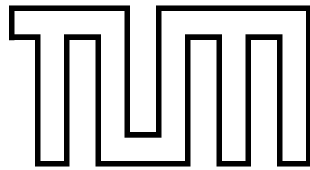


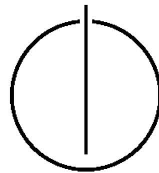
FAKULTÄT FÜR INFORMATIK

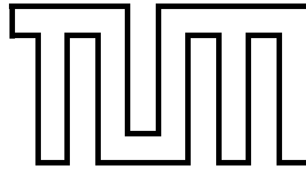
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Control Flow Analysis for Event-Driven Programs

Florian Scheibner





FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

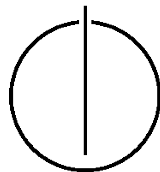
Bachelor's Thesis in Informatics

Control Flow Analysis for Event-Driven Programs

Kontrollflussanalyse von ereignisgesteuerten Programmen

Author: Florian Scheibner
Supervisor: Dr. Christian Grothoff
Advisor: Sree Harsha Totakura

Date: July 16, 2014



I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Munich, July 16, 2014 Florian Scheibner

Acknowledgments

I would like to thank Dr. Christian Grothoff for supervising my thesis and his ideas.

I also want to thank Sree Harsha Totakura for his helpful advice and support throughout the last months.

Abstract

Static analysis is often used to automatically check for common bugs in programs. Compilers already check for some common programming errors and issue warnings; however, they do not do a very deep analysis because this would slow the compilation of the program down. Specialized tools like Coverity or Clang Static Analyzer look at possible runs of a program and track the state of variables in respect to function calls. This information helps to identify possible bugs. In event driven programs like GNUnet callbacks are registered for later execution. Normal static analysis cannot track these function calls. This thesis is an attempt to extend different static analysis tools so that they can handle this case as well. Different solutions were thought of and executed with Coverity and Clang. This thesis describes the theoretical background of model checking and static analysis, the practical usage of wide spread static analysis tools, and how these tools can be extended in order to improve their usefulness.

Zusammenfassung

Statische Codeanalyse wird of zur automatisierten Fehlersuche verwendet. Compiler überprüfen bereits einfache Programmierfehler und geben Warnungen aus, es wird jedoch keine tiefere Analyse des Programms durchgeführt da dies den Compiliervorgang verlängern würde. Spezialisierte Programme, wie zum Beispiel Coverity oder Clang Static Analyzer, untersuchen alle möglichen Programmpfade und verfolgen die Werte von Variablen. Mithilfe dieser Informationen können sie auf mögliche Fehler schließen. In ereignisgesteuerten Programmen, wie zum Beispiel GNUnet, werden Callback Funktionen registriert und später ausgeführt. Durch normale statische Codeanalyse können diese Callback Funktionen nicht analysiert werden. Diese Bachelorarbeit verfolgt das Ziel die Tools so zu erweitern damit auch dieser Fall abgedeckt werden kann. Verschiedene Lösungen wurden für Coverity und Clang erarbeitet. Diese Arbeit beschreibt die theoretischen Hintergründe zu Modellprüfung und statischer Codeanalyse, die Verwendung von verbreiteten statischen Analyse Tools und wie diese für diesen Fall verbessert werden können.

Contents

Acknowledgements **vii**

Abstract **ix**

1. Introduction **1**

2. Related Work **3**
2.1. Model Checking 3
2.1.1. Flow Analysis 3
2.1.2. Type Systems 5
2.2. Static Analysis 6

3. Expected Results **9**
3.1. Bugs Expected to Find 9

4. Static Analysis Tools **11**
4.1. Coverity 11
4.1.1. Extendable by Models 12
4.1.2. Restrictions of Coverity 12
4.1.3. Commandline Usage 13
4.2. Clang 14
4.2.1. Different Checkers 15
4.2.2. Restrictions of Clang 16
4.2.3. Commandline Usage 16
4.3. Comparison 17

5. Approach **19**
5.1. Use Model for Coverity 20
5.2. Patching the Source Code 21
5.3. Analysis with Clang 21
5.4. Problem 22
5.5. Aggregate Multiple Source Files 22

6. Source Code Aggregation **23**
6.1. Structure of Automake Files 23
6.2. How Normal Builds Work 24
6.3. Aggregating TUs 24
6.4. Parsing Makefile.am 25
6.4.1. Control Flow Conditions 25

6.5. Prevent Naming Conflicts . . . 26
6.5.1. Renaming Data Types . . . 27
6.5.2. Renaming Variables and Functions . . . 27
6.6. Encountered Problems . . . 28
6.7. Drawbacks . . . 29

7. Results **31**
7.1. GNUnet . . . 31
7.2. Libevent . . . 33
7.2.1. Simple Test Program . . . 33
7.2.2. Tor and tmux . . . 34

8. Conclusion **37**

Appendix **38**

A. Aggregation Script in Python **41**

B. Typesystem for Functional Programs **49**

C. Coverit E-Mail Support **51**

D. Aggregation Script Commandline Usage **53**

Bibliography **55**

1. Introduction

This thesis started out with the goal of improving the analysis of control flows in event driven programs written in the C language. Static analysis tools allow finding common programming errors by analyzing a program's control flow. By using the techniques described in Section 2.1, these tools follow possible program flows and infer possible bugs. This is done by tracking the value of variables over multiple function calls in the control flow and checking if it reaches any illegal program states. This offers the ability to observe wrong behaviour. For example, a flaw could be a variable that has been allocated using `malloc()` and is freed twice via `free()`. Static analysis tools remember that this variable has already been freed and flags the second free as a bug in the analysis report.

Event driven programs have a scheduler that implements an event loop. Event loops are used to wait for events such as interrupts from Input/Output(IO) and timed alarms. When an event occurs, the scheduler executes a callback function registered for that event. Since the execution of the callback is asynchronous, the callback is registered with a closure which contains state information that is passed to the callback function when it is executed. Static analysis tools cannot understand this scheduler function because the callback and the closure are stored in the memory first and they are executed only later after invocation of the event loop function. This is explained in Chapter 5. This deficiency means that the static analysis tools cannot follow the control flow via these paths, and consequently they do not detect when there is a bug in these control flows. Chapter 3 shows which bugs could occur. In this thesis we outline the different approaches we tried so that a static analysis tool can also follow the program flow via these event callbacks. As a result, more bugs could be found.

The motivation for this thesis stemmed from the GNUnet project, [1], developed by the Free and Secure Network Systems Group at TUM. The source code is regularly analyzed for common programming errors by using the commercial tool Coverity[2] and the Clang Static Analyzer[3]. The features of these tools are described in Chapter 4. Since GNUnet employs event loops, the existing static analysis tools were unable to track bugs resulting from asynchronous control flows.

Efforts are made to extend these tools to also track asynchronous control flows. Model extensions are tried with Coverity to extend functions of event loops, but due to its inability to track function pointers this approach is not successful. Clang is able to track function pointers but it can only analyze each translation unit separately. A translation unit (short TU) is the output of the preprocessor, *i.e.* it is the source file with all the macros and include

[1] `https://gnunet.org/`
[2] `http://www.coverity.com/`
[3] `http://clang-analyzer.llvm.org/`

directives expanded[1, p. 9]. Finally, the intended analysis could be carried using Clang by combining all translation units into one as explained in Chapter 6.

In order to test the generality of our approach, the libevent[4] project was analyzed as well. It offers a generic event loop functionality and can be used by other projects as a library. Different projects such as Tor, tmux, and Google Chrome use this event loop library. In this thesis, the Tor and the tmux project have been analyzed using the same approach that was used for GNUnet. The results can be found in Chapter 7. Possibilities to improve the approach are discussed in conclusion (Chapter 8).

[4] http://libevent.org/

2. Related Work

There are some properties that specify the precision of analysis techniques. *Flow sensitive* means that the analysis considers the order of execution of statements. *Path sensitive* means it distinguishes between different paths through the program. *Interprocedural* means that the analysis also follows into the called function. *Context sensitive* means that the return value that is used for the analysis depends on the callsite, context insensitive analysis can only infer a set of possible return values. [2, p. 1166]

Model checking and static analysis are two different techniques to verify correctness properties of a program. Model checking tries to achieve this by transforming the high level source code to some formal model for which special verification algorithms exist. This is for example a propositional or temporal logic[3]. It is a very precise analysis which is always flow sensitive. Static analysis, on the other hand, is often more imprecise. There are different kinds of static analysis. Compilers do a shallow static analysis for optimization, such as finding unused variables. More precise analysis would slow down the compilation speed. There are special tools for that purpose. They analyze the propagation of values through the program and find flaws like division by zero[4]. The distinction between model checking and static analysis became less with time. Model checking used to be very precise and flow sensitive, whereas static analysis was very abstract and flow insensitive. This has changed and abstraction is also used for model checking and static analysis is often flow sensitive[2]. In practice there is still the distinction that static analysis is used for a general analysis over a large code base whereas model checking is more used to analyze a small part of the code.

2.1. Model Checking

2.1.1. Flow Analysis

In order to apply model checking to a program, all statements have to be represented as logical formulae. The concept is to have all program transitions in the model and define the wanted and unwanted states. A proof can then be constructed to show that from a given starting state the unwanted states are never reached. Figure 2.2 shows an example of a model using propositional logic for the source code in Listing 2.1. A SAT solver can be used to determine whether the error state is reachable. The syntax is based on the lecture "Model Checking" by Professor Andrey Rybalchenko [5].

```
1  int ret
2  main(x)
3      assume(x >= 0)
4      f(x) 2
5      assert(ret >= 0)
6
7  f(a)
8      if (a >= 0)
9          f(a − 1)
10         ret = a − ret
11     else
12         ret = 1
```

Listing 2.1: Sample program

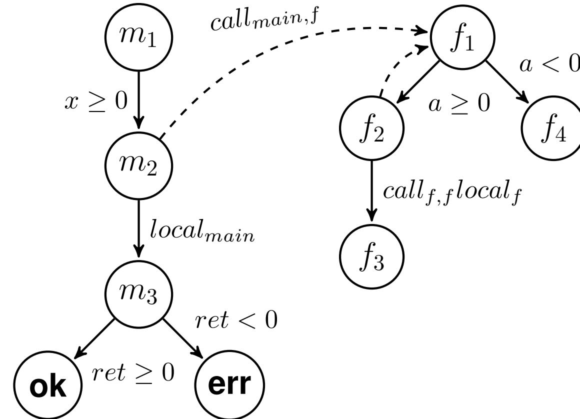


Figure 2.1.: Call graph of procedures main and f

To apply logical reasoning to the program, each statements' effect on the variables has to be expressed as logical formula. The derived logical formulae are called the model of the program. Each statement can change the global and local variables. This is modelled as having the set of global variables V_G before the statement and the modified set V'_G afterwards. Equivalently the changes to the set of local variables V_p are defined in V'_p. The modelling for procedures $p, q \in P$ can be constructed as:

init(V_G**,** V_p**)** —init condition
This clause assigns default values to the global variables and sets the program counter to the first line of the procedure p.

error$_p$**(** V_G**,** V_p**)** —error condition
All program states that satisfy the error condition are unwanted behaviour. The goal of model checking is to prove for a given program that the error condition will never be reached.

step$_p$**(** V_G**,** V_p**,** V'_G**,** V'_p**)** —Intra procedure step
Each action in a procedure is modelled with the step clause. Global and local variables can be read and changed; this includes the program counter that moves to the next step. The shortcut $move_p(s_1, s_2)$ can be used instead of $pc_p = s_1 \land pc'_p = s_2$. The program counter does not move to another procedure but only inside of one procedure. Conditions like `if` statements are written as inequalities over the given variables. All variables of V_p that are still needed later on need to be defined for V'_p by means of $x' = x$ or just $skip(x)$.

call$_{p,q}$**(** V_G**,** V_p**,** V_q**)** —procedure call
The call clause initializes a new set of local variable V_q for the sub procedure. It contains the arguments, local variables defined in q, and the program counter. No additional variables are changed.

ret$_p$**(** V_G**,** V_p**,** V'_G**)** —return value passing
The ret clause sets the return value of a procedure by setting the global ret variable in V'_G. No further changes are made.

local$_p$**(**V_p**,** V'_p**)** —callsite program counter change
This is the third part of the call constellation. While the first two clauses, *call* and *ret*, change the program counter of the called procedure and set the return value, *local* applies to the calling procedure and moves its program counter forward.

