Eötvös Loránd University

Faculty of Informatics

Dept. of Programming Languages and Compilers

Enhancing pattern matching-based static analysis of C-family software projects with project-level knowledge

Supervisor:
Richárd Szalay
Ph.D. student

Author:
Benedek Attila Bahrami
Computer Science B.Sc.

Budapest, 2022

This page should be the original Thesis Topic Declaration.

Contents

Chapter 1

Introduction

1.1 Motivation

There are programming languages, like `Pascal`, `Ada` or `BASIC`, that distinguishes functions that do and do not have return values, the latter are known as procedures. C family languages do not. Pre-standardisation C language did not have `void` functions, instead an unspecified return value defaulted to `int`. This resulted in functions declared with `int` return value not returning anything and the supposed return value was unused on purpose. One of its consequences was for example, that for a while if you wrote a function, that was declared to return `int` without the return value, the compiler did not act on it. When writing code in C++ we often use functions from C, and a lot of standard library POSIX function behaves as a C function.

1.2 Problem Statement

There are quite a few functions whose return value is often ignored, which could lead to potential bugs. Some examples:

- POSIX `read`: returns the number of files read; this return value can also indicate errors with it being -1.
- POSIX `scanf`: returns the number of items in the argument list successfully filled; also indicates errors with EOF return.
- (cstdio) `std::remove`: return indicates success or error.

- (algorithm) `std::remove`: Does not remove. It returns an iterator, and we still need to use erase for all elements after this iterator.
- `std::remove_if`: Same as remove.
- container specific `erase`: Returns an iterator to the next element after the removed.
- container specific `insert`: Returns an iterator to the first of the new elements inserted.

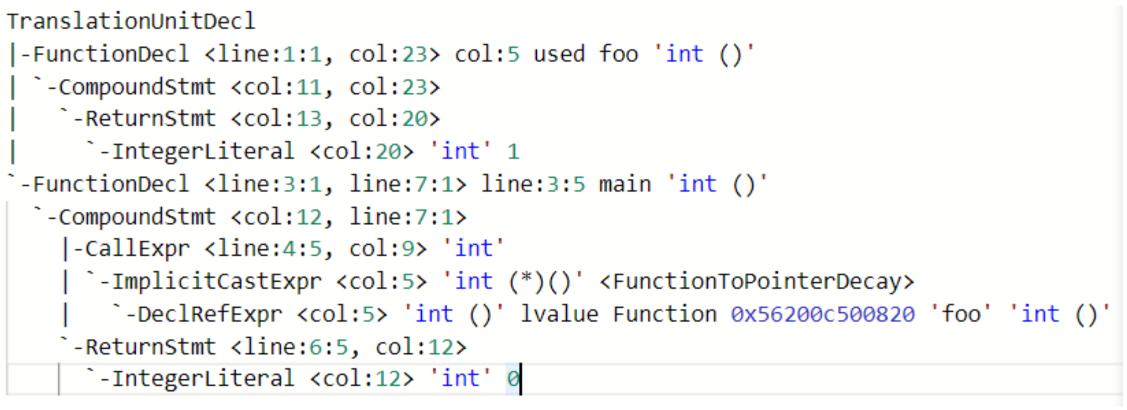
Later the attribute `[[nodiscard]]` [1] was introduced to notify and give warnings to the user if the return value was unused in case of a function with this attribute, but in ask the compiler to give warnings on unchecked values, we would need permission to modify the library code. In case of external source code such as POSIX, STL or any third party project, we will not have permission to do so. We still need to notify the user on the cases where they do not check non-void return value. This brings us back to static analysis.

1.3 Static analysis

Static analysis is a method to analyse the source code of software projects without performing a real execution of the application. It is widely used in industry to find bugs and code smells during development, to aid in the prevention of bad code that misbehaves in production. Among various methods, the most important techniques are the ones that are based on pattern matching on a syntactic representation of the software project.

