Linköping University | Department of Computer and Information Science
Bachelor Thesis, 16hp | Programming
Spring term 2016 | LIU-IDA/LITH-EX-G--16/011--SE

Static Code Analysis of C++ in LLVM

Olle Kvarnström

Supervisor, Jonas Wallgren
Examiner, Klas Arvidsson

Static Code Analysis of C++ in LLVM

Olle Kvarnström
IDA, Linköping University
olle@iix.se

ABSTRACT
Just like the release of the Clang compiler, the advent of LLVM in the field of static code analysis already shows great promise. When given the task of covering rules not ideally covered by a commercial contender, the end result is not only overwhelmingly positive, the implementation time is only a fraction of what was initially expected. While LLVM's support for sophisticated AST analysis is remarkable, being the main reason these positive results, it's support for data flow analysis is not yet up to par. Despite this, as well as a lack of thorough documentation, LLVM should already be a strong rival for any commercial tool today.

INTRODUCTION

Motivation

Developing software for airborne systems is a delicate process, where even the smallest flaws may lead to death, severe damage to property, or harm to the environment. This defines airborne systems as being *safety-critical* [1].

Working in an environment with such high stakes requires a great deal of care to be taken. Both the source code and resulting programs must be thoroughly analyzed and tested to iron out any minor inconsistencies. Analyzing code can be done either dynamically or statically.

Dynamic code analysis means that you let the code execute, usually in a controlled environment. The execution is carefully studied, and used to confirm that the program behaves as expected. This type of analysis can be difficult to apply in a safety-critical system with limited resources.

Static code analysis, on the other hand, reads the program's source code and reasons how the running application will behave. This means that the code can be analyzed without access to the safety-critical system, and without any risks involved. Since access to the system might be limited, this can also save the developers' time.

Saab is a Swedish defense and security company, commonly known for its aircraft fighter JAS 39 Gripen. In its continuous ambition to produce cutting-edge technology, static code analysis is a natural part of Saab's software development process. To ensure that their software is not only safe, but also easy to maintain and verify, it is developed according to strict code standards. These standards can have hundreds of rules, which are oftentimes much stricter than the underlying program language. Assuring compliance with these rules manually would require great time and effort, and the work is therefore more suitably done with an automatic tool. However, since these rules are custom tailored to suit the needs of Saab, there is a limited set of tools available. At the moment, Saab strives to enforce a higher degree of strictness than contemporary tools can provide.

Creating a new tool for static code analysis from scratch would be a major undertaking, requiring a lot of time and effort. However, a new actor in the open source community might provide a solution.

LLVM, formerly an acronym for Low Level Virtual Machine, is a compiler infrastructure, which originates from a thesis on aggressive multi-stage optimizing compiler technology [2]. The design of the LLVM internal code representation would allow for sophisticated C and C++ code analysis [3].

Objective

This paper examines the use of LLVM in static code analysis. The end goal is evaluating if LLVM is suitable for working in conjunction with, or even replacing, using a commercial tool. To do this, we focus on covering rules not correctly supported by the current tools at Saab. These tools cover the rules either too strictly, which emits *false positives*, too leniently, which causes *false negatives*, or not at all.

It is important to have a 1:1 relationship between issues and error message. False positives is a major issue in static analysis today, and is the main contributing reason for static analysis tools not being used in a wider scale [4, 5]. False negatives are also problematic, since they force the end user to do additional manual checks.

In order to implement custom rule checkers in LLVM, a prototype tool for analyzing C++ source code is created. This prototype runs through a Continuous Integration(CI)-tool as part of the build process. It is also desirable for the prototype to be usable by the developers, either by running it manually, or through their Integrated Development Environment(IDE). The rules to cover are the following:

1. No operations are allowed to make machine or compiler dependent presumptions on a type's bit representation.
2. Variable and function names are not to be reused in nested scopes, or from base classes.
3. Pointers shall not be used for iterating over arrays for reading or writing.

4. Nothing should be defined in the global scope, except for in header files.
5. The assignment operator in a class must return a const reference to *this*, or void.
6. Pointers to functions must have a typedefined type.
7. Inside a function, all definitions must happen before any executable code. Exception: Defining variables in a for-loop's initializer is always permitted.
8. When assigning or comparing a pointer to NULL, the NULL must be type casted to the same type.
9. *#define* shall not be used for numeric constants.
10. Bitwise operations on Booleans are not allowed.
11. No function parameters shall have side effects.
12. No class, struct or enum declaration shall happen at the time of its type definition.
13. The keywords register, auto, export and mutable are not to be used.
14. The operators *=, /=, <<=, >>=, &=, ^=, %= and |= are not to be used.
15. Pointers to functions are not to be used.
16. The standard types are not to be used. Exceptions: void and char.
17. Free standing block statements delimited by {} are not permitted.

Research Questions

- Can LLVM be used to cover the rules described above, without emitting false positives or false negatives?
- How long does it take to write a rule check in LLVM?
- How well suited is LLVM for static analysis?