The states from Figure 2.1 are used as values for the program counter. For each intraprocedural step there is one conjunction in the *step* formulae in Figure 2.2. For the sake of simplicity no stack is modelled here that could handle the return values. Consequently, a global variable ret is used, which is modified at the end of each procedure and passes the value on to the calling procedure. This change is modelled in the set of global variables V_G. The model that can then be verified is the conjunction of all the separate formulae. If the model satisfies the error condition, then there is a bug in the program.

$$
\begin{array}{rcl}
init(V_G, V_{main}) & = & (p_{main} = m_1) \\
step_{main}(V_G, V_{main}, V'_G, V'_{main}) & = & (move_{main}(m_1, m_2) \wedge x \geq 0 \wedge skip(ret, a)) \vee \\
 & & (move_{main}(m_3, ok) \wedge ret \geq 0 \wedge skip(ret, x)) \vee \\
 & & (move_{main}(m_3, err) \wedge ret < 0 \wedge skip(ret, x)) \\
step_f(V_G, V_f, V'_G, V'_f) & = & (move_f(f_1, f_2) \wedge a \geq 0 \wedge skip(ret, a)) \vee \\
 & & (move_f(f_1, f_4) \wedge a' < 0 \wedge skip(ret, a) \\
call_{main,f}(V_G, V_{main}, V_f) & = & (pc_{main} = m_2 \wedge pc'_f = f_1 \wedge a' = x) \\
call_{f,f}(V_G, V_f, V_f) & = & (pc_f = f_2 \wedge pc'_f = f_1 \wedge a' = a - 1) \\
local_{main}(V_{main}, V'_{main}) & = & (move_{main}(m_2, m_3) \wedge skip(x)) \\
local_f(V_f, V'_f) & = & (move_f(f_2, f_3) \wedge skip(a)) \\
ret_f(V_G, V_f, V'_G) & = & (pc_f = f_3 \wedge ret' = a - ret) \vee \\
 & & (pc_f = f_4 \wedge ret' = 1) \\
ret_{main}(V_G, V_{main}, V'_G) & = & (pc_{main} = ok \wedge ret' = ret) \\
error_{main}(V_G, V_{main}) & = & (pc_{main} = err) \\
model(V_G, V_{main}) & = & init(V_G, V_{main}) \wedge step_{main}(V_G, V_{main}, V'_G, V'_{main}) \wedge \\
 & & step_f(V_G, V_f, V'_G, V'_f) \wedge call_{main,f}(V_G, V_{main}, V_f) \wedge \\
 & & call_{f,f}(V_G, V_f, V_f) \wedge local_{main}(V_{main}, V'_{main}) \wedge \\
 & & local_f(V_f, V'_f) \wedge ret_f(V_G, V_f, V'_G) \wedge \\
 & & ret_{main}(V_G, V_{main}, V'_G)
\end{array}
$$

Figure 2.2.: Model of Listing 2.1

The first step for model checking has been done. The original program code has been transformed into a model that can be analyzed for correctness properties. There are different proof systems: for example, to show that a program terminates or that the error state is never reached.

2.1.2. Type Systems

Besides analyzing the flow in a program, the types of variables should always match as well. In a statically typed language like C the compiler can already check most forms of

$$\dfrac{\dfrac{\texttt{T(a) = bool}}{\texttt{T} \vdash \texttt{a : bool}} \quad \dfrac{\texttt{T(b) = int}}{\texttt{T} \vdash \texttt{b : int}} \quad \dfrac{3 \in \mathbb{Z}}{\texttt{T} \vdash \texttt{3 : int}}}{\texttt{T=[a: bool, b: int]} \vdash \texttt{if a then b else 3 : int}}$$

Figure 2.3.: Example of type inference

wrong assignments. There are, however, possible programming styles that prevent these checks. For example, using void pointers as function arguments.

To show the idea behind these checks, a simple functional language is used. The syntax can be found in Appendix B. For this simple case, only `int`s and `boolean`s are supported. The type environment T saves the type for each variable. Then different inference rules are applied. When no inference rule applies, then a type mismatch has been detected. Figure 2.3 shows an example of such an application of inference rules. In this case there is no error because all types match.

2.2. Static Analysis

Data flow analysis is a simple approach to static analysis. It is not very powerful and is only used by compilers for optimization as it is too simple to be useful for verification. There are two kinds of data flow analysis: forward and backward analysis. For example, expression that occur twice in the program can be computed only once when the participating variables don't change in between. Forward analysis can be used to detect whether variables have changed or not, this is called *Available Expression Analysis*. Another compiler optimization is to store variables in CPU registers for faster access. When the end of a block is reached, the compiler needs to know whether this variable is still going to be needed afterwards, if not it does not need to be stored. This case can be detected with backward analysis. Data flow analysis stores the variables that are changed or accessed in each step in order to draw these conclusions. [4]

Abstract interpretation is one technique to to detect errors in a program. It introduces new semantics that are an approximation of the original program. Lattices are used for this new abstract domain. A variable can then vary between not initialized and all values possible. The possible intermediate values can be defined differently. This abstract domain has to ensure that the program always terminates; therefore, there must be some heuristic in loops that sets the value to "all possible values" at some point. This analysis can be used to detect errors such as division by zero. [4]

Symbolic analysis is used when the input values are not known yet. The unknown values are represented as symbols and these symbols are used to describe expressions that use these values. Reading from a file can be such an unknown variable, for example. The variable a is read from a file and thus has the symbolic value ∇_1. The definition of $b =$

$a + 5$ results in b having the value $\nabla_1 + 5$. Symbolic analysis and be used for compiler optimization, but it can also be used to compute the worst-execution time.[4]

3. Expected Results

Static code analysis tools like Coverity find many bugs that occur through common programming errors. These include wrong use of `malloc()` and `free()`, usage of uninitialized pointers, and some buffer overflows. These conclusions can be drawn by building a call graph and inferring the possible values of variables and their propagation through function calls. The reports issued by the static analyzer tools help the programmer to improve his program. The automated tools, however, are not perfect. For example, they cannot understand constructs like the scheduler function introduced in Chapter 1. For a static analysis tool, this looks as if a function pointer and a closure are passed to a function and then stored in memory. The actual event, a timeout or IO interrupt, cannot seen by the tool and, as a result, the call of the callback is not analyzed.

These different parts of a scheduler system have to be taken into account:

TaskIdentifier scheduler_add(callback, closure) —Adds the callback and the closure to a queue. Returns an identifier of this task.

scheduler_cancel(task_identifier) —Removes the callback associated with this task identifier from the queue.

scheduler_run() —Starts the event loop which then invokes the callbacks.

callback(closure) —The actual execution of the callback when some event occurs.

The perfect model would take the scheduler_add and scheduler_cancel function into account. Consequently the callback function should only be analyzed when cancel is not going to be called. This kind of precision has not been achieved yet in this thesis; the approach taken for this thesis is described in Chapter 5. There are ways to improve this as shown in Chapter 8.

3.1. Bugs Expected to Find

Static analysis can find the kinds of bugs mentioned in Chapter 2. In this section I outline some examples of typical bugs that are not normally found in combination with scheduler callback functions. They can, however, be found using the method explained in this thesis.

```
1 int scheduler_add(void (*task) (void *closure), void *closure);
2 void scheduler_run();
3
4 void callback(void* closure) {
5     free(closure);
6 }
```

```
7
8  int main() {
9      void* m = malloc(sizeof(int));
10     scheduler_add(&callback, m);
11     free(m);
12     scheduler_run();
13 }
```

Listing 3.1: Double free bug

It is, for example, undefined behaviour when one block of memory that has been allocated using `malloc()` is deallocated twice using `free()`. Listing 5.2 shows such a bug. A callback is registered with the `scheduler_add()` function and the closure is deallocated directly afterwards. As soon as this callback is executed, the closure is deallocated a second time.

Furthermore, wrong casts of the `void* closure` pointer should be detected in the callback function. Sections 4.1.2 and 4.2.2 will show that the capabilities of the tested tools are limited in this respect. There is one version that is supported. Listing 3.2 shows a read in uninitialized memory by casting the pointer to a `struct` with a bigger size. Due to the way Clang tracks memory usage, this problem is found.

```
1  int scheduler_add((*task) (void *closure));
2  void scheduler_run();
3
4  struct A {
5      char a;
6  }
7  struct B {
8      long b;
9  }
10
11 void cb(void* m) {
12     struct* b = (struct B*) m;
13     printf("%i", b->b);
14     free(m);
15 }
16
17 int main() {
18     void* m = malloc(sizeof(struct A));
19     scheduler_add(&cb, m);
20     scheduler_run();
21 }
```

Listing 3.2: Wrong struct size

These test bugs will also be used throughout this thesis to test the effectiveness of the analyzed methods.

4. Static Analysis Tools

Model checking techniques and static analysis tools have the same general goal of finding programming errors. The approach is very similar by analyzing the possible program flows. There is, however, a methodical difference. While model checking is based on general mathematical proofs, static analysis provides generic checkers for common bug types. The first approach is often more complex, because preconditions and postconditions have to be explicitly defined for the given program. All bugs that can be inferred under these conditions are guaranteed to be found: this is called soundness. Then it is proven that the postconditions hold. As a drawback the modelling sometimes has to be simplified for more complex programs. This reduces the precision of the method and possibly declares a program as safe when it still contains a critical bug. On the other hand, static analysis can directly start with the analysis of source code without the need for special modelling. Common assumptions and assertions are built into the analysis tool. These tools are often optimized for a low false positive rate rather than soundness. For non critical programs, static analysis is the way to go. When hard proofs for the correctness of a system are needed then the model checking is inevitable. It is important to create a sufficiently detailed model.[6] In this thesis, the analysis of the two tools Coverity and Clang has been evaluated. Both tools do not give hard proofs of correctness with their analysis as it is possible with the Model checking approach from Chapter 2. For that, special pre- and postconditions would need to be written for every specific program. Instead, the two tools implement general checkers that only check constraints that need to hold for every program. For example, each block of memory that is allocated using `malloc()` needs to be deallocated with a respective call to `free()`. Both tools perform an flow sensitive and interprocedural analysis.

