# Detecting C++ Lifetime Errors with Symbolic Execution

Réka Kovács
Eötvös Loránd University
Budapest, Hungary
rekanikolett@caesar.elte.hu

Gábor Horváth
Eötvös Loránd University
Budapest, Hungary
xazax@caesar.elte.hu

Zoltán Porkoláb
Eötvös Loránd University
Budapest, Hungary
gsd@inf.elte.hu

## ABSTRACT

One of the reasons why it is so hard to statically analyze C++ source code is because of its *Standard Template Library (STL)*. The STL is a monstrous collection of complex code base whose semantics is hard for static analyzers to understand. Unfortunately, many of the most serious memory management bugs in C++ are connected to the lifetimes of STL containers. This paper describes a method of adding knowledge of STL ownership semantics to a static analysis engine. It was implemented in an open-source symbolic execution framework widely used in the industry, and produced new and serious lifetime-related error reports in popular open-source projects.

## CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; Software maintenance tools.

## KEYWORDS

C++, STL, lifetime, static analysis, symbolic execution

**ACM Reference Format:**
Réka Kovács, Gábor Horváth, and Zoltán Porkoláb. 2019. Detecting C++ Lifetime Errors with Symbolic Execution. In *Proceedings of 9th Balkan Conference in Informatics (BCI '19).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

C++ language implementations come with a set of standard utilities called the Standard Template Library (STL). The STL is a huge, complicated code base tuned for high performance, presenting a challenge to source code analysis tools.

This presents a real problem. Memory management bugs are the most infamous issues of the C programming language family, and can cause undefined behavior and runtime errors. Big tech companies reporting that 70% of their bug fixes amend memory errors [19]. Static program analysis tools can come to our aid in catching the most serious issues early in the process, during development. This, however, is not the reality for modern C++ software yet. In modern C++ programs, a great deal of memory-related errors revolve around object lifetimes, and their understanding requires knowledge of the STL and its ownership semantics.

*BCI '19, September 26–28, 2019, Sofia, Bulgaria*
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This is what most C++ analyzers still lack. To our knowledge, there are no freely available bug finding static analysis tools that can detect all lifetime-related errors involving STL containers. Some notable efforts to mitigate this issue are discussed in Section 2.

The authors are contributors to an open-source static code analysis tool called the Clang Static Analyzer (henceforth also referred to as *the Analyzer*) [32]. It is a powerful symbolic execution engine built atop the Clang [31] open-source compiler for C family languages, It is one of the most widely used freely available tools in the software engineering industry.

Prior to our work, the Analyzer was able to find a wide range of C-style use-after-free errors through its memory checkers. *Checkers* are special modules "plugged into" the analysis process. They monitor the state of the program as seen by the symbolic execution engine, extract additional information from the source code, and flag potentially erroneous code lines as such.

Our contribution is enhancing the engine with knowledge about the semantics of the `std::string` type and its API. This way it is able to find sophisticated memory management errors stemming from raw pointers outliving the container they point into. Our approach can be easily extended to other STL containers, and can also support `std::string_views` and user-defined containers.

The paper is structured as follows. Section 2 compares our approach to other research efforts dealing with lifetime issues of C++ containers. Section 3 gives an overview of symbolic execution, which is the main working mechanism of the analysis framework we extended. Section 4 details the method we used to add information about STL semantics to the analysis. In Section 5, we describe our experiences of running the enhanced analysis on popular C++ projects. Section 6 lists some possible improvements we plan in the future, and finally Section 7 summarizes our findings.

## 2 RELATED WORK

The significance of lifetime related issues in C++ is best described in Herb Sutter's paper *Lifetime safety: Preventing common dangling* [15], available in the C++ Core Guidelines repository. Herb Sutter has been the chairman of the ISO C++ Committee since 2009. He introduces a methodology for catching use-after-free errors in C++ by categorizing variable types into generalized "Owner" and generalized "Pointer" classes (among others). He also proposes a flow-sensitive analysis to produce warning based on these type categories, which is currently being implemented in some of the most popular C++ compilers, Clang and the Microsoft Visual C++ Compiler (MSVC).