Clang-Tidy is a declarative, object oriented, strong typed static analysis rule collection that is built upon the LLVM Compiler Infrastructure's C-family compiler, Clang. It performs pattern matching on Clang's "Abstract Syntax Tree" (AST) representation, and generating diagnostics based on which analysis modules, called "checks", the user turns on. We will address both the LLVM library and AST matchers in later chapters of the thesis.

Let us imagine a checker, that keeps a statistics on how many times a return value of a non-`void` function is used, or otherwise known as checked. This property is, as we previously stated important, because there exist a great amount of functions,



```
TranslationUnitDecl
|-FunctionDecl <line:1:1, col:23> col:5 used foo 'int ()'
| `-CompoundStmt <col:11, col:23>
|   `-ReturnStmt <col:13, col:20>
|     `-IntegerLiteral <col:20> 'int' 1
`-FunctionDecl <line:3:1, line:7:1> line:3:5 main 'int ()'
  `-CompoundStmt <col:12, line:7:1>
    |-CallExpr <line:4:5, col:9> 'int'
    | `-ImplicitCastExpr <col:5> 'int (*)()' <FunctionToPointerDecay>
    |   `-DeclRefExpr <col:5> 'int ()' lvalue Function 0x56200c500820 'foo' 'int ()'
    `-ReturnStmt <line:6:5, col:12>
      `-IntegerLiteral <col:12> 'int' 0
```

Figure 1.1: An example of an Abstract Syntax Tree

whose return value should be checked in most situations but remain unchecked in quite a lot.

1.4 Infrastructure Limitations

Unfortunately, for programming languages in the C family, such as C++, the concept of "separate translation" causes issues for static analysis. As most static analysers are built upon compilers, and in C++, each compiler only sees the local information in the source file (also known as the Translation Unit) it is to compile or analyse (as opposed to project-level knowledge), crucial details might be hidden, which lowers, or in most cases, completely distorts the accuracy of the analysis. This means that the way our infrastructure works, a checker like this would be of very limited use.

A function can obviously exist outside of the translation unit of its declaration. If such analysis with our imagined checker is done separately on each translation unit, it is easy to see how that can affect the outcome. A function might be called 100 times and checked 95 times through. We would want to give warnings for the unchecked 5% of those calls, but if, for example, these are in a separate translation unit, then the analysis would return with 0 checks out of 5 calls. We would not want to give warnings to a function that is unchecked in all 100% of its calls. The detail of the statistics, that it was only unchecked in 5% is lost, unless we use project level knowledge during our analysis. Consider another example, with code:

```
// First translation unit
std::vector<int> c{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 15};

```

```cpp
auto iter = c.erase(c.begin());
iter = c.erase(iter + 2, iter + 5);

for (std::vector<int>::iterator it = c.begin(); it != c.end(); ) {
    if (*it % 2 == 0) {
        it = c.erase(it);
    } else {
        ++it;
    }
}

// Second translation unit

std::vector<int> c{0, 1, 2, 3, 4};
for (std::vector<int>::iterator it = c.begin(); it != c.end(); ++it
  ) {
    if (*it % 2 == 0) {
        c.erase(it);
    }
}
```

Code 1.1: An example of the infrastructure's limitations

Here, with separate analysis, we would get 100% checked in the first TU and 0% in the second one. With project level knowledge, however we would get 57%. We used erase without a care towards its return value, and this could lead to potential bugs, but without the project level knowledge, however we can not diagnose it.

The separate analysis could also create false positive results. Imagine we have a function whose return value we could use but it is mostly optional. We could give our checker a treshold of percentage, to only diagnose unused values if we usually use them in most cases. Again, this means that with different translation units, we do not know how many times we actually ignored the value and can not use our treshold properly. This leads to false positive diagnosis.

Unfortunately, the current versions of Clang-Tidy checks can only access what is visible to the compiler, which is a local information. Several classes of security issues and bad coding patterns might be diagnosed if the implemented checks would be capable of creating percompilation knowledge, and reusing the full knowledge about the project during diagnosis. The work of the thesis is to enhance Clang-Tidy on the infrastructure level to support multi-pass analyses in a generic manner, by utilizing the ideas similar to that of MapReduce [2].