Another type of checking tool is Cppcheck which does not look at the control flow, it is context insensitive and intraprocedural. It only finds common programming errors inside of a function but cannot do further analysis. In contrast to Coverity or Clang, no call graph is computed but only a static abstract syntax tree. Values of variables are only tracked by assigning possible values statically to the Abstract Syntax Tree (AST). Its design is explained in [7]. Cppcheck helps to find many simple bugs, but it is not sufficient for analysis involving function pointers.

4.1. Coverity

Coverity is a static analysis tool that has been developed at Stanford university by Dawson Engler. He wrote the original paper [8] about the static analysis checks in 2000. Later he cofounded the company Coverity with some of his students in order to commercialize

the product. It is widely used and offers free analysis to opensource software [9]. The commercialization had a huge influence on the further development. The analysis had to be accessible to companies without too much academic background. Therefore the bug reports had to be simple. There were even some kinds of error checking removed because they would not offer useful information to an average developer [10].

4.1.1. Extendable by Models

Coverity needs information about library functions to be able to infer what they do. Models for most standard libraries are already shipped with Coverity. For example, the standard `malloc()` call has a default model which calls `__coverity_alloc__()` to tell Coverity that memory is allocated by this function. Listing 4.1 shows this model from the Coverity source code[1]. From this model, Coverity can derive that the return value is It sees these two options because The `success` variable is not set to any value, therefore Coverity derives that either a null pointer or a pointer to memory on the heap is returned. Coverity can then check that whether `free()` is called when a pointer to memory has been returned.

```
1  void *
2  malloc(size_t size) {
3      int success;
4
5      /* 1. Sinks if size is negative */
6      __coverity_negative_sink__(size);
7
8      /* 2. Returns a pointer to newly allocated memory block of size "size",
9       * or NULL otherwise.
10      */
11     if (success)
12         return (void*)__coverity_alloc__(size);
13     else
14         return NULL;
15 }
```

Listing 4.1: Coverity `malloc()` model

This modelling approach is not only available internally to Coverity but it is also exposed as an interface to the user[11]. This interface is used in Chapter 5 to create a model to tell the analyzer more about the source code behaviour. This model is shown in Listing 5.3.

4.1.2. Restrictions of Coverity

Coverity does not track the values of all variables or function calls. The heap and variable aliases are not modelled for the complete program, but heuristics are used to find possibly problematic sections [6]. There are some restrictions that arise from this: normally the static analysis can build a call graph through calls of function pointers such as

[1] cov-analysis-linux64-6.0.3/library/generic/libc/all/all.c

(*(&fun))(NULL). However, when the function pointer is assigned to a local variable first void (*pointer)(void*) = &fun; and then called via (*pointer)(NULL)), it is no longer part of the call graph. This also applies when a function pointer is used as an argument to a function, the call of this pointer is not tracked since the function pointer is stored in the argument variable.

The models for Coverity work quite well for the intended use cases, such as modelling a library function. Complex scenarios pose a problem: when modelling a function that has arguments with user defined data types and includes multiple header files, all these headers have to be included in the model file as well. Furthermore, Coverity does not define default macros as the GCC compiler. For example #define __UINT8_TYPE__ unsigned char, which is used by linux headers, has to be defined by hand in order to compile the model.

Another problem occurs when a model for function pointers has to be created. There is a special construct that supports this, however, only one function can be linked to this model as pointed out by the Coverity support in Appendix C. That is why it is not possible to use the function pointer for multiple functions. The model is more like an alias to the actual function.

As a result of not modelling the heap, Coverity does not always detect reading of uninitialized memory. The example bug in Listing 4.2 is not found. The printf() in Line 13 reads the variable d->b which is not initialized as there is only one integer in the smaller struct One.

```
1  struct One {
2      int a;
3  };
4
5  struct Two {
6      int a;
7      int b;
8  };
9
10 struct One *c = malloc(sizeof(struct One));
11 struct Two *d;
12 d = c;
13 printf("%i", d->b);
```

Listing 4.2: Different struct sizes

There is another problem that arises from the limited value tracking of variables. Coverity does not detect when two pointers point to the same memory block. When Listing 4.2 is extended by free(d);free(c);, then this is not detected as a bug. The memory block is freed twice, but Coverity only sees that two different variables are freed.

4.1.3. Commandline Usage

The Coverity analysis is started from the command line. The commands are straight forward. A Coverity model is created from a C-file and is stored in a special XML format.

In a second step, the build process is monitored by Coverity and it creates information files about it. Next, the analyzer engine is executed under usage of the previously created model file. At the end, the number of bugs found is shown. As a final step, these can be committed to the Coverity web interface, where the bug reports can be viewed and further processed.

```
cov-make-library -of usermodel.xmldb usermodel.c
cov-build --dir $i/.cov/ make install check
cov-analyze --dir $i/.cov/ \
    --user-model-file usermodel.xmldb
cov-commit-defects --host localhost --stream $stream \
    --user admin --dir $i/.cov/
```

4.2. Clang

Clang Static Analyzer (henceforth referred to as Clang) is a compiler frontend to the LLVM Compiler Collection. Clang builds an Abstract Syntax Tree (AST) using the compiler features of LLVM. It saves the state of each variable using an abstract memory model and then uses it for symbolic execution during analysis. In contrast to Coverity, there is no static call graph structure, but every possible execution path is simulated. All this does not find any bugs yet. However, this is the framework for the actual checkers that are written as modules for Clang. By using this modular approach, Clang can be easily extended by writing custom checkers. The Clang documentation at [12] shows how this can be done. During the path traversal each checker is informed about the current state. In this thesis I do not explore this feature; as mentioned in Chapter 8 it is left as future work.

In contrast to Coverity, Clang precisely tracks the memory allocations and variable assignments throughout the whole analysis. Coverity uses a much less in-depth method for this process. This supports the aim of Coverity to be fast at the analysis. Dawson Engler's research at Stanford University [6] showed that this is not necessarily a drawback in respect to bug finding precision.

The goal of Clang is to track the value and refinements of variables. A simple value store per variable could be used like:

$$Environment = Variable \rightarrow Value$$

This is not sufficient for the flexible memory model of C. By the use of pointers, the language allows multiple variables to point to the same data. So when one variable is edited, Clang has to infer the change of the other as well. In short, variables need to map to locations, which then contain the actual abstract data representation.

$$Environment = Variable \rightarrow Location$$

$$Store = Location \rightarrow Value$$

In the actual implementation, another model is used that can simulate the C syntax precisely. This advanced memory model is called region based memory model and is described in [13].

$$Environment = Expr \rightarrow SVal$$
$$Store = Region \rightarrow SVal$$

A *SVal* can either be a memory location a concrete value, or a symbolic value. Symbolic values are needed for the arguments of a function. The actual value is only known later during the analysis run. Therefore the value is first stored as a placeholder. All further expressions containing this value are symbolic as well. For example, when the symbolic value of `x` is multiplied by `2`, then the result would be stored as a `SymIntExpr` object that references the symbol of `x` and the integer `2`.

The *Environment* contains all expressions that occur in the source code. Each of them has an associated unique SVal object. For example the SVal object for the expression `2` would reference a `ConcreteInt` object with the value of 2. For variables, a distinction has to be made between usage as *locator value* (short lvalue) and *value of an expression* (short rvalue) [1, p. 46]. The SVal object associated to lvalue usage always references a memory region. The rvalue can be a memory region or some kind of value. For example a pointer would have a memory region as rvalue, whereas an integer has a symbolic or concrete value assigned. Clang does not always know concrete values, but it can infer constraints on the value. An `if` construct can give additional information to the analyzer. For example, the following code tells that `x` is greater than `0` in that branch: `if (x > 0) then foo(x);`. This value is then stored as a `SymbolVal`. All the possible datatypes in Clang are documented in [14].

Memory regions can also be symbolic, similar to symbolic values, if there is no information about it yet. When the value of a variable is needed, then the Store is queried for the associated Region. Each region saves some information, such as size and location. The location is hierarchically implemented. There are base regions, such as stack and heap space. Furthermore, each array region has sub regions for each element. This way, each element has a distinct SVal and also knows that it belongs to an array. This is, for example, useful for checking out of bound conditions.

4.2.1. Different Checkers

The call graph and variable tracking described in the previous section, cannot find any bugs yet. Clang first has to know which program flows it should mark as problematic. This is what checkers are for. Checkers are implemented specifically for one kind of bug. They have access to the data structure that Clang has built. This information is then used to find common programming errors. There are different kinds of checkers. Core checkers test for common errors such as “division by zero” or “null dereference”. There are also platform specific checkers for Unix that test for correct usage of `malloc()` and `free()` in order to prevent “double free” and “use after free” problems. Furthermore, there are OSX

specific checkers that test for correct usage of the Cocoa API. There are also some useful checkers that are still in an alpha phase.[12]

For example, Clang has a checker that can analyze wrong usage of variable casts. The "CastToStruct" checker detects when a pointer that does not refer to a struct is cast to a pointer that does. The "BoolAssignment" checker ensures that only values of 0 or 1 are assigned to a boolean variable. In addition, the core "NonNullParamChecker" detects when memory is read that has not been allocated yet. For `struct` allocation, there is also a "CastSize" checker for assuring that `malloc()` is called with the correct size, that is a multiple of the `struct` size.

4.2.2. Restrictions of Clang

The checks work in general; however, most of the pointer related checks do not work when the pointer is cast to a void pointer in between. For example, the "CastToStruct" does not complain when an `int*` is cast to a `void*` pointer and then to a pointer to a `struct`. The only check that still works through void casting, is the "NonNullParamChecker". Listing 4.2 shows an example with two structures of different sizes. Clang does not detect anything wrong at Line 12, but only at Line 13 when the function tries to read memory that has not been allocated before. This works because Clang has a very exact memory model that stores the sizes of all allocated memory regions.

When analyzing bigger projects like GNUnet, the "CastToStruct" checker crashed Clang at the time of writing. It is still marked as alpha, so it may well work in the future.

The main drawback of Clang is that it does not support analyzing across translation units. When a *C* project consists of multiple TUs, a separate Clang process is executed for each of them. No data is shared between those instances. There is a script called "scan-build" that combines the resulting reports of these instances and creates an index list. It also removes duplicates that can arise through bugs in header files [2]. One possible solution on how this can be mitigated is shown later in Chapter 6

4.2.3. Commandline Usage

There are different ways to start the analysis with Clang. Single source files can be analyzed with the `clang` command.

```
clang --analyze -Xanalyzer -analyzer-output=html test.c
```

With this command, the reports are output as html files. Different checkers can be activated or deactivated.

There are also some commands that can improve the precision of the analysis, but they also increase the analysis time. The following `max-nodes` argument tells Clang to stop

[2] `http://llvm.org/bugs/show_bug.cgi?id=16809#c2`

analyzing longer paths. So it can learn more about the call graph and use this information for the analysis.

```
clang --analyze -Xanalyzer -analyzer-output=html -Xanalyzer    \
 -analyzer-stats -Xanalyzer -analyzer-config -Xanalyzer        \
 max-nodes=300000 -I . -I ../libevent/include/ -I ../libevent/ \
 total-d.c  -o ./outputhtml-togetherd10
```

For scanning a whole project, Clang offers the `scanbuild` command[3], which is comparable to the Coverity `cov-build`, and `cov-analyze` commands. The options are similar to the `clang` command; however, there are some slight differences.

```
scan-build -enable-checker alpha.core.CastToStruct -plist-html \
 -analyzer-config max-nodes=300000 gcc test.c
```

The report can then be viewed with the `scan-view` command. A browser is automatically started with the index page of all bug reports that have been created.