Herb Sutter's analysis will find many lifetime-related problems in compile time, but it will still suffer from the limits of flow-sensitive analysis. Flow-sensitive analyses do not consider whether a given execution path is feasible during real execution, and is prone to

```
void g(int b, int &x) {
   if (b)
      x = b+1;
   else
      x = 42;
}
```

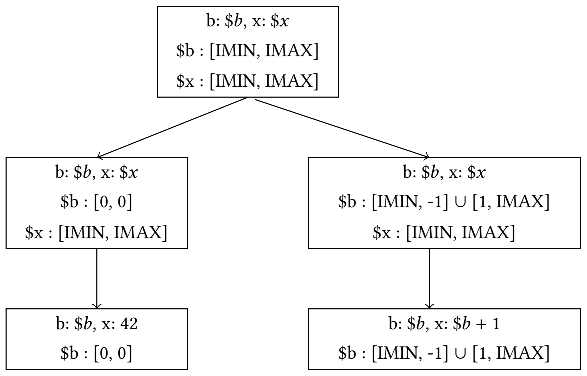

**Figure 1: A simplified version of the exploded graph built during the symbolic execution of a simple function. The right-hand side of the graph represents the *true* branch of the `if` statement, while the left-hand side describes the *false* branch.**

producing false positive reports. The Clang Static Analyzer, however, performs path-sensitive analysis, discarding paths found to be infeasible by a built-in constraint solver.

Another work dealing with lifetime issues in C++ is an abstract interpreter called ARC++ [37]. The authors of ARC++ introduced an abstract representation of the C++ source code that makes object creation, use, and destruction explicit. They also defined a lifetime dependency analysis to link objects that have connected lifetime semantics. Unfortunately, abstract interpretation suffers from the same false positive problem as flow-sensitive analysis.

On the other end of the spectrum, one can attempt to *verify* the usage of STL, such as in [5], which uses counterexample-guided abstraction refinement. Similarly, the CMC model checker [20] verifies C++ code compiled with templates and STL constructs via explicit-state exploration. The disadvantage of verifying tools is that their duration times are often not acceptable for less safety-critical industrial projects.

## 3 SYMBOLIC EXECUTION

This section introduces symbolic execution in depth, concentrating on methods used by the Clang Static Analyzer.

Symbolic execution interprets the source code, assigning a symbol to represent each unknown value. Calculations are carried out symbolically. During the interpretation process, the analyzer attempts to enumerate all possible execution paths. To represent the internal state of the analysis, the analyzer uses a data structure called the *exploded graph* [27]. Each vertex of this graph is a (symbolic state, program point) pair. A *symbolic state* corresponds to a set of real program states, while the *program point* determines the current location in the program, similarly to an instruction pointer. The edges of the graph are transitions between vertices. Memory is represented using a hierarchy of memory regions [39]. The analyzer is building the graph on demand during the analysis using a *worklist algorithm* that implements a path-sensitive walk over the control flow graph (CFG).

The symbolic state consists of 3 components:

- **Environment**: A mapping from source code expressions to symbolic expressions.
- **Store**: A mapping from memory locations to symbolic expressions.
- **Generic data map**: A data structure where the analysis engine and checks store domain-specific information.

During the execution of a path, the analyzer collects constraints on symbolic expressions called *path constraints*. The built-in constraint solver represents these constraints as a disjunction of ranges. The constraints are most importantly used to skip the analysis of infeasible paths, but the constraint solver can also be utilized by the checkers to query certain information about the program states. This functionality can be used, for instance, to detect array out-of-bounds accesses or division by zero errors.

An example analysis can be seen along with its simplified exploded graph in Figure 1. Each box in the exploded graph represents a symbolic program state. The first line of the box is the *store*, the other lines represent path constraints over symbols collected during exploration. We omitted any remaining components for brevity.

The function `g` has two execution paths. Since the value of `b` and `x` is initially unknown, these values are represented by the corresponding symbols `$b` and `$x`. As the analysis continues, on one of the execution paths the value of `b` is known to be zero, and later on this path we discover that the value of `x` is the constant 42. The symbol `$x` is no longer needed on this path. On the second path, the value of `b` can be anything but zero. On this path, we also discover that the value of `x` is one larger than the original value of `b`. The symbol `$x` is no longer needed on any of the paths, it can be garbage collected.