Limitations

To avoid unnecessary complexity, this project limits itself to prove compliance in correctly written, compiling C++.

THEORY

Static code analysis has existed since the 70s, when a program called Lint was released for the C programming language. Lint enforced type rules and portability restrictions more strictly than the compilers of that time. It also attempted to detect wasteful or error prone constructions [6]. However, since Lint reported all suspicious code, the high amount of warnings generated by it made it hard for developers to use. As a result, Lint was never widely adopted [5].

The second generation of analyzers began appearing in the early 2000. These tools had the additional functionality of being able to perform analysis in the code base as a whole, instead of just separate files or functions [7]. This was done by creating an abstract representation of the underlying source code and analyzing the program flow [5].

Today, several kinds of methods for static analysis exist. The common denominator is how they transform the source code into a mathematical model. Through this model, the code is then analyzed for inconsistencies. Below are a few common methods.

Abstract syntax tree

An abstract syntax tree(AST) is an attempt to describe the structural essence of the underlying programming language [8]. Each node in the tree represents a language construct, where the trunk of the tree contains operators, and the leaf nodes are variables. By comparing the nodes' relation to its neighbors, one can detect local errors such as type differences. The strength of analyzing the AST, is in its efficiency when it comes to assessing well-defined, structural rules [9]. AST-detection has previously been successfully applied in finding cases of code reuse in a large code base [10]. This was done by checking for duplicates, or near duplicates, of parts of the AST. AST-detection is however limited, and more advanced methods are needed for more sophisticated checks.

Data flow analysis

Data flow analysis separates the code into blocks, which are sequences of instructions with one entry point and one exit point. Variables within these blocks are analyzed on how they will behave, depending on which paths the code execution will take [11]. For example, when reaching an if-statement, the program will investigate each possible route separately, and afterwards compile a set of all the variables' possible values. This set will be used in future instructions, to ensure that no matter which route the code takes, the variable data is correct.

The strength of using data flow analysis is its thoroughness in which it tests all possible, even how unlikely, ways the code execution may travel. Error messages created by a data flow analysis tools can also contain exact information on how to reproduce said error. However, it is a very time expensive task, and as projects grow, this kind of analysis can quickly become untenable [12].

Model checking

Model checking has a more theoretical approach. It is done by constructing an abstract model of the given system, and then formally proving its correctness. This results in a very thorough analysis of the code, with linear execution time [13]. However, the process requires a great deal of manual effort, since the model must partly be constructed by hand. This effort can take considerably more time than the original source code took to write. It is also vulnerable to errors which, due to human factor, may appear in the source code but not in the model [14].

Abstract interpretation

Abstract interpretation starts by making an abstract model of the source code. Where other models contain exact information of the data, this model gains its efficiency from being selectively imprecise. Oftentimes, there is no need to know the exact value of a variable, but instead, the maximum and minimum possible values are stored. This range is then checked against given rules, instead of all the variable's

possible values. This can be of tremendous help, as it greatly reduces complexity in large projects [15].

In a study done by Bruno Blanchet, et al., abstract interpretation was used to statically analyze a large amount of safety-critical software code [15]. Their goal was to guarantee that no undefined behavior was executed, such as dividing by zero, or accessing memory past the stack. They also focused intensely on reducing the amount of false positives. By fine tuning the precision of the analyzer, they ended up with a, by their account, high precision rate, reasonable computation time and low number of false positives.

In the end, abstract interpretation gives you the possibility to trade precision for speed.

LLVM

Due to its origins as an optimizing compiler, LLVM's Internal Representation(IR) is very low level, Static Single Assignment(SSA) based and looks similar to RISC architecture. This makes it a suitable target for all source languages [2]. It also means that in order to do AST analysis, one would need to gather this information from the compiler before its AST-to-LLVM-IR transformation. This could be done though Clang, which has been developed by the same group, as an LLVM frontend [16, 17]. Clang also has support for data flow analysis, which is considered work-in-progress [18].

While there is no built-in support for model checking or abstract interpretation, third party projects have successful implemented both. An example of model checking in LLVM is LLBMC [19], and abstract interpretation has has been done in PAGAI [20].

Clang

There are three ways to interface with Clang. Depending on the interface chosen, both the implementation process and the end result is different.

LibClang

LibClang is a high-level C Application Programming Interface(API), and is the recommended way to interface Clang. The API is kept stable, so future changes to Clang will not break existing functionality. Since LibClang exposes a C API, it is easy, and recommended, to extend with Python, which can help development speed. However, the exposed functionality is limited, and might not provide enough flexibility to cover all the rules. This also removes the possibility of exposing the checks to other tools, such as an IDE, since the checks do not become a part of the compiled Clang library.

Clang Plugins

Clang Plugins are libraries, which are dynamically loaded through execution flags. This makes it easy to create many checks which can be selected and deselected at will. Since the plugins are compiled separately, they can be used by anyone with a local installation of Clang. As opposed to LibClang, plugins give full access to AST information, and can be exposed to, for example, an IDE. Using plugins works well with CI-tools, or a build process where developers already use Clang for compilation. A drawback with using Clang plugins, is that they are closely knit with Clang's internal structure, and may need manual updating to new Clang versions.