4.3. Comparison

Coverity and Clang both yield similar results. They find errors like division by zero or problems with heap allocated memory. Due to the differences in their architecture, they have different usage scenarios: Coverity is trimmed for speed, at the cost of its soundness, so that it can be invoked on a regular basis, for example for every commit, to find bugs quickly. This is achieved by a more shallow analysis of the programs execution and good heuristics. Only program parts with possibly problematic constructs are analyzed [15]. Clang, on the other hand, has a very precise memory model and call hierarchy and looks at every possible path in the program. Each checker is informed about all the execution steps and can use the information for its analysis.

Coverity has limited ability to reason about values that are stored in variables. The actual content is not stored for any kind of pointers. Because this was needed for this thesis, it was necessary to use Clang for the analysis as well. The restriction on inter translation unit analysis of Clang can be circumvented with some work which is discussed in Chapter 6.

The amount of documentation available differs as well. As a commercial software, Coverity has an extensive and cleanly structured online documentation. Clang, on the other hand, has less information available, probably due to its open source nature with multiple independent developers involved. The homepage offers a good overview and some information about possible use cases. However, it is often necessary to look for further information in the mailing list archives or the documentation generated from the source code. For both tools, Coverity and Clang, there are some papers on the actual implementation that have been referenced throughout this chapter.

[3] `http://clang-analyzer.llvm.org/scan-build`

5. Approach

The problem with the analysis of asynchronous events is that they can occur at an arbitrary point in time. As a result, the static analyzer tool does not know when this callback function will be executed. In a dynamic execution this can actually happen at different points in time depending on when the event fires. Hence, in static analysis this dynamic behaviour cannot be modelled, because each analysis run has to be deterministic. In the beginning I planned to let the static analyzer save the events in the queue. However, this was soon discarded as neither Coverity nor Clang were able to track function pointers or other data in global variables over multiple functions.

For this thesis, a simplified model of execution was chosen: the actual execution of the callback function already occurs when the event is added to the eventloop queue. Figure 5.1 shows the simplified execution order. There are no longer any events that need to happen. In order to change the behaviour of the add function, several approaches have been taken and will be explained in the following.

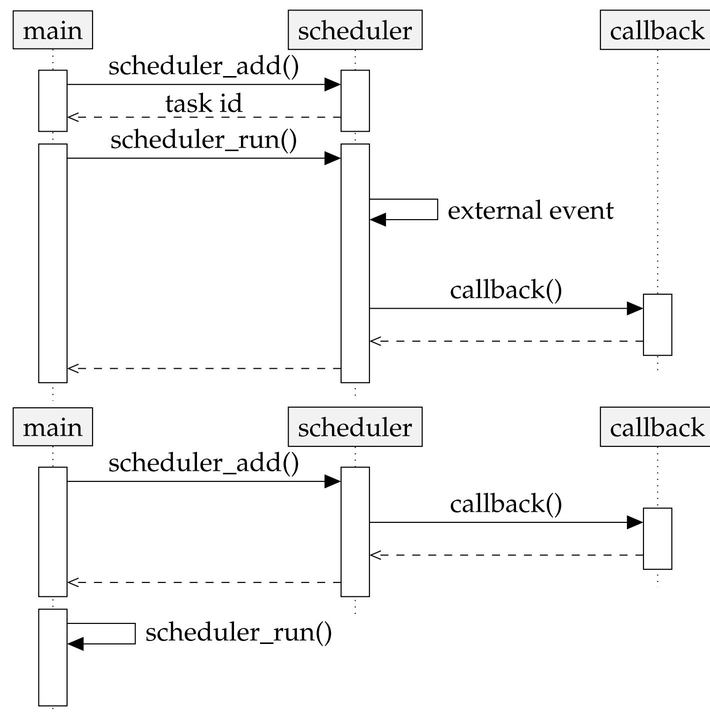


Figure 5.1.: Changed event callback

This simplified model for adding callbacks could potentially produce more false positives because the `scheduler_cancel()` function is not taken into account. Listing 5.1 shows a code where no "double free" bug occurs. However, the static analysis does not know that the task is cancelled and thus it infers that the pointer is freed twice.

```
1  void callback(void* closure) {
2      free(m);
3  }
4  void main() {
5      void* closure = malloc(1),
6      int task_id = scheduler_add(&callback, closure);
7      free(closure); // ok since the task is cancelled
8      scheduler_cancel(task_id);
9      scheduler_run();
10 }
```

Listing 5.1: False positive

This chapter explores how this analysis can be done with Coverity and Clang. The tests were executed with the sample programs GNUnet and libevent. The obtained results are then shown in Chapter 7.

```
1  int scheduler_add(void (*task) (void *closure), void *closure);
2
3  void callback(void* closure) {
4      free(closure);
5  }
6
7  int main() {
8      void* m = malloc(sizeof(int));
9      scheduler_add(&callback, m);
10     free(m);
11 }
```

Listing 5.2: Double free bug

5.1. Use Model for Coverity

The commercial Coverity analysis tool offers a built-in method to change the behaviour of certain functions. These models are source code files that contain a redefinition of the original function. During the analysis run this model can be passed to Coverity. Its intended use case is to teach Coverity how certain library functions behave where the source code is not available. However, it is also a viable solution for the problem at hand.

This approach was tested with GNUnet. Listing 5.3 shows a model for the scheduler function that adds a callback to an event queue. A Coverity model is just a normal C-file with a function definition that is used during analysis instead of the original one. Modelling the scheduler function was difficult because it uses datatypes that are specific to GNUnet. When Coverity parses a model file, it does not define default datatype macros like GCC. For example `#define __UINT8_TYPE__ unsigned char` has to be defined manually. After adding all necessary includes, the model can be parsed.

```
1 GNUNET_SCHEDULER_TaskIdentifier
2 GNUNET_SCHEDULER_add_now (GNUNET_SCHEDULER_Task task, void *task_cls) {
3     struct GNUNET_SCHEDULER_TaskContext queue_macro_tc;
4     queue_macro_tc.reason = 0;
5     (*task)(task_cls, &queue_macro_tc);
6     GNUNET_SCHEDULER_TaskIdentifier a = 1;
7     return a;
8 }
```

Listing 5.3: Scheduler model for GNUnet

Implementing this model in Coverity did not yield the expected result. No additional bugs or false positives were found. This leads to the conclusion that Coverity is not able to store function pointers in variables and that they cannot be passed to the model function. This prevents Coverity from generating the complete call graph.

5.2. Patching the Source Code

As the model approach did not work, another method was needed. There is one possible way to call a function pointer that Coverity understands: it has to be called directly via `(*(&handler))(data);`. So instead of having an external model file, the source code of GNUnet was patched. The function call to the scheduler was replaced with a macro function that directly executes the callback. That way there is no need for storing the function pointer in a variable. Listing 5.4 shows the necessary macro function. As a result, the output of the preprocessor only contains calls to the scheduler function that are supported by Coverity. Coverity can now analyze calls to the scheduler function. To see which bugs are found by this method, the analysis was first started without the patch and then again with the patch applied. The additional bugs could then be related to the improved analysis of the scheduler function.

```
1 static struct GNUNET_SCHEDULER_TaskContext queue_macro_tc;
2 static GNUNET_SCHEDULER_TaskIdentifier task_identifier = 1;
3 #define GNUNET_SCHEDULER_add_now(task, task_cls) ( \
4         queue_macro_tc.reason = 0, \
5         (*(task))(task_cls, &queue_macro_tc), \
6         task_identifier \
7 )
```

Listing 5.4: GNUnet scheduler macro function

5.3. Analysis with Clang

Clang does not support a model mechanism like Coverity. Therefore, the only possibility is to change the source code itself. This patch works like the Coverity model by calling the callback directly and not putting them in a queue. Alternatively the macro function from Listing 5.4 could be used as well.

The patch for GNUnet is shown in Listing 5.3, the “scheduler.c” file has been changed. It is the same code a used before in the model for Coverity, the callback function `task` is called directly. The Clang analysis engine can then trace the variable propagation to the callback function.

5.4. Problem

A drawback of Clang is that it can only consider one translation unit at a time. This means that function calls are only followed inside of one file. When a function of another file is called, then the analyzer cannot use this information for its analysis. There are some bug reports for Clang that ask for this feature, but as of now there are no plans to implement this [1]. This feature is also necessary for this approach, because the scheduler function is typically in a separate file. As a result, Clang fails to reason about the callback functions even when the patch to the scheduler function is applied. The separate file is a barrier to the analyzer.

In order to solve this problem, the function for adding an event to the scheduler has to be in the same translation unit as the calling function. The first approach to solve this was to implement the function in the header file, then each translation unit has its own scheduler function. This already increases the amount of analysis that can be done. However, this does not work when the calling function wants to add a callback function from another file to the scheduler. Then the same restriction applies: when Clang analyzes the calling function, it can only follow the call to the scheduler function. Next, it does not know about the definition of the callback function at that time and cannot follow it anymore.

5.5. Aggregate Multiple Source Files

The chosen approach in this thesis is to combine all translation units (TU) into one. The previous approach only combined the scheduler function with each source file separately. The basic idea is to have one global TU for the static analyzer to look at, so it has all the function definitions during the analysis and can use the knowledge for analyzing the complete program flow.

After aggregating the source files, inter translation unit function calls can actually be tracked. However, this technique is not yet fully automated. The following Chapter 6 shows the approach taken for this thesis. By applying this technique, the static analyzer is able to analyze callback functions and their corresponding closure. The results obtained are shown in Chapter 7.

[1] `http://llvm.org/bugs/show_bug.cgi?id=16809#c1` and `http://llvm.org/bugs/show_bug.cgi?id=18209#c1`

6. Source Code Aggregation

As described in the previous chapter, Clang cannot track function calls over multiple Translation units. The tool can only analyze on a file by file basis. However, the function that handles event callbacks is usually in a separate file or even in a separate library like libevent. Also the callbacks can be in a different file. The solution is to combine all the necessary files into one huge C-file and run the analyzer on that. However, in contrast to recent languages as C# or Python, C relies on the separate files to model private members (variables, functions and data structures). By combining multiple files, all of these private members get into conflict with each other because they are now visible to every function in the program. It is no longer possible to have private members.

The solution chosen for this thesis is to rename all the static members with a prefix so that every C-file regains its private members for itself. The cleanest method would be to write a C parser to find the uses and semantics of all identifiers, but for the sake of simplicity the method used here employs regular expressions for the renaming process.

The following describes the usual build process for the Automake build system and which problems have to be taken care of so that multiple files can be combined into one.

6.1. Structure of Automake Files

The script that handles the renaming process only supports the GNU Automake system. These have a more structured way and can be more easily read than regular Makefiles because Automake abstracts all the platform specific code away. This section is a short introduction to the format of Makefile.am files.