After this introduction into symbolic execution, the next section describes our extension that added information about STL semantics to the analysis process.

## 4 DETECTING CONTAINER LIFETIME BUGS

### 4.1 Motivation

Let us consider the code snippet in Listing 1. This shows a C-style function allocating memory on the heap and creating a new raw pointer `c` pointing to that memory. The use of `c` after releasing the memory is a typical memory management bug, and most static analyzers (including the Clang Static Analyzer) report a use-after-free bug on Line 10.

```
#include <cstdlib>

void use(char *) {}

void use_after_free() {
    char *s = (char *) std::malloc(10 * sizeof(char));
                 // Note: Memory is allocated
    char *c = s;
    std::free(s); // Note: Memory is released
    use(c); // Warning: Use of memory after it is freed
}
```

**Listing 1: A C-style use-after-free error. Comments indicate warnings and notes emitted by the Clang Static Analyzer.**

In Listing 2, we see a conceptually equivalent version of the previous example, only that this time we entrust the C++ STL type `std::string` to allocate the string for us. We obtain a raw pointer `c` pointing to the `s`'s inner buffer with its method `c_str()`, then `clear()` its contents, invalidating `c`. The use of `c` in Line 9 is just as much of a use-after-free type of error as that in Listing 1. Still, most tools (including the Clang Static Analyzer) fail to report a problem for this code.

```
1  #include <string>
2
3  void use(const char *) {}
4
5  void use_after_clear() {
6      std::string s = "hello";
7      const char *c = s.c_str();
8      s.clear();
9      use(c);
10 }
```

**Listing 2: A C++-style use-after-free error that uses the `std::string` STL type.**

We can say that the fundamental piece of information missing is the connection between the object owning the memory (in our example, our `std::string`-type `s`), and the raw pointer `c` giving access to the object's internals, thus tied to its lifetime. Additionally, we need to incorporate knowledge of the C++ language standard, which clearly lists those situations when the internals of an STL object change so significantly that all pointers pointing to its internals need to be invalidated.

## 4.2 Modeling the `std::string` API

*Path-sensitivity.* One advantage of working with a static analysis tool built atop an open-source compiler is that we have the code's *abstract syntax tree (AST)* at hand. This means that even though the tool is capable of path-sensitive analysis, we can choose to add path-insensitive, syntactic operations that match the AST, saving resources for the rest of the analysis.

It was our conscious decision, however, to implement the modeling logic in a path-sensitive manner. Let us consider the example in Listing 3.

```
  return std::to_string(name).c_str();
```

**Listing 3: An easy-to-miss statement local use-after-free bug.**

The bug in Listing 3 can be described in terms of path-insensitive analysis by "`c_str()` is called on a temporary object expression". However, avoiding false positive (i.e. bogus) warnings rapidly becomes more and more complicated.

```
  strcpy(dest, std::to_string(name).c_str());
```

**Listing 4: A valid case of calling `c_str()`, on a temporary string.**

For example, the code in Listing 4 is valid because the destructor of the string is invoked after `strcpy()`, but avoiding a report here would require additional reasoning like "... and the result is not consumed before the end of the full expression."

Another scenario that path-sensitive analysis enables us to handle is seen in Listing 5.

```
const char *f(bool cond) {
   std::string s;
   if (cond)
      return s.c_str();
   else
      //...
}
```

**Listing 5: A case only discovered by path-sensitive analysis.**

In this case, the local variable `s` is only destroyed at the end of the function, by an automatic destructor, but we are still able to find the bug.

*Implementation.* Our basic idea is to keep record of raw pointers referring to the inner buffer of an `std::string` container (obtained by a `c_str()` or `data()` call) in the program state. The task is to recognize if any of the tracked pointers is used after a potentially invalidating operation.

Because this is a kind of a use-after-free problem, much of the functionality we wish to have has already been implemented in an Analyzer module called *Malloc Checker*. The name of Malloc Checker is somewhat misleading, as it finds general cases of use-after-free, double-free, and similar issues by holding information about symbols referring to memory returned by allocation functions (such as `malloc()`, `alloca()`, the C++ new operator, etc.) in its own data structure in the program state.