LibTooling

LibTooling is similar to Clang Plugins, but instead of creating plugins, it is used for creating stand-alone tools. Since these tools still need the LLVM library, they are comparable to Clang plugins with separate binary files. Thus, LibTooling is mostly suited for creating tools not part of the build process.

METHOD

As mentioned in the introduction, the prototype's use case is mainly through a CI-tool. If possible, the prototype tool should also be usable by the developers themselves. Taking this into consideration, Clang plugins seem to offer the best overall solution. By combining Clang plugins with the use of a CI-tool, the potential issue of the plugins not being compatible with new versions should also be offset. The reason for this is that CI-tools typically support running several concurrent versions of the same software. In the case of a new version of Clang not supporting older tests, the CI-tool can be set to run an older version of Clang when executing legacy tests, while using the newer version for newer tests. This makes it possible to always use the latest version of Clang for development, while retaining compatibility with anything previously written.

A short pre-study of differences between recent Clang versions shows that internal API changes are usually minor. For example, Clang 3.5.0 changes raw *T** pointers to the use of *std::unique_ptr<T>*, and Clang 3.8.0 changes the name of some of the AST-nodes. This means that porting the plugins should be a minor task, which may even make the above use of concurrent Clang versions redundant.

In the case of developers having access to Clang, Clang plugins will also provide them with the possibility of running the checks themselves. Since Clang plugins are loaded at runtime, they can be used by any person with standard Clang installation just by adding the necessary execution flags. This also makes it possible to include them into popular IDEs supported by Clang, such as Xcode and Eclipse.

For our prototype, each rule is implemented as its own plugin, which are then linked together to form a common shared library. Depending on how Clang uses plugins, this may affect execution time negatively compared to creating a single plugin containing all rules. However, as this approach makes the project highly modular with no dependencies between the different rules, its strengths outweigh the weaknesses. In the case of execution speed being significantly slowed down, this solution may need changing.

Implementation

Since the rules are originally written in plain language, they must first be rewritten to remove as much ambiguity as possible on what to cover. If a rule is considered too complex to write as a single rule, it is instead converted into a set of smaller rules, which are then easier to analyze.

When a rule has been rewritten, basic unit tests are created to cover different parts of the rule. This is done with the help of Saab. Since they have previously done thorough research on the rules and their potential edge-cases, most tests are based on their tests suites. This part also works as a validation that my interpretation of a rule is correct. In the event of my interpretation not matching the provided test suites, the interpretation is tweaked until they match.

After the tests cover every conceivable use case, the checker is implemented. There are many possible ways to implement the checker, of which there are two preferred options described below. Only when option A does not provide a solution of sufficient quality is option B attempted. If neither option A nor B offer an acceptable solution, the next attempted method will be depending on the issues faced.

Option A: AST-analysis

Because of its clear cut structure, AST-analysis can often make it easy to narrow down small parts of the code to analyze. Not only does this make it the least complex of the mentioned methods, it also means less risk of emitting false positives, which happens more frequently the more data you have to take into account. In Clang, AST-analysis can be done either through *Recursive AST Visitors* or *AST Matchers*. These are two different interfaces to the same underlying data. *AST Matchers* use a Domain Specific Language(DSL) to query the AST, which results in a short and concise program. A *Recursive AST Visitor* performs a more controlled walk through the tree and executes function calls based on the node types visited. Because walking the AST manually provides better support for statefulness, which will be described below, using *Recursive AST Visitors* is deemed the better option.

The *Recursive AST Visitor* works by doing a pre-order depth-first traversal on the AST. See Figure 1 for clarification. In order to operate on the nodes, a new class which derives from the *RecursiveASTVisitor* class is created. When added to the new class, visitor methods are automatically executed when a given node is encountered. For example, to prohibit the use of C-style casts, one adds a method with the name *VisitCStyleCastExpr*, containing a relevant error message. By feeding your class the AST generated by Clang, any use of C-style casts will now emit this error message. There are also pre-visit and post-visit methods available, which help statefulness, such as keeping track of which function that is currently begin evaluated.

