    *p = 5;
}
```

Listing 11: Possible null pointer dereference with multiple reaching definitions.

Sadly, checking domination relationship only is not sufficient to exclude all the false positives from infeasible paths. Listing 11 has a snippet where the null assignment dominates the dereference. This report would not be filtered by a dominator based filtering algorithm. However, it still can be a false positive, in case we always update the pointer before dereferencing it. The gist of this example is the fact that there are multiple values that can flow to the dereference. Fortunately, there is a well known way to formalize this phenomenon.

**Definition 3.4. (Definition)** *The definition of a value is the instruction where the value was created/assigned.*

**Definition 3.5. (Reaching definitions)** *For a given instruction, the set of reaching definitions contains definitions that can reach this instruction without being overwritten.*

To exclude the code from Listing 11, we will only report warnings that have the following properties:

- The source dominates the dereference, or the dereference post-dominates the source. Note that the source is not always an assignment, it can be other events like a variable going out of scope, being deleted, or the memory of a container is being reallocated.
- The value of the pointer is not changed on any paths between the source and the dereference. I.e., there are no reaching definitions of the dereference that is reachable from the source.

Calculating the dominator and reaching definition sets are classical dataflow problems that are discussed in detail in textbooks. In this paper, we used the algorithms from the book Engineering a compiler [5]. See the algorithm to calculate dominator sets below. We denote the predecessors of a basic block $n$ with $preds(n)$.

$$\begin{aligned} Init &: Dom(n_0) = \{n_0\} \\ &\quad Dom(n) = \{n_0, n_1, ..., n_m\}, \forall n \neq n_0 \\ Iter &: Dom(n) = \{n\} \cup \left( \bigcap_{m \in preds(n)} Dom(m) \right) \end{aligned} \tag{1}$$

Note that post-dominator sets are equivalent with the dominator sets on a reversed CFG if there is a single dedicated exit node.

**Definition 3.6. (Downward-exposed definitions)** *Downward-exposed definitions for a node $n \in CFG$ ($DEDef(n)$) are the set of definitions that are not redefined in the same basic block.*

**Definition 3.7. (Definition kill)** *The $DefKill(n)$ set contains all the definitions from any CFG nodes that are redefined by a definition in n.*

Using these definitions, we can formulate the algorithm to calculate reaching definition sets. It is shown below.

$$\begin{aligned} &Init : Reaches(n) = \emptyset, \forall n \in CFG \\ &Iter : Reaches(n) = \bigcup_{m \in preds(n)} \Big( DEDef(m) \cup \\ &\qquad (Reaches(m) \cap \overline{DefKill(m)}) \Big) \end{aligned} \tag{2}$$

Unfortunately, Clang does not have a reaching definition algorithm and it is quite an effort to develop one. As a workaround, we implemented an algorithm to check if the analysis state of the pointer changes on any of the paths between the source and the dereference.

This filtering also provides the users with a good migration path to introduce the full flow-sensitive lifetime analysis. Users can start fixing filtered problems first, and turn the filtering off gradually for each translation unit to move towards a warning free code base.

At the time of writing the paper, we were not aware of any open-source project that is following the relevant parts of the C++ Core Guidelines making the evaluation of this filtering harder. Running the flow-sensitive analysis on a project that does not follow the assumptions of the analysis does not provide a meaningful measure about the usefulness of the analysis. These assumptions are the following:

- Non-null Pointers in formal parameters are marked with the appropriate annotation or type
- Output only and input/output formal parameters can be distinguished (using annotation or a convention, e.g. references are always input/output).
- For the rare case when the user defined Owners and Pointers are not recognized correctly, they are annotated accordingly.
- For the rare case when the default inferred function contract triggers false positives, the function is annotated.

For the reason mentioned above it is not meaningful to calculate the false positive ratio before and after applying this filter. We run the filtered version of the analysis on Fuchsia [31] including all of its dependencies. Fuchsia is an open-source capability-based operating system with a strong focus on security mainly developed by Google. After reviewing manually the filtered results, there were no false positive results due to the infeasible path problem. This method successfully filtered a whole class of false positives. Despite the aggressive filtering, the analysis were able to find useful results.

Alternatively, this filter is also suitable to rank certain results higher that are less likely to be false positives. This helps the developers to first focus on fixing warnings that are more likely to uncover true errors.

## 4. RELATED WORK

Rust is referenced in this paper a lot. It is a systems programming language with great safety features. Safe Rust enforces a strict property: there is only one mutable reference to an object at a time. This helps Rust analysis to utilize distributive frameworks in the form of Gen/Kill sets which circumvents some of the problems we solved in this paper. It is not realistic to expect industrial users of the lifetime analysis to rewrite their code in a style that enforces this property. Moreover, Rust is not a silver bullet. A study about the use of `unsafe` in Rust [19] argues that `unsafe` is often inevitable and its usage can introduce bugs into Rust code. The study recommends developing and using additional tooling to catch bugs in Rust code.