Automake files have a simple base structure. All data is defined as variables. Some variables have a special meaning and get interpreted by the Automake program. Other custom variables can be defined by the user. Also simple control flow structures with `if` are supported. Listing 6.1 shows how a small `Makefile.am` could look like. The variable ending with "_PROGRAMS" defines which programs should be compiled. A program has a list of associated source code files. The variable ending with "_SOURCES" defines the source files associated with one program, where the first part of the variable is the program name. Other variables exist for documentation. Furthermore, different methods to structure the Makefile in different files are supported. The first method is to have separate `Makefile.am` files in each sub directory and to reference them with the `SUBDIRS` variable. The second method is to include files with the `include` command. This way all the variables of the included files propagate to the main file. [16]

```
1 SUBDIRS = src/util
2
3 bin_PROGRAMS = hello
4 hello_SOURCES = main.c
5
6 include src/doc/include.am
```

Listing 6.1: Sample `Makefile.am`[16]

6.2. How Normal Builds Work

The compilation of C-files is a multi-step process as shown in Figure 6.1. First the file gets preprocessed by the preprocessor, and macros are expanded. Also in this step, the included header files are inserted. Next the c-file gets compiled by the compiler (gcc or clang) and converted into an object file. So far the process happens independently for each file. As soon as all the files for an executable have been compiled, they will be linked together by the linker program. This replaces external function and variable references by the actual position in the resulting object file. During this process, the program gets linked with external libraries as well. This step can fail when some symbol that has only been declared but never defined is not found. This can be due to some library that was forgotten to add to the link process.

Figure 6.1.: Compilation process

6.3. Aggregating TUs

As described earlier in Section 5.5 the concept of multiple files in C enables a private scope per file. Only the public interface can be accessed by other files and this feature gets lost with the aggregated file. Another problem is posed by the fact that every header now only gets included once. Some projects like Tor use macro defines to enable certain parts of a header file, like internal data types. A problem arises if the file gets included normally the first time, and the second time in the extended mode. It is common practice to prevent multiple inclusions of a header file via a `#ifndef` macro construct called header guard. On first normal inclusion, the header guard is activated, then the second time when the macro for the extended mode is set, the internal data types are not included. To circumvent this, the defines have to be set before starting the compilation.

When there is only one source file, then there is no need to link the object files together as all own files are combined in one. However, the result still has to be linked to library files. When a library should be analyzed as well, it has to be aggregated as well. For example, this is necessary to analyze projects that use the libevent library.

6.4. Parsing Makefile.am

Automake is a platform independent abstraction of the normal Makefile. For my purpose of aggregating multiple source files, this format is easier to parse as it is more structured and does not contain platform specific code.

The language chosen for this purpose is Python. It is a simple language that has some powerful programming constructs. It also offers good support for regular expressions that I use for parsing the Makefile and later on also C-files. The whole script is shown in Appendix A.

In the Makefile syntax, lines can be continued with a backslash. Listing 6.2 shows an example. My script wraps the Python readline function, so by using this function, the actual algorithm does not need to care about line continuation. Also the `include` command is directly implemented here. The Automake file is set up with different macros and is described with environment variables. Each line either defines a variable or contains some basic control flow command. The variables can also be used to define new variables.

```
1  noinst_PROGRAMS = \
2   gnunet-config-diff
3
4  noinst_PROGRAMS += $(W32CAT)
5
6  W32CAT = w32cat
```

Listing 6.2: Makefile.am variables

Variables are assigned with the = operator. In addition, an append operator += is supported as well. Variable values can be used to define new variables using the `$()` operator. The variable assignment uses lazy evaluation. This means that only when the variable is used, the value of the variables on the right hand side are evaluated. In the example in Listing 6.2 the value of `noinst_PROGRAMS` is "w32cat" when the variable is used by Automake.

6.4.1. Control Flow Conditions

In the standard setup with the `configure` script and Automake, different variables determine the build process. These variables are then saved in the `config.status` file. The script parses this file and adds the necessary flags to the environment before beginning to parse the Makefile.am. This means that the `configure` script has to be run before the aggregation.

Automake supports basic if-else structures which test if a given variable is set or not. Due to the macro nature of Automake, the encoding of these status variables is not straight forward. Listings 6.3 to 6.5 show the transformation of an Automake if clause to an equivalent statement in the Makefile. The necessary variable with a `_TRUE` and `_FALSE` ending are stored in the `config.status` variable. `VARIABLE_TRUE=` and `VARIABLE_FALSE=#` mean, for example, that the if block is executed and the else block is commented out. The

script for this thesis does not apply macros as Automake does, but instead it directly interprets the if statement and ignores branches that would be commented out.

```
if TEST
  noinst_PROGRAMS = test
else
  noinst_PROGRAMS =
endif
```

Listing 6.3: Control flow in Makefile.am

```
@TEST_TRUE@noinst_PROGRAMS = test
@TEST_FALSE@noinst_PROGRAMS =
```

Listing 6.4: Automake result

```
noinst_PROGRAMS = test
#noinst_PROGRAMS =
```

Listing 6.5: After `configure.sh` when `TEST` is true

The programs and source variables are then used to generate a list of files. Then all file contents are simply concatenated into one file by applying the variable renaming algorithm described in the next section.

6.5. Prevent Naming Conflicts

The next step is to prevent identifier name conflicts. These occur because there is no separate scope per translation unit anymore when all the source files are aggregated. My pseudo-parser works by distinguishing between public and TU local identifiers. All public identifiers have to be declared in a header file, otherwise they cannot be accessed from another file. All variables and functions that are declared with the `static` keyword have a scope limited to the TU. This means they are only accessible from the current file and cannot be part of the public interface[1, p. 30]. The static declarations are detected by a regular expression. For the declarations this works well because the `static` keyword can be reliably detected. The actual renaming of occurrences of the function name is more difficult as the same string could also be present in, for example, a structure definition. Some more complex regular expressions have been used to filter this out. But the algorithm still cannot detect all possible cases that are allowed by the C-standard. A real parser would be necessary to solve this. Data types, such as structures, enumerations, and unions, are not declared static and consequently more difficult to detect. I analyze the included header files and look for the structure identifiers there. All structure definitions that are not found in the header files are then considered to be local to that file.

After all the occurrences have been found, the actual renaming is the same for all identifiers. It is important that all occurrences are found, because otherwise the program will not compile anymore; this is ensured by the regular expressions below. The goal for the renaming algorithm was to have unique names for each local variable. Therefore, the file name is used as a prefix for the identifier. For example, `int counter = 0;` in the file "src/network/protocol.c" would get renamed to `int src_network_protocol_c_counter = 0;`. This is not an identifier that would be used in a program so there are no new naming conflicts introduced by this naming scheme. Another way is to hash the file path, but this

removes the traceability from the bug report. With the first method it is possible to see which file an identifier belongs to.

Identifiers in C only allow a restricted amount of different characters [1, p. 51]. The regular expression syntax of Python is used here for the peudo parser. The identifiers can be modelled with a regular expression as follows:

```
(?P<identifier>[a-zA-Z_][a-zA-Z_0-9]*)
```

It may only start with a non-numeric letter, followed by any alphanumeric character. The bounds of any identifier in the document are then defined by an arbitrary other character. All the following regular expressions are based on this identifier pattern. For the different use cases like function or variables, different restrictions have to be used on the text before and after the identifier.

6.5.1. Renaming Data Types

The regular expression to represent structure and other data types is the same as for a normal identifier, there is only a prefix prepended, separated with one or more spaces.

```
(?P<identifier>(?:enum|struct|union)\s*[a-zA-Z_][a-zA-Z_0-9]*)
```

```
(?P<identifier>(?:enum|struct|union)\s*[a-zA-Z_][a-zA-Z_0-9]*)\s*
(?:\{|;)
```

This regular expression finds all structure declarations and definitions. It starts with the identifier part from above. Then, after the identifier, either a semicolon has to follow for a declaration or a curly brace for a definition. Due to the `struct` keyword this detection is very reliable and does not cause any problems during parsing. Next, all data structures that have not been defined in any header file are renamed. The renaming of all the occurrences is straight forward as well as discussed earlier.

6.5.2. Renaming Variables and Functions

As a next step, all variables and functions that are defined as static will get renamed. The corresponding regular expression searches for all `static` keywords at the beginning of a line. Otherwise also static variables inside of functions would be matched, but which do not need to be renamed. The next part is the data type of the variable. The same characters as for identifiers are allowed. A star character is also matched for pointers. Then follows the expression for the variable name. There are different possibilities for the final part that delimits the identifier. If it is a variable definition then it can end with a semicolon, an equal sign, when a value is assigned directly, or a square bracket for arrays. A function identifier is always delimited by an opening parentheses.

```
\nstatic[\sa-zA-Z_0-9]+(?: |\*|\n)+(?P<identifier>[a-zA-Z_]
[a-zA-Z_0-9]*)\s*(?:;|=|{|\[|\()
```

In order to find the usage occurrences of the static variables or function, the following regular expression is used:

```
(?P<before>(?<=\-[^_a-zA-Z\.\>]|[^\-][^_a-zA-Z\.]))(?P<identifier>
[a-zA-Z_][a-zA-Z_0-9]*)(?P<after>[^a-zA-Z_0-9])
```

The first part ensures that only variable occurrences are found and not members of structures which could have the same name. The members are either accessed by a dot or by `->`, so these cannot be in front. The end of the variable name is delimited by any character that is not valid in an identifier.

6.6. Encountered Problems

There was one kind of macro from libevent that caused problems with the renaming script. Listing 6.6 shows a macro function that takes a type as parameter. Inside the macro body, this type is prepended with `struct`. When my renaming algorithm wants to find all occurrences of `struct evmap_signal`, it does not detect the usage in this macro function as only the term `evmap_signal` is used in Line 4. To fix this, I excluded these structures from renaming.

```
1 #define GET_SIGNAL_SLOT(x, map, slot, type) \
2     (x) = (struct type *)((map)->entries[slot])
3
4 GET_SIGNAL_SLOT(ctx, map, sig, evmap_signal);
```

Listing 6.6: Libevent macro

Another problem occurred when compiling Tor. Some headers only execute declarations when a special macro definition is set. For example, the header file in Listing 6.7 requires `#define ADDRESSMAP_PRIVATE` to be defined in the source file that includes the header. In order to ensure that always all parts of the header files are included, these macro definitions are passed to the compiler via the `-D` flag.

```
1 #ifdef ADDRESSMAP_PRIVATE
2 typedef struct virtual_addr_conf_t {
3   tor_addr_t addr;
4   maskbits_t bits;
5 } virtual_addr_conf_t;
6
7 STATIC void get_random_virtual_addr(const virtual_addr_conf_t *conf,
8                                     tor_addr_t *addr_out);
9 #endif
```

Listing 6.7: Tor header "src/or/addressmap.h"

In some GNUnet source files there are members of `struct`s and functions that have the same name. Listing 6.8 shows an example. When renaming the function, it has to be ensured that either all occurrences of the member identifier are renamed as well, or none. For this example it would work to make the renaming regular expression from Section 6.5.2 more general and rename all occurrences. However, there are also header files with structure definitions that cannot be renamed. So the only possibility is to not rename these structure members. The uses of the member are easily detected, but it is more difficult to check whether an identifier is inside of a structure definition. Therefore all structures are analyzed in the beginning and the regular expression for renaming can then test whether the current line is part of a structure and exclude it from renaming. Appendix D shows the necessary commands for excluding these identifiers.

```
/**
 * Connection to the NAMECACHE service.
 */
struct GNUNET_NAMECACHE_Handle
{
...
  /**
   * Reconnect task
   */
  GNUNET_SCHEDULER_TaskIdentifier reconnect_task;
  // This would be prefixed as bysrc_namecache_namecache_api_c_reconnect_task

  ...
}

static void
src_namecache_namecache_api_c_reconnect_task (void *cls,
                    const struct GNUNET_SCHEDULER_TaskContext *tc);
```

Listing 6.8: GNUnet struct

6.7. Drawbacks

While renaming is automated, manual intervention is still needed due to the shortcoming of not using a parser. One solution would be to make the script significantly smarter and practically turn it into a c-compiler of its own so that it understands the semantics and can differentiate between different identifiers. As the current script is only using regular expressions in order to find identifiers, it cannot handle variables that get declared in different scopes. This would also help to understand the functionality of macros. An approach to solve this is outlined in Chapter 8

Furthermore, not all programming paradigms are supported at the moment. Different dynamic libraries like services in GNUnet that usually get included dynamically at runtime cannot be examined by the method described here. To mitigate this, the loading of the services would have to be programmed differently for this compile setup.