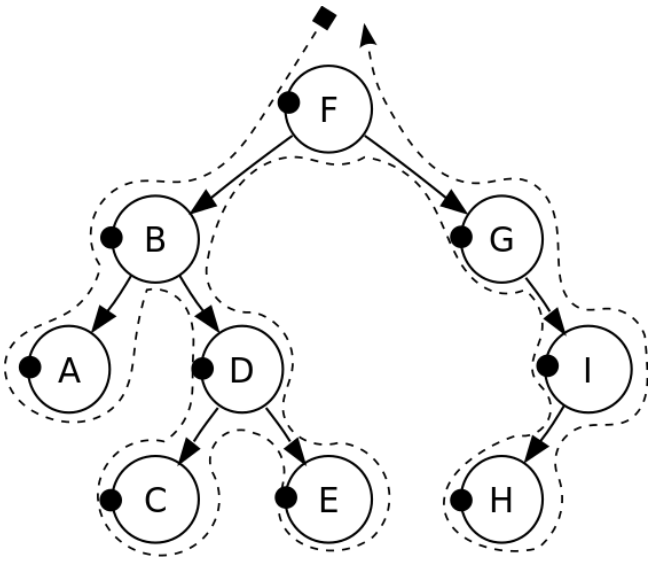

Figure 1: Pre-order depth-first traversal F-B-A-D-C-E-G-I-H

Option B: Data flow analysis

Data flow analysis in Clang is done through the *Checker* class. For example, in order to make sure a variable's memory is allocated and freed once, a new class which derives from the *Checker* class is created. To the new class, the methods *checkPreCall*, *checkPostCall*, *checkDeadSymbols*, and *checkPointerEscape* are added. During the pre-call and post-call methods, one keeps track of the variable's status, such as warning if allocation happens when it already points to data. In the dead symbols-method one should make the final check that the variable's data is deallocated before the variable is garbage collected. In pointer escape one needs to handle the case when the analyzer no longer can make any guarantees regarding the variable's state. This can happen if a variable is exposed to unknown functions. With this possibility of uncertainty, data flow analysis often has a significantly higher rate of either false positives or false negatives.

When the rule checker sufficiently covers all the test cases successfully, it is run on a few thousand lines of real source code at Saab. The result is analyzed to assert that there are no false positives or negatives. In case these occur, a test case is added for the given case, and the process repeated. Otherwise, the rule is considered solved, and work on the next rule begins.

Each rule is expected to take approximately two days, or sixteen work hours, to solve, which results in eight different investigated rules. However, as the rules vary wildly in complexity, it is possible to get stuck on an especially tricky one. For this reason, the maximum allotted days for a single rule is four days. If four days pass without a rule being sufficiently covered, it is left as-is. Thus, in a worst case scenario, only four different rules are partly covered, which should result in enough data to analyze.

Sometimes, the rule checkers are similar, or identical, to existing built-in Clang checks. Since this project is focused on the process of implementing custom checkers in Clang, the built-in warnings are ignored. The exception for this is when Clang raises errors, instead of warnings. When this happens, the code is deemed incorrect C++ and therefore not covered with a custom checker.

Evaluation

The first research question is treated as true for every rule which is covered with a 1:1 match between rule violations in the code and warnings raised from the checker.

The second research question includes the time it takes for both testing and implementing the given rule, as these processes are tightly knit together. The initial startup time for the project, including getting familiar with Clang, is not included in rule 1, but is instead a separate post. This is done to make sure the implementation time listed for the first rule is not unfairly long compared to the subsequent rules.

For the third research question, the key factor is the combined results from RQ1 and RQ2. In case of exceptionally good or bad traits, such as execution time, or memory usage, these are also taken into consideration.

RESULT

This section starts with a listing of the resulting interpretation of the different rules. They are either rewritten into subsets of rules, clarified, or in some cases almost identical to the initial rule. After this, information on implementation time is listed. Finally, the general result and the presented data is explained.

Rule interpretation

Rule #1

- Warn on casting to or from a Boolean type.
- Warn on casting to or from the char type.
- Warn on casting to or from an enumerable type.
- Warn on casting to or from a void pointer type.
- Warn on casting to or from a function pointer type.
- Warn on casting any pointer into a Boolean type pointer.
- Warn on casting a pointer to one type into a pointer to a potentially larger type.
- Warn on casting a floating point type into an integer type.
- Warn on casting a pointer type into an integer type or vice versa.
- Warn on the operator== method not returning a Boolean.
- Warn on the use of unions.
- Warn on unions mixing floating point types with integers or PODs.
- Warn on the use of increment operator on a Boolean type.
- Warn on the use of bitwise operators on enumerable types which may be negative.

Rule #2

- Warn on reusing variable names in nestled scopes.
- Warn on reusing variable and method names from a base class, unless they override said method.

Rule #3

- Warn on the use of increment and decrement operators on pointers.
- Warn on dereferencing of a parenthesis expression, e.g. **(ptr +1)*.
- Warn on reading from or writing to array subscriptions, e.g. *int a = arr[1]*.

Rule #4

- Warn on type, object and function definitions in source files, which are not inside a namespace.
- Exception: the *main* function.

Rule #5

- Warn when the assignment operator of a class T does not return a value of type *void*, *T const&* or *T const**.

Rule #6

- Warn when pointers to functions do not use a typedefined type representing the function pointer.
- Exception: *typedef* declaration are allowed to use pointers to functions.

Rule #7

- Warn on variable definitions in a function which take place after any executable code.
- Exception: A definition with an executable initializer does not prevent future definitions from taking place.
- Exception: For-loops are always allowed to define variables in their initialization.

Rule #8

- Warn on implicit conversions from pointers to Booleans.
- Warn on binary operations using NULL not having the same type on the left and the right hand side

Rule #9

- Warn on the use of *#define* to create a numeric constant.