This is achieved by allowing individual checks to store check-specific data on a thread-safe location. A subsequent execution of the analysis will be able to do the pattern matching fine-tuned with the data stored in the previous step also available. To prove the usability of the solution, a new safety and security related check, currently not provided by Clang-Tidy, will be developed utilizing the new infrastructure created in this work. In the end, the results of the thesis will allow the international community behind LLVM to develop and make available a wider potential of checks, as we are planning to upstream the entire project into LLVM. [3] [4] [5]

1.5 Related Works

The topic of cross translation unit analysis shares other different solutions and implementations as well, one great example is Clang CTU [6]. As previously stated, the infrastructure enhancement follows the philosophy of MapReduce [2]. There exists a static analysis rule in Coverity Scan static analysis tool, that is similar to our checker [7].

1.6 Thesis layout

After the Introduction, in chapter 2, the User Guide will have instructions on how to download, compile and set up the static analysis tool, and how to run it on a C++ project. In chapter 3, the Developer Guide will explain how the checker, and the infrastructure itself works, in detail.

Chapter 2

User documentation

Both the changes in the Clang-Tidy infrastructure and our new checker obviously focuses heavily on LLVM's Clang-Tidy. Clang-Tidy is a clang-based C++ "linter" tool. Its purpose is to provide an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis. Clang-Tidy is modular and provides a convenient interface for writing new checks.

This tool can be found in the LLVM project repository.

2.1 Install guide

2.1.1 System Requirements

Table 2.1 shows the system requirements and supported compilers for building LLVM. The checkers were developed with Ubuntu 20.04 and tested on Ubuntu 18.04, and WSL Ubuntu 20.04.

Building and using LLVM's Clang-Tidy takes a lot of time on weaker computers. The minimum recommended memory size for building is 16 GB, the optimal amount is 64 GB of memory.

Operating System	Processor Architecture	Compiler
Linux	x861	gcc, clang
Linux	amd64	gcc, clang
Linux	arm	gcc, clang
Linux	Mips	gcc, clang
Linux	PowerPC	gcc, clang
Solaris	V9	gcc
FreeBSD	x861	gcc, clang
FreeBSD	amd64	gcc, clang
NetBSD	x861	gcc, clang
NetBSD	amd64	gcc, clang
macOS2	PowerPC	gcc
macOS	x86	gcc, clang
Cygwin	x86	gcc
Windows	x86	Visual Studio
Windows64	x86-64	Visual Studio

Table 2.1: System requirements and supported compilers for building LLVM

Software requirements include (at least) GCC version 7.1.0, CMake version 3.13.4, Python version 3.6 and GNU Make version 3.79.

2.1.2 Building from source

These commands will compile LLVM from source. The building process with parameters can be found on the README.md of LLVM project Github repository[1], or the Getting Started[2] page of Clang documentation. These are the commands I used for the compilation.

```
# On windows, git clone --config core.autocrlf=false https://
    github.com/llvm/llvm-project.git
git clone https://github.com/llvm/llvm-project.git
cd llvm-project
mkdir Build

# cmake -S llvm -B build -G <generator> [options]
cd Build/
cmake \
  -DCMAKE_EXPORT_COMPILE_COMMANDS=ON \
  -DLLVM_ENABLE_PROJECTS="llvm;clang;clang-tools-extra" \
  -DLLVM_TARGETS_TO_BUILD="X86" \
```

[1]https://github.com/llvm/llvm-project#readme
[2]https://clang.llvm.org/get$_s$*tarted.html*