What we need to add is a tracking of `std::string` API calls. Because *checkers* provide the "subscription" mechanism in the Analyzer we look for, we implemented the `std::string` API modeling in the form of a *modeling*-type checker module. The role of our API-specific modeling checker is to figure out exactly what to track in case of a particular container object, and it can then make good use of Malloc Checker's information about whether the memory associated with a symbol has been released. We also added our own bug reporter visitor to attach useful notes to the bug path leading to this new type of use-after-free warning.

Our implementation can be summarized in two points:

(1) We implemented a new module called *Inner Pointer Checker* that tracks raw inner pointers of strings, and recognizes operations on the string that may corrupt the inner buffer. In such cases, it "hands over" the corrupted pointer symbol to Malloc Checker in a "released" state.
(2) We extended Malloc Checker to recognize these special, inherently "released" symbols originating from Inner Pointer

Checker, so that it can issue an appropriate warning message and appropriate diagnostic notes along the bug path.

Before we began our work, the Analyzer already posessed the infrastructure needed to find use-after-free bugs. Our contribution was to teach the Analyzer the semantics of STL, which is essentially about which variables alias each other.

### 4.3 Results

Our module is available in the main Clang repository since version 7.0 [23]. It can be used on one file by invoking the command seen in Listing 6. Our work is enabled by default.

```
$ clang --analyze file.cpp
```

**Listing 6: Invoking the Clang Static Analyzer on one file.**

Clang Static Analyzer checkers can also be accessed from within *Clang-Tidy* [33] (another static analysis tool in the Clang family). To conveniently analyze whole software projects, we recommend using one of the open-source tools created for this purpose: *scan-build* (in the Clang repository) or *CodeChecker* [12].

We believe nothing illustrates our results as well as an example. Returning to Listing 2, as the result of our efforts, the Clang Static Analyzer now emits a warning as seen in Listing 7.

```
#include <string>

void use(const char *) {}

void use_after_clear() {
    std::string s = "hello";
    const char *c = s.c_str();
    // Note: Pointer to inner buffer of 'std::string'
    //       obtained here
    s.clear(); // Note: Inner buffer of 'std::string'
               //       reallocated by call to 'clear'
    use(c);    // Warning: Inner pointer used after
               //          re/deallocation
}
```

**Listing 7: The erroneous code in Listing 2 is now recognized by the Clang Static Analyzer, resulting in the path-sensitive bug report illustrated in comments.**

### 4.4 Modeling other containers

So far we have focused on `std::strings`. The reason for this is that the misuse of its `c_str()` function is one of the most frequently occurring (and, we could say, "iconic") lifetime related error in C++. The foundation we built considering `std::strings` can however be extended to deal with other container structures as well.

Adding support for STL containers with a similar usage profile to `std::strings` is relatively straightforward. E.g. the `std::vector` and `std::array` classes also have a `data()` method that returns a raw pointer to their internals, similarly to the `c_str()` method of `std::string` (additionally, `std::string` also has a `data()` method). After a crosscheck with the standard, support for these classes could be added easily. The main reasons for it not being realized yet are resource limits in our team.

There exist some open-source libraries that are not part of the Standard Template Library, but are used almost as widely in practice as their counterparts in the STL, e.g. Boost, WebKit, LLVM. Some of the most used containers in these libraries could be hard-coded into our checker module like `std::string`, but this would raise maintenance problems. Such projects tend to change dynamically, supported by a huge developer community (unlike our Analyzer), and we also need to be aware of the size and speed of our checker module. Furthermore, user-defined containers need a different kind of solution. We explore possible directions to support non-STL containers in Section 6.1.

## 5 EVALUATION

### 5.1 Results on open-source projects

We ran the analysis enhanced with our STL semantics modeling on a number of open-source software projects written in C++ that we found to use `std::strings` extensively, together with the libraries they depend on: Bitcoin [36], Ceph [21], Harfbuzz [1], ICU [35], LibreOffice [30], LLVM [16], and qBittorrent [2]. We used the current `master` branch version of each project as available on GitHub on 7 August 2018, except for ICU, for which we used release 62.1.