Rule #10

- Warn on the use of bitwise unary or binary operations on a value of Boolean type.

Rule #11

- Warn on unary or binary operations which modify a variable's value inside a function's call parameters.

Rule #12

- Warn on structs, classes and enums which contain a declaration in its type definition.

Rule #13

- Warn on the use of keywords *register*, *auto*, *export* and *mutable*.

Rule #14

- Warn on the use of the binary operations *=, /=, <<=, >>=, &=, ^=, %= and |=.

Rule #15

- Warn on the use of pointers to functions.
- Exception: *typedef* declarations are allowed to use pointers to functions.

Rule #16

- Warn on the use of standard C++ types.
- Exception: *void* and *char* are allowed.
- Exception: *typedef* declarations are allowed to use standard types for defining new types.

Rule #17

- Warn on block statements which do not follow control statements such as *if*, *for* or *switch*.

Implementation time

The implementation time for a single rule turns out to be between thirty minutes and eighteen hours, where the average is 3.9 hours and the median two hours. See Table 1 below for full information.

In addition to the time for each rule, approximately eight hours is spent prior to rule 1 getting acquainted with Clang plugins. This time includes figuring out the boilerplate code necessary to create a Clang plugin, which is later reused in each rule implementation. It also includes the process of understanding how to extract information from the AST.

Task	Implementation time
Startup time	8 hours
Rule 1	18 hours
Rule 2	9.5 hours
Rule 3	2.5 hours
Rule 4	8.5 hours
Rule 5	1 hour
Rule 6	1 hour
Rule 7	5.5 hours
Rule 8	1.5 hours
Rule 9	5.5 hours
Rule 10	1 hour
Rule 11	2 hours
Rule 12	0.5 hours
Rule 13	4.5 hours
Rule 14	0.5 hours
Rule 15	1 hour
Rule 16	2 hours
Rule 17	2.5 hours

Table 1: Implementation time per rule

General results

The resulting rule implementation time is much lower than the initially expected two days, which results in a surprisingly large number of successfully implemented rules. The reasons for this are investigated in the discussion chapter.

Every implemented rule, with the exception of rule 13, has a 1:1 rate for rule violations to error messages, both when applied to unit tests and real code. The reason rule 13 cannot be covered by AST analysis is that one of the four keywords, namely *export*, is not supported by Clang. The lexer can create an *export* token, but when this token reaches the parser, Clang emits a warning and simply discards it. Normally, this would result in the case begin covered, however, as this prototype has the requirement of ignoring all built-in Clang warnings, the rule violation cannot be detected by normal means. A work around, which may not be compatible with other plugins, is expanded upon in the discussion chapter.

The execution speed and memory usage difference between running all plugins and no plugins is not noticeable. This is the case for both the test files, and the real code at Saab. It is noteworthy that the tested source files are a few hundred lines each, and very large files may yield a different result.

DISCUSSION

The discussion chapter will first focus on interpreting the resulting data from developing the prototype. After this, the method and the selected rules are evaluated.

Result

The biggest deviation from the initial expectations is no doubt the development speed. Even expecting a two days' implementation time was initially expressed as optimistic by the project supervisors at Saab. The greatest factors affecting the development times are the possibility of code reuse, node helper functions, and the Clang documentation.

Code reuse

When creating a Clang plugin, much of the code is almost identical between the different plugins. There are a few classes and a plugin registration which needs to exist to be able to load and execute the plugin. This boilerplate code, which in the end is quite short, initially takes a few hours'

time figuring out. When reusing the boilerplate code from previous plugins, the time spent on writing actual code in a new plugin is often only a few minutes. The large majority of the time is instead dedicated to understanding a rule, or attempting to find false positives or negatives. The fact that such a small amount of time is spent on writing new code is due to the rich built-in functionality of the node helper functions.

Node helper functions

The most impressive part about Clang's AST is the amount of help it gives you for absolutely free. There is seldom any need to create smaller utility functions, as Clang's developers have these already figured out and implemented. For example, to know if a node is a pointer to a function, you simply check the following:

```
node->getType()->isFunctionPointerType()
```

A simplified version of implementing rule 10, without any error handling, can be as simple as this:

```
if ((op->isBitwiseOp() || op->isShiftOp()) &&
    (op->getLHS()->getType()->isBooleanType() ||
     op->getRHS()->getType()->isBooleanType())) {
  // Report error
}
```

There are even niche methods in abundance, such as:

```
class_node->hasCopyConstructorWithConstParam()
```

In addition to this, all Clang types are made up of layers, which can easily be peeled off to check their underlying types. This makes it easy to do thorough investigations of even complex types, such as arrays of typedefined pointers to functions, by looping over the type and gradually unpacking it.

As a result, one of the main problems during the prototype development becomes finding the correct method to use, which can be tricky because of how Clang's documentation is handled.