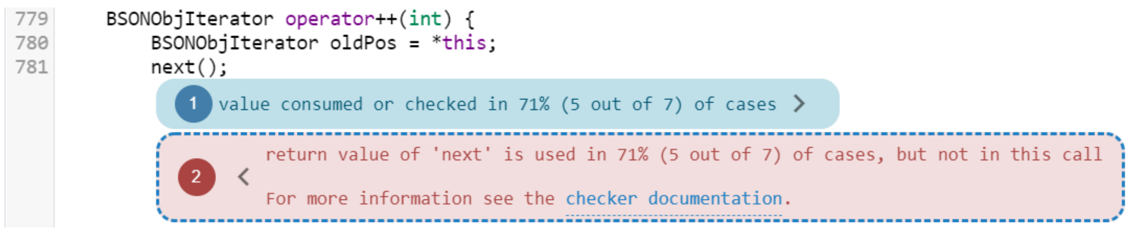
```
    -DLLVM_APPEND_VC_REV=OFF \
    -DLLVM_ENABLE_BINDINGS=OFF \
    -DLLVM_USE_RELATIVE_PATHS_IN_FILES=OFF \
    -DBUILD_SHARED_LIBS=ON \
    -DLLVM_USE_LINKER="lld" \
    -DLLVM_PARALLEL_LINK_JOBS=3 \
    -DCMAKE_BUILD_TYPE=Release \
    -DLLVM_ENABLE_DUMP=ON \
    -DLLVM_ENABLE_ASSERTIONS=ON \
    -G Ninja \
    ../llvm

# cmake --build build [-- [options] <target>] or your build
    system specified above directly.
ninja -j12 clang-tidy llvm-symbolizer
```

Explanation for some flags: at `DLLVM_USE_LINKER` we can change the linker we are using, either Gold or LLD (Linker for LLVM). The latter needs to be installed. `DLLVM_PARALLEL_LINK_JOBS` and -j at Ninja sets the CPU capacity. The recommended amount for `LINK_JOBS` is one quarter of the amount of cores, and Ninja job amount should be cores - 2. I used these commands on a server with 32 GB memory and 14 CPU cores.

2.2 Running by Translation Units

You can give Clang-Tidy multiple translation units to run on, and it will give you diagnosis separately for each one. You run it by using `clang-tidy -checks='-*,misc-discarded-return-value' -p ./Build a/main.cpp b/main.cpp`, where the "checks" first disables all checkers with -*, then enables our checker, the flag "p" gets the build path and finally we give the paths to our code. Here we are getting two separate diagnoses for our two separate files or translation units.

```
/home/bahramib/MyFolder/TestFolder/a/main.cpp:12:9: \
warning: return value of 'maybe_check_this' is used \
in most calls, but not in this one [misc-discarded-return-value]
    MyClass::maybe_check_this();
    ^
/home/bahramib/MyFolder/TestFolder/a/main.cpp:12:9: note: \
value consumed or checked in 75% (3 out of 4) of cases
```



```
BSONObjIterator operator++(int) {
    BSONObjIterator oldPos = *this;
    next();
```

1 value consumed or checked in 71% (5 out of 7) of cases

2 return value of 'next' is used in 71% (5 out of 7) of cases, but not in this call

For more information see the checker documentation.

Figure 2.1: A report of the diagnosis of MongoDB's source on CodeChecker with 50% treshold on a single TU

Code 2.1: Diagnosis output without project level knowledge

Clang-Tidy is supported by CodeChecker. [8]

2.3 Multiple Phase Version

The updated infrastructure contains two new flags for running Clang-Tidy, multipass-phase and multipass-dir. Multipass-phase is an enum flag, that has three values, "collect", "compact" and "diagnose" with the latter as default. Multipass-dir needs a path to a directory where the checkers that support the collect feature can dump their collection datas that they are going to compact and use later.

2.3.1 Collect

Collect phase, as the name suggest, will have the checkers collect data on each translation unit and write them into unique yaml files to later reuse this data. This is how you normally run collect phase on the desired files:

```
clang-tidy \
-checks='-*,misc-discarded-return-value' \
--multipass-phase=collect \
--multipass-dir='MyCollectionDirectory' \
-p ./Build \
a/main.cpp b/main.cpp
```

What Discarded Return Value checker (or DRV) does in this phase, is count the amount the declared non-void functions were called, and count the amount that these function's return values were checked. After finishing counting in one translation unit, it writes the collected numbers and function names into a yaml file as a struct.