*Duration and memory usage.* For our experiments we used a machine with Intel Xeon X5650 CPU @ 2.67 GHz and ran analyses on 12 threads. This however turned out to be less important, because the effects of our semantic additions are so negligibly small compared to the process of symbolic execution, that their impact on analysis duration and memory usage is undetectable.

*Number of bug reports.* In the above listed 7 widely used and well tested projects, our enhanced analysis found 3 new true positive errors, all following the basic format showed on Listing 8.

```
  const char *msg = std::to_string(name).c_str();
```

**Listing 8: Basic format of the container lifetime related errors found in popular open-source projects.**

In all cases, developers obtained a raw pointer pointing to the inner buffer of a temporary string, and later used that pointer even though the buffer was destroyed at the end of the full expression. The errors were found in Ceph, Facebook's RocksDB (which is Ceph's dependency), and GPGME (which is LibreOffice's dependency). All of them were reported to the respective communities and fixed within a day [24–26].

One of the greatest advantages of our work is that it produced zero false positive reports on the analyzed projects.

The number of real bugs not reported, i.e. the false negative rate, is harder to investigate. Before we began our work, we created a long test file containing all the cases we wished to cover after teaching the analyzer to understand `std::string` operations. We later "implanted" these errors into real projects to see if they are being recognized (they were, but we probably could not obscure them as much as they would be during real use). The only important cases not recognized to our knowledge are a raw pointer escaping into a global variable, and the usage of `std::string_views` in the same role as raw pointers.

## 5.2 Comparison to other static analysis tools

We are aware of many great static error detection tools for software written in C and C++. Some of the most well-known are Coverity [29], performing a form of "may" analysis; CodeSonar [14], performing interprocedural analysis; the Fortify Static Code Analyzer [17], for security issues; KlocWork [28], performing interprocedural data flow analysis; PolySpace [34], doing abstract interpretation; PREfast [18], using intraprocedural analysis and statistics; and PREfix [6], doing interprocedural data flow analysis, among others. The problem with these software products is that all of them are proprietary.

Other tools such as Splint [11] and CQual++ [9], which detect security-related bugs, require annotations present in the source code. Others, like ESP [10], Goanna [13], SLAM [3], and Blast [4], are model checkers, and need a description of the system to verify. Archer [38] is a symbolic analyzer specialized on C arrays, while KLEE [7] is a symbolic execution engine working on the intermediate representation level.

The bulk of these tools are either not open-source or designed for a different use case than the home of our developments, the Clang Static Analyzer. The Analyzer's philosophy is to be able to run a general-purpose, deep analysis on an industrial C, C++ or Objective-C project, without any formal preparations and desirably without the need to add annotations, within a reasonable time frame, and produce error reports that cover as many aspects of the C++ language as possible.

An open-source source code analysis tool for C/C++ following a similar philosophy that we could compare our work to is Cppcheck [8]. Cppcheck uses data flow analysis and regular expressions to identify erroneous code patterns. CppCheck was able to find the some STL related lifetime. It tries to avoid potential false positive results like the one on Listing 4, thus it only finds a small subset of the problems that could be detected without path- or flow-sensitivity. Our solution can find more problems while maintaining the same good false positive ratio.

## 6 FUTURE WORK

### 6.1 Understanding container semantics

One disadvantage of our current approach is that if we wish to extend the analyzer to understand the semantics of other containers, we need to add all that knowledge to the analyzer core, which is viable for STL types, but not for custom containers defined by users.

*Annotations.* In order to facilitate the addition of custom container API modeling, we need to extend the analyzer's infrastructure. One of our options that is already used throughout the compiler and the analyzer is adding *annotations* for this purpose. Annotations attached to function declarations appear in the abstract syntax tree and can be queried by the analyzer. Then, if users annotate methods of their containers that give access to the container's inner buffer, and methods that potentially corrupt the buffer, they allow the analyzer to apply the same reasoning to them as to STL types.