Clang documentation

Clang's documentation is at the same time highly impressive and highly disappointing. Since it is automatically generated from Doxygen [21], there is information on every method and variable. Sadly, there are very few code examples available, and names can be ambiguous, such as *overridden_methods* in *CXXMethodDecl*. It can be difficult to understand if this means methods which have overridden this method, or vice versa. As a result, usage is often discovered through trial and error, by manually testing functions until a good match is found, or delving into the source code to understand something in greater depth.

An example of this problem is when attempting to find parents to different code statements. A declaration node can simply be asked for its parent through its *getParent*-method, which will return the parent's context. Statement nodes, on the other hand, does not have this. Instead, one is supposed to create a *ParentMap* object for a given statement, and execute:

```
parent_map.getParent(substatement)
```

in order to receive the sub statement's parent, which needs to be somewhere between the statement which has the *ParentMap* and the sub statement. This is not an obvious method, and was only discovered by searching Clang's developer mailing list.

The fact that many functions are not thoroughly explained can also sometimes makes it hard to trust using Clang's built-in functions, instead of writing your own. For example, rule 16 basically requires all variables to use types declared through *typedef*. By reading the Clang documentation, it becomes clear that there is are *isCanonical* and *getCanonicalType*-methods for types, which seem like a perfect match. The manual states that a canonical type is a type with all *typedefs* removed. However, when running this method on deeply nested types, this fact does not always seem to hold true, and it is hard to know if this is due to a Clang bug, a bug in the prototype implementation, or the methods simply not being properly explained. Most of the time, however, the Clang documentation works well when knowing what to look for, and abysmal when attempting to find an approach a given problem.

Notable rules

As mentioned in the results, all implemented rules, with the exception of rule 13, have successfully reached a 1:1 match between error messages and rule violations. This does not tell the whole story however. For example, rule 1 has problems resulting from its ambiguity and complexity, and the interpretation of rule 11 can be considered lacking. Below is a list of rule-specific issues faced during the implementation process.

Rule 1

The first rule is no doubt the most complex of them all, due to its relation to different machine architectures and compilers. While the rule has a large prewritten test suite, not all of these tests are easily understood, and some of them are in the end not even valid C++, according to Clang. There is no problem implementing the understood rules from the test suites, however, it is found obvious that the existing tests cannot possibly cover all edge cases for such a large scope as all machines and compilers. At this stage, my personal lack of experience in compiler and machine differences is the limiting factor for this rule. With the help of an expert in compiler technology, it may be possible to cover most, if not all, of this rule. However, in the end the implementation only covers some known problematic uses, which should only be treated as a complement to manual analysis.

Rule 9

The preprocessor directive *#define* does not show up in the AST, which makes it not solvable by option A or B in the method section. Instead, a new option C is used, which works through using so called *PPCallbacks*. *PPCallbacks*, which

stands for preprocessor callbacks, are triggered by the preprocessor when certain directives are encountered, such as *#if*, *#define* or *#pragma*.

Since *PPCallbacks* are hooks to the preprocessor, which is the first stage in the compilation process, there is less information available compared to normal AST analysis. However, basic information, such as tokens data, is still available. In this case, it is possible to prohibit using *#define* for numeric constants simply by checking if the next token encountered is a numeric constant, which has its own type of token. As a result, this method is well suited for basic checks, but may not work satisfactory for more advanced checks regarding preprocessor-directives.

Rule 11

A keen reader may notice that there is another way of introducing side effects than considered in my interpretation, i.e. though binary or unary operations. For example, there is no guarantee that a function call does not have any side effects. Consider the following code:

```
fun1(fun2(), fun3())
```

Without access to the source code of fun2 and fun3, there is no way of knowing if either function modifies the global state. To solve this, there is either need for annotations in the function definition in the header file, or the use of data flow analysis.

Suitable annotations for this purpose would be *__attribute__ ((pure))* or *__attribute__ ((const))*, which are both GNU extensions and therefore not part of the official C++ language. These annotations guarantee a slightly different degree of lack of side-effects, which could be used to determine if the function is allowed as a parameter to a function call. The problem with this approach is the need to rewrite all existing header files, including for the standard library, if these are to be allowed. All function calls without these, or similar, attributes should thereafter be disallowed as parameters to another function call.

The other approach, data flow analysis, would in theory be able to compare variables used by both functions *fun2* and *fun3*, and warn if one modifies a variable used by the other. In practice, this cannot be covered sufficiently through Clang, as its static data flow analysis does not support several translation units. This means that if *fun2* or *fun3* belong in another source file, no investigation can take place. As the C++ standard library is precompiled, these functions cannot be investigated either.

Another solution to this problem would likely be using dynamic data flow analysis, which is supported by Clang. However, this is not inside the scope of this project, and therefore not further researched.

Rule 13

As mentioned in the results, one of the prohibited keywords does not end up in the AST, and is therefore impossible to find using only AST analysis. Since the Clang lexer does create tokens for even this keywords, any attempt to find it must be done prior to parsing. This issue is similar to the one faced in rule 9, however there seems to be no counterpart to *PPCallbacks* for the lexer to execute on tokens received. The most viable solution in this case would normally be using the built-in warnings in Clang, however, as this is out-of-scope for this prototype tool, another potential solution is attempted.