After collecting, you do not get any diagnosis or output text, but the desired amount of (in this case 2) yaml files are going to be generated.

```
bahramib@cc:~/MyFolder/TestFolder/MyCollectionDirectory$ ll
total 8
-rw-r--r-- 1 bahramib bahramib 196 Apr 30 13:00 misc-discarded-
    return-value.main.cpp.12949585208029997868.yaml
-rw-r--r-- 1 bahramib bahramib 196 Apr 30 13:00 misc-discarded-
    return-value.main.cpp.4924802982073527590.yaml
```

Code 2.2: The yaml files containing the collection data

```
  # First TU
  ---
  - ID:                 'c:@S@MyClass@F@check_that#S'
    Consumed:           3
    Total:              3
  - ID:                 'c:@S@MyClass@F@maybe_check_this#S'
    Consumed:           0
    Total:              3
  ...
  # Second TU
  ---
  - ID:                 'c:@S@MyClass@F@check_that#S'
  Consumed:           1
  Total:              3
  - ID:                 'c:@S@MyClass@F@maybe_check_this#S'
  Consumed:           3
  Total:              4
  ...
```

Code 2.3: Contents of the collection files

2.3.2 Compact

Compact phase will iterate through the collected data per checker and have the checkers read, use and compact all the data collected into one yaml file. Flags aside from checks, multipass-phase and multipass-dir have no effect.

```
  clang-tidy \
  -checks='-*,misc-discarded-return-value' \
  --multipass-phase=compact \
--multipass-dir='MyCollectionDirectory' \
```

DRV reads the data on each function back and constructs new data similar to the previous ones. If one function is called in multiple translation units, then it simply adds the numbers from the TU's yaml file to the new structure. After it is finished, the data is written into a single yaml file.

This phase does not write anything on standard output either, but will construct the compacted yamls per checker.

```
bahramib@cc:~/MyFolder/TestFolder/MyCollectionDirectory$ ll
total 12
-rw-r--r-- 1 bahramib bahramib 196 Apr 30 13:00 misc-discarded-
    return-value.main.cpp.12949585208029997868.yaml
-rw-r--r-- 1 bahramib bahramib 196 Apr 30 13:00 misc-discarded-
    return-value.main.cpp.4924802982073527590.yaml
-rw-r--r-- 1 bahramib bahramib 196 Apr 30 13:06 misc-discarded-
    return-value.yaml
```

Code 2.4: The new file containing the collected data

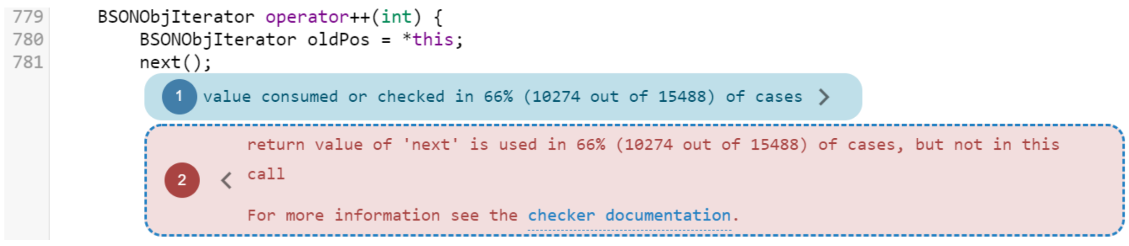
```
---
- ID:                'c:@S@MyClass@F@check_that#S'
Consumed:          4
Total:             6
- ID:                'c:@S@MyClass@F@maybe_check_this#S'
Consumed:          3
Total:             7
...
```

Code 2.5: Contents of the compacted file

2.3.3 Diagnose

For backwards compatibility, the default diagnose phase, will do exactly what the non-multiple phase Clang-Tidy did, give diagnoses for each translation unit separately, as demonstrated in section section 2.2, if compact has not happened for a checker. Otherwise that checker will read and use the data compacted in the respective yaml file, and give its diagnosis calculated with project level knowledge.

```
clang-tidy \
-checks='-*,misc-discarded-return-value' \
```



```
BSONObjIterator operator++(int) {
    BSONObjIterator oldPos = *this;
    next();
```

1 value consumed or checked in 66% (10274 out of 15488) of cases

2 return value of 'next' is used in 66% (10274 out of 15488) of cases, but not in this call

For more information see the checker documentation.