Cclyzer [2] is a points-to analysis on the LLVM bitcode. While the application of this tool differs from our lifetime analysis, this work promotes field- and type-sensitivity in the points-to analysis for C++. These properties are also present in Herb Sutter's analysis.

Encoding ownership information in the type system explicitly (or inferring them implicitly) is an active area of research [4]. While it would be possible to extend the type system of C++ with type category information, we did not see any advantages over our current solution. Our main reason is that type categories are derived from empirical observations. Thus, we do not plan to formalize the category inference.

ARC++ [29] also works by abstract interpretation; hence it has similar trade-offs compared to the lifetime analysis described in this paper. It is, however, using a slightly different approach. The authors introduced an abstract representation of the C++ source code that makes object creation, uses, and destruction explicit. They also defined lifetime dependency in order to link objects that have related lifetime semantics.

Ironclad C++ is a project that aims at type and memory safety in C++ [7]. This solution proposes a C++ template library as an essential type system that enforces the safety rules. Dangling pointers are considered, but the approach does not deal with dangling pointers to former temporary objects. Also, this solution does not distinguish between user-defined classes based on their category.

A common alternative to static analysis is dynamic analysis. The most popular C++ compilers support sanitizers [22] that can help us catch lifetime-related problems with no false positives. Unfortunately, these methods can also have false negatives. More importantly, not all of the code is checked; only the parts that were executed during testing.

## 5. FUTURE WORK

We plan to continue investigating what are the major obstacles preventing the adoption of Herb Sutter's flow-sensitive lifetime analyis. For each obstacle, we want to introduce mitigations to give users a migration path from non-conforming code that produces high volume of warnings to conforming code. This migration path should help users to fix warnings incrementally starting with the most severe ones that are the most likely to be true positives.

Moreover, we believe that the local path-sensitivity can be generalized to every CFG node that does not change the analysis state. We call such nodes transparent. Unfortunately, it is analysis dependent to decide which nodes are transparent. We believe this approach can increase the precision of the analysis significantly without exploding the run time and memory usage. It can also provide the users with a knob that can be used to fine tune the balance between performance and precision.

## 6. CONCLUSION

Object lifetime related errors are a significant problem for C family languages. While we have some tools to prevent some of these errors, their adoption can be hindered by false positive findings. Bug reports need to be reviewed by developers one-by-one in order to be corrected. If the tool presents an overwhelming amount of false warnings to the developer, it becomes cumbersome to use, and developers eventually lose their trust and interest in the tool.

In this paper, we introduced two methods to get rid of some false positive findings. The first method helps eliminate some false findings that are specific to some coding patterns in C and C++. This method is adding local path-sensitivity in some cases. This does not result in path explosion and does not introduce additional false negatives.

The second method, on the other hand, eliminates every warning that can be prone to the infeasible path problem. While this approach eliminates a large class of errors it comes at the cost of introducing significant amount of false negatives. The applicability of this dominance-based filtering depends on the requirements of the analyzed project.

While we did not eliminate the need to follow some guidelines to use this analysis, we made it easier to conform to those guidelines. Reducing the amount of work needed to be done by the user is key to make the analysis more popular. Note that one of the main reasons for the lack of users is the fact that the implementations of the original analysis is not finished yet, some language features such as move semantics are not yet supported. Nevertheless, it is worth exploring how to make the analysis more user-friendly even before it reaches maturity.

We believe that these results will help the adoption of the analysis as a statement-local variant of this analysis already proved to be successful [11] and is available in Clang 10 and on by default.

## REFERENCES

[1] BABATI, B. – HORVÁTH, G. – MÁJER, V. – PATAKI, N.: Static Analysis Toolset with Clang. Proceedings of the 10th International Conference on Applied Informatics (2017), pp. 23–29 https://doi.org/10.14794/ICAI.10.2017.23

[2] BALATSOURAS, G. – SMARAGDAKIS, Y.: Structure-sensitive points-to analysis for C and C++. International Static Analysis Symposium (2016), pp. 84–104.

[3] BLANDY, J.: The Rust Programming Language: Fast, Safe, and Beautiful. O'Reilly Media, Inc. (2015).

[4] CLARKE, D. – ÖSTLUND, J. – SERGEY, I. – WRIGSTAD, T.: Ownership types: A survey. Aliasing in Object-Oriented Programming. Types, Analysis and Verification (2013), pp. 15–58.

[5] COOPER, K. D. – TORCZON, L.: Engineering a compiler (2nd ed.). Elsevier/Morgan Kaufmann (2011).

[6] COUSOT, P. – COUSOT, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (1977), pp. 238–252.

[7] DELOZIER, C. – EISENBERG, R. – NAGARAKATTE, S. – OSERA, P.-M., MARTIN, M. – ZDANCEWIC, S.: Ironclad C++: A library-augmented type-safe subset of C++. Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (2013), pp. 287–304. http://doi.acm.org/10.1145/2509136.2509550

[8] EMANUELSSON, P. – NILSSON, U.: *A comparative study of industrial static analysis tools*. Electronic notes in theoretical computer science **217** (2008), pp. 5–21.