By executing the *Preprocessor*'s *Lex*-method, it is possible to manually extract one token at a time from the source code. Through this method, one can loop over all tokens until end of file is reached. During the loop, the tokens can be analyzed, and warnings created in case of rule violations. In this case, each token encountered is compared to one of the four keywords in rule 9. As each keyword has its own token, this approach guarantees that there can be no false positives nor false negatives.

The problem with this approach is that manually running the lexer eats up all input data, which removes the possibility of later creating the AST. It would therefore not work as a Clang plugin, as it would effectively block the other plugins from working, as they all rely on the AST for analysis. If there is a way to reload input data, effectively causing the lexer to run once for analysis and once for creating the AST, this can be a viable solution. At the moment, this only works as a stand-alone plugin, and is therefore a low quality solution. For future work, it could be interesting to investigate the possibility of extracting tokens and still creating the AST. This could be possible by overloading either the lexer class, the parser class, or the preprocessor class.

Method

The use of Clang plugins to approach the problem seems like a good solution. It is too early to tell any problems with new versions of Clang, which can only be experienced during the actual act of porting the rules. The possibility of speedier development through LibClang does not feel tempting as the Clang plugins are very fast to develop, where the bottleneck instead lies in understanding the rule to implement, as well as finding the suitable functions in Clang's documentation.

Adding many plugins to Clang does not seem to notably affect memory usage or execution time, and is therefore not further investigated. The additional plugins do, however, require a lot of execution flags to be passed to Clang. In fact, they are too many to be comfortably written by hand. Our solution is a written python script which adds flags depending on the plugins to load. A more suitable solution for this would be writing all rules as a single plugin and letting the plugin take arguments depending on which rules to enable. This is something which plugins have built-in support for. It should be possible to retain a modular design with this approach by placing each rule in a separate source file and calling them from the main plugin source file.

Of the implementation options A and B, namely AST analysis and data flow analysis, only AST analysis is actually implemented. This can be viewed as either a success or a failure. It is a success since it proves that the support for AST analysis is sufficient enough to cover rules which do not depend on path-specific code execution. It can also be seen as a failure, as investigating Clang's support for data flow analysis affects how viable LLVM's support for static code analysis as a whole should be viewed.

One of the reasons for data flow analysis not being implemented is the fact that data flow analysis is most suitable when states are involved. Examples of this can be file stream, where data flow analysis can help make sure the streams are opened and closed once; or heap allocated, to make sure data is deallocated. As neither of these, nor similar rules, exist, the method does simply not provide any advantage compared to AST analysis. While it possible that a rule should be added to require the need for data flow analysis, the fact that static data flow analysis in Clang does not support several translation units, severely limits its use.

Instead of option B, an option C is required to solve rule 9, i.e. registering preprocessor callbacks for preprocessor-specific analysis. As the rule clearly depends on information which is not usually part of the AST, it can be seen as a lack in preparation not including an option for pre-AST analysis from the beginning. The reason for this happening is that neither the Clang internals manual, nor the Doxygen page provide much information regarding pre-AST analysis at all. Only after first attempting to override the whole *Preprocessor* class, which does not seem to provide a solution, and thereafter methodically testing the *Preprocessor* class' methods was *PPCallback* found. Therefore, the lack of an option C from the start could be justified.

Regarding the rules selected for implementation, it is discovered that most of these require a different approach to solve. Except for rules 1, 3, 8, 14, and 15, every rule has its own unique aspects with their own problems and solutions. While these rules mentioned, in hindsight, do not add to the project, this fact can only be realized by attempting their implementation. Admittedly, both rules 14 and 15 are exceedingly similar to previous rules, which should be obvious even before starting the implementation.

A last observation, before concluding the method discussion, relates to the second research question. The implementation time of a rule, especially when combining the implementation with testing, depends very much on the programmer. Not only does one have to take into consideration the speed at which the programmer creates new code, but also their thoroughness in testing. Replicating this study will surely result in different implementation time per rule. However, the conclusion to draw is not the exact time it takes per rule, but rather its order of magnitude.

Wider context

All development in the field of code analysis strongly contributes to a safer society. As people depend more and more on technology, it becomes more and more vital that breakthroughs are made in this field. Not only is it required for the aircraft industry. Technical issues in other fields, such as hospital equipment and online banking can have very severe consequences. Therefore, the advent of a free, open source alternative should not be disregarded.

Source Criticism

To help ensure an overall high quality in sources for this paper, information from well known, reputable sources such as ACM and ICSE has been prioritized. Information regarding the subject matter comes from official sources, being papers published by the main author of LLVM, Christ Lattner, as well as the LLVM homepage.

CONCLUSIONS

It is my opinion that this thesis is successful in its research on using LLVM for static code analysis.