*API notes.* Another possible path that has been proposed earlier is a compiler feature called *API notes*, which already works in the Swift compiler, but has not been accepted into Clang yet. The idea behind API notes is that often it would ease the compilation and analysis of system header functions if they had some attributes attached to them (e.g. `noreturn` to the `exit()` function). API notes are separate YAML format files parsed during compilation that contain additional information of this kind about various functions. Thus, if this feature gets accepted into the compiler, the analyzer can also use API notes as an alternative to annotations.

*Type categories.* Perhaps the most convenient way of achieving the same goal would be to not do anything at all - the compiler would understand container lifetime semantics automatically. This is not as much beyond our reach as we might think. Ongoing work implementing Herb Sutter's lifetime analysis mentioned in Section 2 strives to bring automatic type category inference into the most popular C++ compilers in the next few years.

### 6.2 Views as pointers

A somewhat orthogonal improvement is adding support for `view`-like types e.g. `std::string_views`, `std::reference_wrappers`, and `std::spans`. These are alternatives to the built-in raw pointer types currently handled on a symbolic level. Just like containers, their methods are also not inlined by the analyzer to avoid confusing bug reports ending in STL header files, and thus even the simplest operations on them, e.g. assigning one `std::string_view` to another, are not understood. Adding a translation of `view` types to symbol aggregates, and modeling operations on them will be a huge engineering effort. Their handling is also important as a number of new `view`-like types are coming in the C++20 language standard, such as `function_refs` and `range`-related `views`.

## 7 CONCLUSION

Detecting object lifetime related errors in C++ source code is hard, because the internals of Standard Template Library types are hard for static analyzers to understand. We presented a method of adding knowledge of STL ownership semantics to a symbolic execution engine, which resulted in the analyzer finding new important bugs with no observable false positive reports. We showed the enhanced analysis fared in comparison to other open-source C++ analyzers and explored the possibilities for generalizing our approach to custom user-defined containers and `view`-like objects. We believe that our solution is a promising approach towards empowering static analyzers to understand C++ source code in depth and with all the complexity of STL containers.

## ACKNOWLEDGMENTS

The work presented in this paper has been funded by a Google Summer of Code program completed at the LLVM Foundation in 2018 [22]. The authors would like to express their gratitude towards members of the open-source community who added their insights and invested their time to review changes upstreamed to the Clang project: Artem Dergachev, George Karpenkov and Devin Coughlin.

The publication of this paper is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

## REFERENCES

[1] 2019. HarfBuzz, a text shaping engine. https://www.freedesktop.org/wiki/Software/HarfBuzz/ (last accessed: 21-04-2019).

[2] 2019. qBittorrent, a free and reliable P2P Bittorrent client. https://www.qbittorrent.org/ (last accessed: 21-04-2019).

[3] Thomas Ball and Sriram K Rajamani. 2002. The SLAM project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 1–3.

[4] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. 2005. Checking memory safety with Blast. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2–18.

[5] Nicolas Blanc, Alex Groce, and Daniel Kroening. 2007. Verifying C++ with STL containers via predicate abstraction. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 521–524.

[6] William R Bush, Jonathan D Pincus, and David J Sielaff. 2000. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience* 30, 7 (2000), 775–802.

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[8] Daniel Marjamaki. 2019. Cppcheck, a tool for static C/C++ analysis. http://cppcheck.sourceforge.net/ (last accessed: 21-04-2019).

[9] Daniel Wilkerson. 2019. CQual++. http://dsw.users.sonic.net/oink/qual.html (last accessed: 21-04-2019).

[10] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. *ACM Sigplan Notices* 37, 5 (2002), 57–68.

[11] David Evans. 2019. Splint, annotation-assisted lightweight static checking. http://splint.org/ (last accessed: 21-04-2019).

[12] Ericsson Hungary. 2019. CodeChecker, a static analysis infrastructure built on the LLVM/Clang Static Analyzer toolchain, replacing scan-build in a Linux or macOS (OS X) development environment. https://codechecker.readthedocs.io/en/latest/ (last accessed: 21-04-2019).

[13] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. 2006. GoannaâĂŤa static model checker. In *International Workshop on Parallel and Distributed Methods in Verification*. Springer, 297–300.

[14] GrammaTech Inc. 2019. CodeSonar. https://www.grammatech.com/products/codesonar (last accessed: 21-04-2019).