[9] GODEFROID, P.: The soundness of bugs is what matters (position statement). BUGS2005 (PLDI2005 Workshop on the Evaluation of Software Defect Detection Tools) (2005).

[10] HECKMAN, S. – WILLIAMS, L.: *A systematic literature review of actionable alert identification techniques for automated static code analysis*, Information and Software Technology **53**, No. 4 (2011) 363–387.

[11] HORVÁTH, G. – PATAKI, N.: Categorization of C++ classes for static lifetime analysis. Proceedings of the 9th Balkan Conference on Informatics, BCI 2019, pp. 21(1)–21(7). https://doi.org/10.1145/3351556.3351559

[12] HOSFELT, D.: Implications of rewriting a browser component in Rust. 2020. https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/

[13] KANVAR, V. – KHEDKER, U. P.: *Heap abstractions for static analysis*. ACM Comput. Surv. **49**, No. 2 (2016). https://doi.org/10.1145/2931098

[14] KREMENEK, T. – ASHCRAFT, K. – YANG, J. – ENGLER, D. R.: Correlation exploitation in error ranking. Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (2004), pp. 83–93. https://doi.org/10.1145/1029894.1029909

[15] LOPES, B. C. – AULER, R.: Getting Started with LLVM Core Libraries. Packt Publishing (2014).

[16] MEYERS, C.: Effective STL. Addison-Wesley (2001).

[17] MILLER, M.: Trends, Challenges, and Strategic Shifts in the Software Vulnerability. Mitigation Landscape, 2018. https://www.usenix.org/conference/woot19/presentation/miller

[18] O'HEARN, P. W.: Incorrectness logic. Proceedings of the ACM on Programming Languages **4**, POPL (2019), pp. 1–32.

[19] QIN, B. – CHEN, Y. – YU, Z. – SONG L. – ZHANG, Y.: Understanding memory and thread safety practices and issues in real-world Rust programs. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (2020), pp. 763–779.

[20] REYNOLDS, J. C.: Separation logic: A logic for shared mutable data structures. Proceedings 17th Annual IEEE Symposium on Logic in Computer Science (2002), pp. 55–74.

[21] RICE, H. G.: *Classes of recursively enumerable sets and their decision problems*. Trans. Amer. Math. Soc., **74** (1953), pp. 358–366.

[22] SEREBRYANY, K. – BRUENING, D. – POTAPENDO, A. – VYUKOV, D.: AddressSanitizer: A fast address sanity checker. Proceedings of the 2012 USENIX Conference on Annual Technical Conference (2012), pp. 28–28.

[23] SMARAGDAKIS, Y. – BALATSOURAS, G.: *Pointer analysis*, Foundations and Trends ®in Programming Languages **2**, No. 1 (2015) 1–69. http://dx.doi.org/10.1561/2500000014

[24] STROUSTRUP, B.: The C++ Programming Language (4th Edition). Addison-Wesley (2013).

[25] STROUSTRUP, B.: Thriving in a Crowded and Changing World: C++ 2006–2020. Proceedings of the ACM on Programming Languages (HOPL), Article 70, 167 pages. https://dl.acm.org/doi/abs/10.1145/3386320

[26] STROUSTRUP, B. – SUTTER, H.: C++ Core Guidelines, a set of tried-and-true guidelines, rules, and best practices about coding in C++ (2016). https://github.com/isocpp/CppCoreGuidelines

[27] SUTTER, H.: Lifetime safety: Preventing common dangling. Microsoft Corporation, Tech. Rep., (2018).

[28] SUTTER, H.: Thoughts on a more powerful and simpler C++ (5 of N). https://www.youtube.com/watch?v=80BZxujhY38

[29] XIAO, X. – BALAKRISHNAN, G. – IVANČIĆ, F. – MAEDA, N. – GUPTA, A. – CHHETRI, D.: ARC++: effective typestate and lifetime dependency analysis. Proceedings of the 2014 International Symposium on Software Testing and Analysis (2014), pp. 116–126.

[30] Chromium memory safety report (2020). https://www.chromium.org/Home/chromium-security/memory-safety

[31] Fuchsia, an open-source capability-based operating system (2020). https://fuchsia.dev/fuchsia-src/concepts

Received September 23, 2020, accepted December 4, 2020

## BIOGRAPHIES

**Gábor Horváth** was born on 26. 08. 1991. In 2016, he graduated (MSc) from Eötvös Loránd University in Budapest. He finished writing his PhD dissertation about static analysis of C++ software recently and awaiting the defense. During his studies he interned at several software companies that helped him focusing on research questions with immediate applicability. His research is focusing on symbolic execution and abstract interpretation methods for real-world use cases.

**Norbert Pataki** was born on 26. 2. 1982. He defended his PhD thesis in 2013 at the Eötvös Loránd University, Budapest. In 2020, he became a habilitated associate professor. He teaches C++ programming language courses at the university since 2005. He also participates in teaching C programming and software project tools. His main research areas are C++, programming languages, static analysis and DevOps.