It is found that LLVM can be used to cover each given rule without emitting false positives or false negatives. While there are limitations in the first rule, the problem does not lie in LLVM, but in the rule itself. If the rule is clarified, it is highly likely that even this rule is fully coverable. There is also the exceptional case of rule 13. This warning already exists in clang, and can be emitted simply by giving Clang the correct execution flag. As there are also workarounds, which may be less than perfect, the end result is clearly that the rule can be covered through LLVM. The result of RQ1 is therefore seen as satisfied.

The method used in this thesis is efficient enough to be highly feasible to analyze, implement and test custom rules. That many rules already exist is another factor which may help when considering using Clang as a main tool for static code analysis. It is noteworthy that Saab has dedicated a lot of time previous to this project in order to find edge cases, and to create test suites. If this project is to be replicated elsewhere, the task of finding false positives and false negatives is likely to be significantly higher.

While the success in both RQ1 and RQ2 would imply that LLVM is very well suited for static code analysis, its lack of thorough documentation should not be taken lightly. Clang itself has indeed surpassed the initial expectations, but in order to develop a production quality tool, one should expect spending countless hours reading both the Doxygen documentation as well as Clang's source code to gather information on how to approach a problem.

In the end, the verdict is that, yes, LLVM is very suitable for static code analysis of C++ source code. However, until there is support for data flow analysis for several translation units, it may not be possible to completely replace a commercial tool with LLVM.

Future work

Applying new rules to a large existing project may evoke huge amounts of rule violations. This usually means giving one or several developers the task of manually updating the source code. By combining the *RecursiveASTVisitor* used in this thesis with Clang's *FixItHint*, it should be possible to automatically generate hints for how to solve the rule violation. If possible, this would provide a great complement to a static analyzer.

As this thesis does not evaluate Clang's built-in support for data flow analysis, this would be a logical next step to further evaluate LLVM's support for static code analysis.

REFERENCES

[1] J. C. Knight, "Safety critical systems: challenges and directions," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, Orlando, 2002.

[2] C. A. Lattner, "LLVM: An infrastructure for multi-stage optimization," University of Illinois at Urbana-Champaign, Illinois, 2002.

[3] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, San Jose, 2004.

[4] W. R. Bush, J. D. Pincus and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software - Practice and Experience,* vol. 30, no. 7, pp. 775-802, 2000.

[5] B. Chelf and A. Chou, "The Next Generation of Static Analysis," 2007. [Online]. Available: https://www.coverity.com/library/pdf/Coverity_White_Paper-SAT-Next_Generation_Static_Analysis.pdf. [Accessed 23 February 2016].

[6] S. C. Johnson, "Lint, a C program checker," Bell Telephone Laboratories, 1977.

[7] I. Gomes, P. Morgado, T. Gomes and R. Moreira, "An overview on the Static Code Analysis approach in Software Development," Faculdade de Engenharia da Universidade do Porto, Porto, 2008.

[8] D. S. Wile, "Abstract syntax from concrete syntax," in *ICSE '97 Proceedings of the 19th international conference on Software engineering*, 1997.

[9] N. Truong, P. Roe and P. Bancroft, "Static analysis of students' Java programs," in *ACE '04 Proceedings of the Sixth Australasian Conference on Computing Education*, 2004.

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Software Maintenance, 1998. Proceedings., International Conference on*, 1998.

[11] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM,* vol. 19, no. 3, pp. 137-147, March 1976.

[12] S. Adams, T. Ball, M. Das, S. Lerner, S. K. Rajamani, M. Seigle and W. Weimer, "Speeding Up Dataflow Analysis Using Flow-Insensitive Pointer Analysis," in *Static Analysis*, Springer Berlin Heidelberg, 2002, pp. 230-246.

[13] E. M. Clarke, E. A. Emerson and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 8, no. 2, pp. 244-263, 1986.

[14] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler and D. L. Dill, "CMC: a pragmatic approach to model checking real code," *ACM SIGOPS Operating Systems Review,* vol. 36, no. SI, pp. 75-88, 2002.

[15] B. Blanchet, " A static analyzer for large safety-critical software," in *ACM SIGPLAN conference on Programming language design and implementation*, San Diego, 2003.

[16] "clang: a C language family frontend for LLVM," [Online]. Available: http://clang.llvm.org/. [Accessed 24 February 2016].

[17] "“Clang” CFE Internals Manual," [Online]. Available: http://clang.llvm.org/docs/InternalsManual.html. [Accessed 24 February 2016].

[18] "DataFlowSanitizer," [Online]. Available: http://clang.llvm.org/docs/DataFlowSanitizer.html. [Accessed 24 February 2016].

[19] F. Merz, S. Falke and C. Sinz, "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR," in *Verified Software: Theories, Tools, Experiments*, Springer Berlin Heidelberg, 2012, pp. 146-161.

[20] J. Henry, D. Monniaux and M. Moy, "PAGAI: A Path Sensitive Static Analyser," *Electronic Notes in Theoretical Computer Science,* vol. 289, pp. 15-25, 6 December 2012.

[21] "Doxygen," [Online]. Available: www.doxygen.org. [Accessed 19 May 2016].