7. Results

The approach described in Chapter 5 was first used to analyze the GNUnet project. The libevent project offers a portable event loop functionality as a library for other projects. Some projects were chosen for the analysis. Tor provides anonymous internet access and uses the libevent library. tmux is an alternative to screen program and will be analyzed as well.

7.1. GNUnet

First, Coverity was used to analyze GNUnet with the macro function mentioned in Chapter 5. There was, however, no difference between the analysis with the patched scheduler and without. Both scans resulted in the same amount of bugs found.

In order to test whether the approach works, a bug was introduced intentionally. Figure 7.1 shows one example of a Use After Free Bug. The `plugin` variable is passed as closure to the scheduler function. After the callback execution, the variable has been freed, and as a consequence the next usage of the variable is marked as a bug by Coverity. This shows that Coverity was able to generate a Call Graph via the macro function for `GNUNET_SCHEDULER_add_select()`.

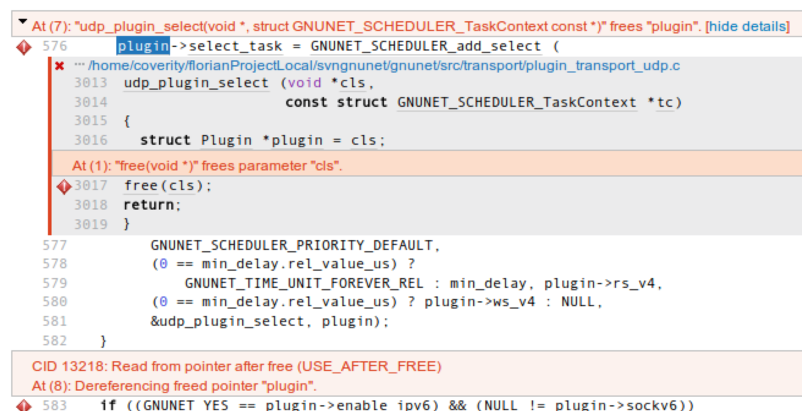


Figure 7.1.: Bug with incorrect scheduler function

As the first approach with Coverity models did not produce any new results, GNUnet was next analyzed with Clang. The scheduler function of GNUnet, located in "src/util/scheduler.c", was patched in order to gain a synchronous control flow for the static analysis.

All source files of GNUnet were then aggregated using the script in Appendix A. After executing Clang's `scan_build` command, the effects of the patch could be observed.

Three additional bugs were found, but after examining them, they turned out to be all false positives. It showed, however, that the method applied actually works. Figure 7.2 shows one of the bugs. The reason for the false positive was that Clang did not correctly consider the copying in Line 177631. The following scheduler call to `_src_fs_fs_publish_ksk_c_publish_ksk_cont` detects a null pointer dereference. The long function name results from the aggregation in Chapter 6. This is not a real bug, because the condition in Line 171849 is true. The problem is that Clang does not know that `pkc->i` is equal to zero. This false positive could be prevented by explicitly copying each field of the structure instead of using `memcpy()`.

```
171914    pkc->ksk_uri = GNUNET_FS_uri_dup (ksk_uri);
              13  ← Calling 'GNUNET_FS_uri_dup' →
177631      ret = GNUNET_new (struct GNUNET_FS_Uri);
177632      memcpy (ret, uri, sizeof (struct GNUNET_FS_Uri));
177643        if (ret->data.ksk.keywordCount > 0)
              18  ← Taking false branch →
177650        else
177651          ret->data.ksk.keywords = NULL;     /* just to be sure */
              19  ← Null pointer value stored to field 'keywords' →
              21  ← Returning from 'GNUNET_FS_uri_dup' →
171915    pkc->ksk_task = GNUNET_SCHEDULER_add_now (&_src_fs_fs_publish_ksk_c_publish_ksk_cont, pkc);
              22  ← Calling 'GNUNET_SCHEDULER_add_now' →
171849      if ( (pkc->i == pkc->ksk_uri->data.ksk.keywordCount) ||
              25  ← Taking false branch →
171850           (NULL == pkc->dsh) )
171851      {
171852        GNUNET_log (GNUNET_ERROR_TYPE_DEBUG, "KSK PUT operation complete\n");
171853        pkc->cont (pkc->cont_cls, pkc->ksk_uri, NULL);
171854        GNUNET_FS_publish_ksk_cancel (pkc);
171855        return;
171856      }
171857      keyword = pkc->ksk_uri->data.ksk.keywords[pkc->i++];
              26  ← Array access (via field 'keywords') results in a null pointer dereference
```

Figure 7.2.: False positive with Clang

As a side effect of the source code aggregation, some minor mistakes were found as well. In some header files the header guard was missing in GNUnet. The normal build process succeeded nevertheless because those files were never included multiple times. As a result of the aggregation, this posed a problem because multiple C-files included this header and so there were redefinition errors.

Coverity is trimmed for speed as mentioned in Chapter 4 and was able to prove it in my analysis. Checking GNUnet took about 10 minutes, whereas Clang needed about 20 minutes to analyze the aggregated source file.

7.2. Libevent

Libevent is an opensource event notification library that is used in different projects like Chrome, Tor, and tmux (an alternative to screen). It offers similar functionality as the scheduler in GNUnet. Therefore it is also subject to the same problems with static analysis tools. By applying the same approach as before, the idea is to find more bugs related to the callback functions.

Libevent offers a structure for each event. Different kinds of events, such as timeouts or IO callbacks, can be modelled in this structure. It is created with the `event_new()` function first, and then it is passed to one general `event_add()` function. The `event_add()` function is then modified like the GNUnet scheduler function. Listing 7.1 shows the modification to the libevent library. The callback function is executed directly instead of adding it to a queue. Because the callback function is stored in the `struct event` and not passed directly to the scheduler function as in GNUnet, this means that Coverity cannot be used for the analysis. As shown in Chapter 4, it does not track function pointers that are stored in variables. Therefore, only Clang was used to analyze libevent programs.

As before, the `event_add()` function (corresponds to the scheduler function of GNUnet) has to be in the same translation unit as the calling function. Otherwise Clang could not follow the calls. Therefore the libevent library and the project that includes it are combined into one single C source file which can then be analyzed.

First, a simple test program has been written to test whether this concept also works for libevent. Then, the two opensource projects Tor and tmux have been analyzed. The results are shown in the following.

```
1  int
2  event_add_nolock_(struct event *ev, const struct timeval *tv,
3      int tv_is_absolute)
4  {
5      struct event_base *base = ev->ev_base;
6      struct event_callback *evcb = &(ev->ev_evcallback);
7
8      switch (evcb->evcb_closure) {
9      case EV_CLOSURE_EVENT_SIGNAL:
10         (*ev->ev_callback)(ev->ev_fd, ev->ev_res, ev->ev_arg);
11         break;
12     ...
13     }
14 }
```

Listing 7.1: Libevent patch in `event.c`

7.2.1. Simple Test Program

In a very simple test case, the patch that was applied to libevent could be analyzed using Clang. The Listing 7.2 shows the simple program that was used. For brevity the libevent interface definitions are not shown here.

The main function registers a callback that gets triggered when an event on the file descriptor occurs (in this example file descriptor 0). The callback function then does some division operation with the passed data. In this test case, the passed value is zero, thus the division yields a `Division by zero` error. During a normal run, the callback function `callback()` is called asynchronously with the closure data that was passed to the `event_add()` function. Static analysis tools, like Clang, do not find this bug as they cannot derive that the callback function is ever called with this closure value. The method described previously to make this call synchronous tells Clang what is meant by this code and enables it to draw a conclusion. As a result, the bug report contains the `Division by zero` bug that was added on purpose. The graphical report in Figure 7.3 shows how Clang was able to find the bug. This example shows that the method for patching libevent and aggregating the source files, as developed for this thesis, is able to find bugs.

```
1  #include <event2/event.h>
2
3  void callback(evutil_socket_t e, short s, void * cls) {
4      int *i = (int*) (cls);
5      int b = 6 / *i;
6      printf("%i", b);
7  }
8
9  int main(int argc, char** argv) {
10     struct event_base *base;
11     base = event_base_new();
12
13     struct event *listener_event;
14     int a = 0; //origin of the bug
15     listener_event = event_new(base, 0, EV_READ|EV_PERSIST, callback, &a);
16     event_add(listener_event, NULL);
17
18     event_base_dispatch(base);
19     return 0;
20 }
```

Listing 7.2: Libevent sample program

7.2.2. Tor and tmux

The two open source project tmux and Tor have also been analyzed with Clang, but there were no bugs found, not even false positives. Tor uses libevent only a few times, so those were probably already well tested.

```
void callback(evutil_socket_t e, short s, void * cls) {
    int *i = (int*) (cls);
    int b = 6 / *i;

        9  ← Division by zero

    printf("%i", b);
}

int __main_c_main(int argc, char** argv) {
    struct event_base *base;
    struct event *listener_event;
    int a = 0; //origin of bug

    base = event_base_new();
    if (!base)

        2  ← Assuming 'base' is non-null →

      3  ← Taking false branch →

        return 1;

    listener_event = event_new(base, 0, EV_READ|EV_PERSIST, callback, &a);
    event_add(listener_event, NULL);

      4  ← Calling 'event_add' →

    event_base_dispatch(base);
    return 0;
}
```

Figure 7.3.: Division by Zero bug

8. Conclusion

I started out in this thesis wanting to be able to analyze event driven programs. I succeeded in a way that I can now also follow program traces which involve asynchronous event calls. However, this is done in a very simple way that does not cover all the cases. The main drawback is that there is no real asynchronous call but a direct one. This also means that the static analysis does not have any information concerning the circumstances of the event. For example, the really hard bugs that involve race conditions with asynchronous callbacks cannot be detected. Even so, it is an advantage over standard static analysis as outlined in this thesis.

The method I used here should only be regarded as a prototype and is not ready for general use. The main drawbacks are that many project specific adjustments have to be made. The event loop in the source code has to be changed, and for Clang it is necessary to tweak the renaming parameters for the project to compile as outlined in Section 6.6.

This concept can still be improved in future Bachelor or Master thesises. One way would be to directly hook into the Clang compiler and do all the renaming and combining of different files there. This has the advantage that macros and such can be treated according to their true nature and not only by using regular expressions.

Another possibility would be to change the Clang Static Analyzer to also support multiple translation units. This would offer the greatest flexibility as no more "precompiling" has to be done.