Figure 2.2: A report of the diagnosis of MongoDB's source on CodeChecker with 50% treshold and project level knowledge

```
--multipass-phase=diagnose \
--multipass-dir='MyCollectionDirectory' \
-p ./Build \
a/main.cpp b/main.cpp
```

DRV reads in the compacted data with the call and check amounts for each function and simply desides if diagnosis is needed or not for each unchecked return value in the current translation unit. The output is obviously the diagnosis.

```
/home/bahramib/MyFolder/TestFolder/a/main.cpp:12:9: \
warning: return value of 'check_that' is used \
in most calls, but noti in this one [misc-discarded-return-value]
MyClass::check_that();
  ^
/home/bahramib/MyFolder/TestFolder/a/main.cpp:12:9: note: \
value consumed or checked in 66% (4 out of 6) of cases

/home/bahramib/MyFolder/TestFolder/a/main.cpp:15:5: \
warning: return value of 'check_that' is used \
in most calls, but not in this one [misc-discarded-return-value]
MyClass::check_that();
^
/home/bahramib/MyFolder/TestFolder/a/main.cpp:15:5: note: \
value consumed or checked in 66% (4 out of 6) of cases
```

Code 2.6: Diagnosis output with project level knowledge

We can clearly conclude, that the multipass diagnosis resulted in different warnings. Figure section 2.3.3 differs from section 2.2 in only the percentage, but with a different treshold that could lead to different results, to either false positives, or hidden true positives. Our own example, however shows both, since we had a call that used to give warning when diagnosed separately which with project level knowledge does not, and one that did not give any warnings, but now it gives two, because it passed the treshold.

In listing 1.1 we talked about our code getting no warnings without project level knowledge. After phases collect and compact, however our diagnosis will result in the following:

```
  /home/bahramib/MyFolder/TestFolder/b/vec.cpp:7:15: \
  warning: return value of 'erase' is used \
  in most calls, but not in this one [misc-discarded-return-value]
            c.erase(it);
              ^
/home/bahramib/MyFolder/TestFolder/b/vec.cpp:7:15: note: \
value consumed or checked in 66% (2 out of 3) of cases
```

We can clearly see, that our diagnosis gave us the real results this time.

Chapter 3

Developer documentation

This chapter will cover the process of developing a Clang-Tidy checker, and the infrastructure that it is build on. The developer should have an advanced knowledge of C++ and strong typing, minimal user experience with Linux and Clang-Tidy, and familiarity with declarative programming paradigm.

3.1 Infrastructure

3.2 Discarded Return Value Check

3.3 Evaluation

Chapter 4

Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In eu egestas mauris. Quisque nisl elit, varius in erat eu, dictum commodo lorem. Sed commodo libero et sem laoreet consectetur. Fusce ligula arcu, vestibulum et sodales vel, venenatis at velit. Aliquam erat volutpat. Proin condimentum accumsan velit id hendrerit. Cras egestas arcu quis felis placerat, ut sodales velit malesuada. Maecenas et turpis eu turpis placerat euismod. Maecenas a urna viverra, scelerisque nibh ut, malesuada ex.

Aliquam suscipit dignissim tempor. Praesent tortor libero, feugiat et tellus porttitor, malesuada eleifend felis. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nullam eleifend imperdiet lorem, sit amet imperdiet metus pellentesque vitae. Donec nec ligula urna. Aliquam bibendum tempor diam, sed lacinia eros dapibus id. Donec sed vehicula turpis. Aliquam hendrerit sed nulla vitae convallis. Etiam libero quam, pharetra ac est nec, sodales placerat augue. Praesent eu consequat purus.