[15] Herb Sutter. 2019. Lifetime safety: Preventing common dangling. https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf (last accessed: 21-04-2019).

[16] LLVM Foundation. 2019. The LLVM Compiler Infrastructure Project. https://llvm.org/ (last accessed: 21-04-2019).

[17] Micro Focus, Ltd. 2019. Fortify Static Code Analyzer. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview (last accessed: 21-04-2019).

[18] Microsoft, Inc. 2019. PREfast, a static analysis tool that identifies defects in C/C++ programs. https://msdn.microsoft.com/en-us/windows/desktop/ms933794 (last accessed: 21-04-2019).

[19] Matt Miller. 2018. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/ (last accessed: 28-02-2019).

[20] Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. 2002. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 75–88.

[21] Red Hat, Inc. 2019. Ceph: a unified, distributed storage system. https://ceph.com/ (last accessed: 21-04-2019).

[22] Reka Nikolett Kovacs. 2018. A checker for dangling string pointers in C++. Google Summer of Code Archive, https://summerofcode.withgoogle.com/archive/2018/projects/6487597828276224/ (last accessed: 21-04-2019).

[23] Reka Nikolett Kovacs. 2019. Inner Pointer Checker in the Clang Static Analyzer. https://github.com/llvm-mirror/clang/blob/release_70/lib/StaticAnalyzer/Checkers/InnerPointerChecker.cpp (last accessed: 21-04-2019).

[24] Bug report. 2018. Use-after-free in db/db_impl_open.cc. https://github.com/facebook/rocksdb/issues/4239 (last accessed: 21-04-2019).

[25] Bug report. 2018. Use-after-free problem in gpggencardkeyinteractor.cpp. https://dev.gnupg.org/T4094 (last accessed: 21-04-2019).

[26] Bug report. 2018. Use-after-free problem in RocksDBStore.cc. https://marc.info/?l=ceph-devel&m=153367941406038&w=2 (last accessed: 21-04-2019).

[27] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[28] Rogue Wave Software, Inc. 2019. Klocwork. https://www.roguewave.com/products-services/klocwork (last accessed: 21-04-2019).

[29] Synopsys, Inc. 2019. Coverity. https://scan.coverity.com/ (last accessed: 21-04-2019).

[30] The Document Foundation. 2019. LibreOffice: free office suite. https://www.libreoffice.org/ (last accessed: 21-04-2019).

[31] The LLVM Foundation. 2019. Clang, a C language family frontend for LLVM. https://clang.llvm.org/ (last accessed: 21-04-2019).

[32] The LLVM Foundation. 2019. Clang Static Analyzer, a source code analysis tool that finds bugs in C, C++, and Objective-C programs. https://clang-analyzer.llvm.org/ (last accessed: 21-04-2019).

[33] The LLVM Foundation. 2019. Clang-Tidy. https://clang.llvm.org/extra/clang-tidy/ (last accessed: 21-04-2019).

[34] The MathWorks, Inc. 2019. Polyspace Bug Finder. https://www.mathworks.com/products/polyspace-bug-finder.html (last accessed: 21-04-2019).

[35] Unicode, Inc. 2019. ICU: International Components for Unicode. http://site.icu-project.org/home (last accessed: 21-04-2019).

[36] Wladimir van der Laan, Jonas Schnelli, Pieter Wuille. 2019. Bitcoin Core. https://bitcoincore.org/ (last accessed: 21-04-2019).

[37] Xusheng Xiao, Gogul Balakrishnan, Franjo Ivančić, Naoto Maeda, Aarti Gupta, and Deepak Chhetri. 2014. ARC++: effective typestate and lifetime dependency analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 116–126.

[38] Yichen Xie, Andy Chou, and Dawson Engler. 2003. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *ACM SIGSOFT Software Engineering Notes* 28, 5 (2003), 327–336.

[39] Zhongxing Xu, Ted Kremenek, and Jian Zhang. 2010. A Memory Model for Static Analysis of C Programs. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I (ISoLA'10)*. Springer-Verlag, Berlin, Heidelberg, 535–548. http://dl.acm.org/citation.cfm?id=1939281.1939332