Appendix

A. Aggregation Script in Python

```
#!/usr/bin/env python
"""Parse Makefile.am recursively in the current folder and create one total c-file.
At the moment this only works for gnunet, with gnunet specific hacks.
"""


import argparse
import re
import os
import string
import copy
import collections


def read_make_line(directory, makefile):
    " returns the specified m_file line by line as a generator and concatenates lines with \\ at the end "
    with open(os.path.join(directory, makefile), "r") as m_file:
        while True:
            line = m_file.readline()
            # end of file
            if not line:
                return
            # remove newline character
            line = line[:-1]
            # check if it is a normal line
            if not line.endswith("\\"):
                # check for included files and read them
                if line.startswith("include "):
                    for r_line in read_make_line(directory, line[8:]):
                        yield r_line
                else:
                    yield line
                continue
            else:
                line = line[:-1]

            # concatenate the lines
            while True:
                line_concat = m_file.readline()
                if not line_concat:
                    raise Exception("Makefile.am is corrupted")
                line_concat = line_concat[:-1]
                if not line_concat.endswith("\\"):
                    line += line_concat
                    yield line
                    break
                else:
```

```
48                  line += line_concat[:−1]
49
50  def interprete_makefile(directory, env, excluded_programs):
51      " expects a directory without trailing slash and a dictionary for environment variables"
52      if not os.path.isfile(os.path.join(directory, "Makefile.am")):
53          return []
54      if os.path.isfile(os.path.join(directory, "config.status")):
55          with open(os.path.join(directory, "config.status"), "r") as conf_file :
56              for line in conf_file:
57                  define_re = re.match(r"S\[\"(?P<ident>[a−zA−Z_0−9]∗)\"\]=\"(?P<value>[^\"]∗)\"", line)
58                  if define_re:
59                      if define_re.group("ident").endswith("_FALSE"):
60                          continue
61                      elif define_re.group("ident").endswith("_TRUE"):
62                          if define_re.group("value") == "":
63                              env[define_re.group("ident")[:−5]] = "1"
64                      else:
65                          env[define_re.group("ident")] = define_re.group("value")
66      env.pop("LINUX", None) # needed for libevent
67
68      condition_stack = []
69      for line in read_make_line(directory, "Makefile.am"):
70          # ignore all statements in if blocks
71          if line.startswith("if "):
72              condition_re = re.match(
73                  r'if (?P<negation>!?)(?P<variable>[A−Z0−9_]∗)', line)
74              result = (
75                  len(condition_stack) == 0 or condition_stack[−1]) and condition_re and str(condition_re.
                        group("variable")) in env
76              if len(condition_re.group("negation")) > 0:
77                  result = not result
78              condition_stack.append(result)
79              continue
80          elif len(condition_stack) > 0:
81              if line.startswith("endif"):
82                  condition_stack.pop()
83                  continue
84              elif line.startswith("else"):
85                  condition_stack[−1] = not condition_stack[−1]
86                  continue
87              elif not condition_stack[−1]:
88                  continue
89
90          # search for environment variables
91          assignments_re = re.match(
92              r'\s∗(?P<key>[a−zA−Z_0−9]∗) ∗(?P<add>\+?)= ∗(?P<value>.∗)', line)
93          if assignments_re:
94              key = assignments_re.group("key")
95              value = replace_environment_variables(assignments_re.group("value"), env)
96              if assignments_re.group("add") == "+":# and key in env:
97                  env[key] += " " + value
98              else:
99                  env[key] = value
100             continue
101
102     files = []
```

```
103         programs = []
104         cur_env = collections.OrderedDict()
105         for key in env:
106             # search for sub directories
107             cur_env[key] = env[key]
108             if key == "SUBDIRS":
109                 for subdir in env[key].split():
110                     # ignoring curdir
111                     if subdir == ".":
112                         continue
113                     if subdir == "ats-tests":
114                         continue # Hack to exclude tests
115                     files += interprete_makefile(directory +
116                                                             "/" + subdir, cur_env.copy(), excluded_programs)
117                 continue
118 
119             # search for _PROGRAMS and _LTLIBRARIES
120             program_re = re.match(
121                 r'\s*(?P<progtyp>[A-Za-z0-9_]*)_(?P<type>PROGRAMS|LTLIBRARIES|DEPENDENCIES|
                     LDADD)', key)
122             if program_re:
123                 progs = env[key].split()
124                 if program_re.group("progtyp") == "check":
125                     continue
126                 for program in progs:
127                     programs.append(program)
128 
129         # look for all source files
130         for program in programs:
131             if program in excluded_programs:
132                 continue
133             for p_type in ["", "dist_", "nodist_"]:
134                 key = p_type + re.sub("[^a-zA-Z_0-9@]", "_", program) + "_SOURCES"
135                 if key in env:
136                     for f_name in env[key].split():
137                         f_name = os.path.join(directory, f_name)
138                         if f_name[-1:] == "c" and not 'plugin_transport_template.c' in f_name and not '
                                plugin_transport_udp' in f_name and not "plugin_transport_unix" in f_name and
                                not "plugin_transport_tcp" in f_name and not "/dv/" in f_name and not "psyc/"
                                in f_name and not 'gnunet-service-xdht' in f_name and not 'gnunet-service-
                                set_union' in f_name and not 'gnunet-service-set' in f_name and not f_name in
                                files:
139                             files.append(f_name)
140         return files
141 
142 
143     def replace_environment_variables(text, env):
144         "env: dictionary (key->value)"
145         pos = string.find(text, "$(", 0)
146         pattern = re.compile(r'\$\((?P<key>[^\)]*)\)')
147         while pos != -1:
148             result = pattern.search(text, pos - 1)
149             key = result.group("key")
150             r_text = ""
151             if key in env:
152                 r_text = env[key]
```

```
153             text = text.replace("$(%s)" % key, r_text)
154             length = len(r_text)
155 
156             pos = string.find(text, "$(", pos + length)
157 
158         # check for @ variables in config.status
159         pos = string.find(text, "@", 0)
160         pattern = re.compile(r'@(?P<key>[a-zA-Z0-9_]*)@')
161         while pos != -1:
162             result = pattern.search(text, pos - 1)
163             if not result:
164                 pos = string.find(text, "@", pos + 1)
165                 continue
166             key = result.group("key")
167             r_text = ""
168             if key in env:
169                 r_text = env[key]
170             text = text.replace("@%s@" % key, r_text)
171             length = len(r_text)
172 
173             pos = string.find(text, "@", pos + length)
174         return text
175 
176 
177 def combine_files(cfiles, output_file, args):
178     " combine multiple c-files and replace all static variables, functions, and structs. Call all main functions
            from one global one. "
179     seen_headers = set()
180     with open(output_file, "w") as output:
181         for prepend in args.prepend:
182             output.write(prepend)
183             output.write("\n")
184         main_calls = set()
185         for cfile in cfiles:
186             output.write("// script debug ouput. File: " + cfile + "\n")
187             with open(cfile, "r") as cfile_handle:
188                 # link directory? or delete includes
189                 if cfile[-1:] == "c":
190                     output.write(prefix_identifiers(
191                         cfile_handle.read(), cfile, copy.copy(seen_headers),
192                         main_calls, args))
193                 else:
194                     print "ERROR: unknown file type " + cfile
195         output.write(
196             "int main(int argc, char** argv){\nprintf(\"total c-file\");\n")
197         for main_call in main_calls:
198             output.write(main_call + "(argc, argv);\n")
199         output.write("}")
200 
201 
202 IDENTIFIER_REGEXP = r'(?P<identifier>(?:enum|struct|union)\s*[a-zA-Z_][a-zA-Z_0-9]*)\s*(?:\{|;)'
203 
204 HEADERS_CACHE = dict()
205 def get_identifiers_from_headers(files, include_dirs, seen_headers):
206     " opens all files and included headerfiles recursively. returns all found identifiers "
207     identifiers = []
```

```
208     for hfile in files:
209         if hfile[−1:] == "h" and not hfile in seen_headers:
210             seen_headers.add(hfile)
211             if hfile in HEADERS_CACHE:
212                 identifiers += HEADERS_CACHE[hfile]
213             else:
214                 with open(hfile, "r") as hfile_handler:
215                     content = hfile_handler.read()
216 
217                     # add identifiers
218                     ident = re.findall(IDENTIFIER_REGEXP, content)
219                     if not ident:
220                         ident = []
221                     ident += re.findall(
222                             r'typedef (?P<identifier>(?:enum|struct|union)\s*[a−zA−Z_][a−zA−Z_0
                                    −9]*)\s*[a−zA−Z_][a−zA−Z_0−9]*;', content)
223 
224                     # search the sub headerfiles as well
225                     includes_re = re.findall(
226                         r'#include\s*"(?P<file>[^"]*)"', content)
227                     ident += get_identifiers_from_headers(
228                         search_include_files(includes_re, include_dirs), include_dirs, seen_headers)
229                     HEADERS_CACHE[hfile] = ident
230                     identifiers += ident
231     return identifiers
232 
233 
234 def get_all_identifiers_from_cfile(content, include_dirs, seen_headers):
235     '' extracts all identifiers from one c_file ''
236     includes_re = re.findall(r'#include\s*"(?P<file>[^"]*)"', content)
237     return get_identifiers_from_headers(search_include_files(includes_re, include_dirs), include_dirs,
            seen_headers)
238 
239 
240 def search_include_files(headerfiles, include_dirs):
241     '' searches the location of header files in the file system ''
242     headers = []
243     for hfile in headerfiles:
244         found = False
245         for include_dir in include_dirs:
246             if os.path.exists(os.path.join(include_dir, hfile)):
247                 headers.append(os.path.join(include_dir, hfile))
248                 found = True
249                 break
250     return headers
251 
252 
253 def prefix_identifiers(content, filename, seen_headers, main_calls, args):
254     '' prefix static functions, variables, and data structues ''
255     # not suported yet: typedef, macros
256 
257     # Hack to ignore stuff inside of struct definitions
258     struct_re = re.findall(
259         r"struct [a−zA−Z_][a−zA−Z_0−9]*\s\{[^\}]*\}", content)
260 
261     content = prefix_static(content, filename, main_calls, struct_re, args)
```

```
262
263     content = prefix_data_types(content, filename, seen_headers, args)
264
265     return content
266
267 def prefix_static(content, filename, main_calls, struct_re, args):
268     # replace static variables
269     variables_regex = re.compile(
270         r"\nstatic[\sa-zA-Z_0-9]+(?: |\*|\n)+(?P<identifier>[a-zA-Z_][a-zA-Z_0-9]*)\s
            *(?:;|=|{|\[|\()")
271     variables_re = variables_regex.findall(content)
272     if args.also_rename_static:
273         variables_re += args.also_rename_static
274
275     # always rename main function
276     main_re = re.search(r"\nint\s*main\s*\(", content)
277     if main_re:
278         main_calls.add(create_unique_identifier("main", filename))
279         variables_re.append("main")
280
281     def repl_variable(matchobj):
282         " replace each variable with a prefixed one "
283         identifier = matchobj.group("identifier")
284         if identifier in variables_re and not identifier in args.exclude_rename:
285             # do not change things inside of struct definitions
286             if matchobj.group(0).endswith(";"):
287                 for struct in struct_re:
288                     if matchobj.group(0) in struct:
289                         return matchobj.group(0)
290             return matchobj.group("before") + create_unique_identifier(identifier, filename) + matchobj.
                group("after")
291         else:
292             return matchobj.group(0)
293
294     variables_sub_occ_regex = re.compile(
295         r'(?P<before>(?<=\-[^_a-zA-Z\.\>]|[^\-][^_a-zA-Z\.]))(?P<identifier>[a-zA-Z_][a-zA-
            Z_0-9]*)(?P<after>[^a-zA-Z_0-9])')
296     content = variables_sub_occ_regex.sub(repl_variable, content)
297
298     return content
299
300 def prefix_data_types(content, filename, seen_headers, args):
301     # analyze header files
302     excluded_identifiers = get_all_identifiers_from_cfile(
303         content, args.include, seen_headers)
304     excluded_identifiers += args.exclude_rename
305
306     # Replace struct, enum and union
307     identifier_re = re.findall(IDENTIFIER_REGEXP, content)
308     if args.also_rename_structure:
309         identifier_re += args.also_rename_structure
310
311     def repl_identifier(matchobj):
312         " replace each identifier with a prefixed one "
313         identifier = re.sub(r"\s+", " ", matchobj.group("identifier"))
314         if identifier in identifier_re and not identifier in excluded_identifiers:
```

```
315             return matchobj.group("before") + create_unique_identifier(identifier, filename) + matchobj.
                    group("after")
316         else:
317             return matchobj.group(0)
318 
319     identifier_sub_regex = re.compile(
320         r'(?P<before>[^_a-zA-Z>\.]|^)(?P<identifier>(?:enum|struct|union)\s*[a-zA-Z_][a-zA-Z_0
            -9]*)(?P<after>;|(?=[^a-zA-Z_0-9]))', re.MULTILINE)
321     content = identifier_sub_regex.sub(repl_identifier, content)
322 
323     return content
324 
325 
326 def create_unique_identifier(identifier, filename):
327     " prefix the identifier with the filename "
328     if " " in identifier:
329         # struct enum or union
330         prefix, ident = identifier.split(" ")
331         prefix += " "
332     else:
333         prefix = ""
334         ident = identifier
335     return "%s%s_%s" % (prefix, re.sub("[^a-zA-Z_0-9]", "_", filename), ident)
336 
337 
338 def main():
339     " start intrepretation in the current directory "
340     parser = argparse.ArgumentParser(description="Combining c-files")
341     parser.add_argument("-o", "--output_file", type=str, default="total.c", help="Filename of the output c
            -file")
342     parser.add_argument('--version', action='version', version='%(prog)s 1.0')
343     parser.add_argument('-I', "--include", type=str, action='append', default=[], help="Include
            directories")
344     parser.add_argument("--also-rename-structure", type=str, action='append', default=None, help="
            Also rename these structures")
345     parser.add_argument("--also-rename-static", type=str, action='append', default=None, help="Also
            rename these functions")
346     parser.add_argument("-e", "--exclude-rename", type=str, action='append', default=[], help="Do not
            rename these names")
347     parser.add_argument("--exclude-program", type=str, action='append', default=[], help="Do not
            include these programs in the total file.")
348     parser.add_argument("-m", "--module", type=str, action='append', required=True, help="Project
            folders to combine, default is .")
349     parser.add_argument("--prepend", type=str, action='append', default=[], help="Prepend this text to
            the total c-file.")
350 
351     args = parser.parse_args()
352     env = collections.OrderedDict()
353     files = []
354     for module in args.module:
355         files += interprete_makefile(module, env.copy(), args.exclude_program)
356     combine_files(files, args.output_file, args)
357 
358 main()
```