Appendix A

Simulation results

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque facilisis in nibh auctor molestie. Donec porta tortor mauris. Cras in lacus in purus ultricies blandit. Proin dolor erat, pulvinar posuere orci ac, eleifend ultrices libero. Donec elementum et elit a ullamcorper. Nunc tincidunt, lorem et consectetur tincidunt, ante sapien scelerisque neque, eu bibendum felis augue non est. Maecenas nibh arcu, ultrices et libero id, egestas tempus mauris. Etiam iaculis dui nec augue venenatis, fermentum posuere justo congue. Nullam sit amet porttitor sem, at porttitor augue. Proin bibendum justo at ornare efficitur. Donec tempor turpis ligula, vitae viverra felis finibus eu. Curabitur sed libero ac urna condimentum gravida. Donec tincidunt neque sit amet neque luctus auctor vel eget tortor. Integer dignissim, urna ut lobortis volutpat, justo nunc convallis diam, sit amet vulputate erat eros eu velit. Mauris porttitor dictum ante, commodo facilisis ex suscipit sed.

Sed egestas dapibus nisl, vitae fringilla justo. Donec eget condimentum lectus, molestie mattis nunc. Nulla ac faucibus dui. Nullam a congue erat. Ut accumsan sed sapien quis porttitor. Ut pellentesque, est ac posuere pulvinar, tortor mauris fermentum nulla, sit amet fringilla sapien sapien quis velit. Integer accumsan placerat lorem, eu aliquam urna consectetur eget. In ligula orci, dignissim sed consequat ac, porta at metus. Phasellus ipsum tellus, molestie ut lacus tempus, rutrum convallis elit. Suspendisse arcu orci, luctus vitae ultricies quis, bibendum sed elit. Vivamus at sem maximus leo placerat gravida semper vel mi. Etiam hendrerit sed massa ut lacinia. Morbi varius libero odio, sit amet auctor nunc interdum sit amet.

Aenean non mauris accumsan, rutrum nisi non, porttitor enim. Maecenas vel tortor ex. Proin vulputate tellus luctus egestas fermentum. In nec lobortis risus, sit amet tincidunt purus. Nam id turpis venenatis, vehicula nisl sed, ultricies nibh. Suspendisse in libero nec nisi tempor vestibulum. Integer eu dui congue enim venenatis lobortis. Donec sed elementum nunc. Nulla facilisi. Maecenas cursus id lorem et finibus. Sed fermentum molestie erat, nec tempor lorem facilisis cursus. In vel nulla id orci fringilla facilisis. Cras non bibendum odio, ac vestibulum ex. Donec turpis urna, tincidunt ut mi eu, finibus facilisis lorem. Praesent posuere nisl nec dui accumsan, sed interdum odio malesuada.

Bibliography

[1] *C++ attribute: nodiscard (since C++17).* URL: `https://en.cppreference.com/w/cpp/language/attributes/nodiscard`.

[2] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation.* San Francisco, CA, 2004, pp. 137–150.

[3] Benedek Attila Bahrami. *[clang-tidy] Add infrastructure support for running on project-level information.* Revision on Phabricator for review. 2022. URL: `https://reviews.llvm.org/D124447`.

[4] Benedek Attila Bahrami. *[clang-tidy] Add the misc-discarded-return-value check.* Revision on Phabricator for review. 2022. URL: `https://reviews.llvm.org/D124446`.

[5] Benedek Attila Bahrami. *[clang-tidy] Add project-level analysis support to misc-discarded-return-value.* Revision on Phabricator for review. 2022. URL: `https://reviews.llvm.org/D124448`.

[6] *Cross Translation Unit (CTU) Analysis.* URL: `https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html`.

[7] *Coverity Scan - Static Analysis.* URL: `https://scan.coverity.com/o/oss_success_stories/69`.

[8] *CodeChecker.* URL: `https://github.com/Ericsson/codechecker`.

List of Figures

List of Tables

List of Algorithms

List of Codes