Listing A.1: Aggregation script

B. Typesystem for Functional Programs

```
<Exp> := <Const> | <Ident > | ( <Expr> ) | <un. Op> <Exp> |
       <Exp> <bin. Op> <Exp> | if <Exp> then <Exp> else <Exp> |
       <Exp> <Exp> | let <Prog> in <Exp> end
```

```
<Type> ::= int | bool | (<Type>) | <Type> → <Type>
```

$\mathtt{T} \vdash \mathtt{e} : \mathtt{t}$ denotes that in the type environment T the expression e has the type t.

$$\frac{\mathtt{T(b)} = \mathtt{t}}{\mathtt{T} \vdash \mathtt{b} : \mathtt{t}} \qquad \frac{\mathtt{k} \in \{\mathtt{true}, \mathtt{false}\}}{\mathtt{T} \vdash \mathtt{k} : \mathtt{bool}} \qquad \frac{\mathtt{k} \in \mathbb{Z}}{\mathtt{T} \vdash \mathtt{k} : \mathtt{int}} \qquad \frac{\mathtt{T} \vdash \mathtt{e} : \mathtt{t}}{\mathtt{T} \vdash \mathtt{(e)} : \mathtt{t}}$$

$$\frac{\mathtt{T} \vdash \circ : \mathtt{t1} \rightarrow \mathtt{t2} \qquad \mathtt{T} \vdash \mathtt{e} : \mathtt{t1}}{\mathtt{T} \vdash \circ\ \mathtt{e} : \mathtt{t2}}$$

$$\frac{\mathtt{T} \vdash \mathtt{e1} : \mathtt{t1} \qquad \mathtt{T} \vdash \circ : \mathtt{t1} \rightarrow \mathtt{t2} \rightarrow \mathtt{t} \qquad \mathtt{T} \vdash \mathtt{e2} : \mathtt{t2}}{\mathtt{T} \vdash \mathtt{e1} \circ \mathtt{e2} : \mathtt{t2}}$$

$$\frac{\mathtt{T} \vdash \mathtt{e1} : \mathtt{bool} \qquad \mathtt{T} \vdash \mathtt{e2} : \mathtt{t} \qquad \mathtt{T} \vdash \mathtt{e3} : \mathtt{t}}{\mathtt{T} \vdash \mathtt{if\ e1\ then\ e2\ else\ e3} : \mathtt{t}}$$

$$\frac{\mathtt{T} \vdash \mathtt{e1} : \mathtt{t1} \rightarrow \mathtt{t2} \qquad \mathtt{T} \rightarrow \mathtt{e2} : \mathtt{t1}}{\mathtt{T} \vdash \mathtt{e1\ e2} : \mathtt{t2}}$$

C. Coverit E-Mail Support

E-Mail from the Coverity support on the modelling of function pointers from Mon, 13 Jan 2014.

```
The purpose of function pointer models is to provide information
to the analysis about function calls (that are performed via
function pointers) for which the analysis cannot track function
pointers properly.

Your usermodel.c does not appear to follow the naming conventions
described in the Coverity 6.0.3 Checker Reference section "4.1.4.
Modeling function pointers", eg,

__coverity_fnptr_<variable>
__coverity_fnptr_<type>_<field>

Also, to track defects through function pointers in general you'll
need to use --enable-fnptr with cov-analyze (which you mentioned
that you are using) AND to model function pointers as described
in the Checker Reference you'll also need --fnptr-models.

Let me know if this is helpful.

Best regards,
Mary, Coverity Support
```

D. Aggregation Script Commandline Usage

```
usage: interpreter.py [-h] [-o OUTPUT_FILE] [--version] [-I INCLUDE]
                      [--also-rename-structure ALSO_RENAME_STRUCTURE]
                      [--also-rename-variable ALSO_RENAME_VARIABLE]
                      [--also-rename-function ALSO_RENAME_FUNCTION]
                      [-e EXCLUDE_RENAME]
                      [--exclude-program EXCLUDE_PROGRAM]
                      -m MODULE [--prepend PREPEND]

Combining c-files

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT_FILE, --output_file OUTPUT_FILE
                        Filename of the output c-file
  --version             show program's version number and exit
  -I INCLUDE, --include INCLUDE
                        Include directories
  --also-rename-structure ALSO_RENAME_STRUCTURE
                        Also rename these structures
  --also-rename-static ALSO_RENAME_STATIC
                        Also rename these identifiers
  -e EXCLUDE_RENAME, --exclude-rename EXCLUDE_RENAME
                        Do not rename these names
  --exclude-program EXCLUDE_PROGRAM
                        Do not include these programs in the total
                        file.
  -m MODULE, --module MODULE
                        Project folders to combine, default is .
  --prepend PREPEND     Prepend this text to the total c-file.
```

Bibliography

[1] ISO. *ISO/IEC 9899:TC3 Programming languages — C*. Committee Draft. September 7, 2007. Chap. 6.4.2.1 Identifiers General. URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=50510.

[2] V. D'silva, D. Kroening, and G. Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27.7 (July 2008), pp. 1165–1178. ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923410.

[3] J.P. Queille and J. Sifakis. "Specification and verification of concurrent systems in CESAR". English. In: *International Symposium on Programming*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Vol. 137. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1982, pp. 337–351. ISBN: 978-3-540-11494-9. DOI: 10.1007/3-540-11494-7_22.

[4] Wolfgang Wögerer. "A survey of static program analysis techniques". In: *Vienna University of Technology* (2005). URL: http://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/Woegerer-progr-analysis.pdf (visited on 07/10/2014).

[5] Andreay Rybalchenko. *Model-Checking*. 2012. URL: https://www7.in.tum.de/um/courses/mc/ss2012/index.php?category=folien (visited on 06/02/2014).

[6] Dawson Engler and Madanlal Musuvathi. "Static Analysis versus Software Model Checking for Bug Finding". English. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernhard Steffen and Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 191–210. ISBN: 978-3-540-20803-7. DOI: 10.1007/978-3-540-24622-0_17.

[7] Daniel Marjamäki. *Cppcheck Design*. Jan. 21, 2014. URL: http://sourceforge.net/projects/cppcheck/files/Articles/cppcheck-design.pdf/download (visited on 06/23/2014).

[8] Dawson Engler et al. "Checking System Rules Using System-specific, Programmer-written Compiler Extensions". In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI'00. San Diego, California: USENIX Association, 2000, pp. 1–1. URL: http://dl.acm.org.eaccess.ub.tum.de/citation.cfm?id=1251229.1251230.

[9] Coverity Inc. *Sofware Testing and Static Analysis Tools | Coverity*. 2014. URL: http://www.coverity.com/ (visited on 06/02/2014).

[10] Al Bessey et al. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World". In: *Commun. ACM* 53.2 (Feb. 2010), pp. 66–75. ISSN: 0001-0782. DOI: 10.1145/1646353.1646374.

[11] Coverity Inc. *Coverity® 6.0.3 Documentation*. Chapter 4. Models and annotations. 2012.

[12] Clang. *Checker Developer Manual*. under construction. 2014. URL: http://clang-analyzer.llvm.org/checker_dev_manual.html (visited on 06/16/2014).

[13] Zhongxing Xu, Ted Kremenek, and Jian Zhang. "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. Heraklion, Crete, Greece: Springer-Verlag, 2010, pp. 535–548. ISBN: 3-642-16557-5, 978-3-642-16557-3. URL: http://dl.acm.org/citation.cfm?id=1939281.1939332.

[14] Clang. *Clang API Documentation. clang::ento::SVal Class Reference*. 2014. URL: http://clang.llvm.org/doxygen/classclang_1_1ento_1_1SVal.html (visited on 06/25/2014).

[15] Dawson Engler. "Statistical Inference of Static Analysis Rules". US 7,505,952 B1 (Menlo Park, CA (US)). 2009.

[16] Inc. Free Software Foundation. *GNU Automake*. Version 1.3. 2008. URL: https://www.gnu.org/software/automake/manual/automake.html (visited on 06/28/2014).

List of Figures

2.1. Call graph of procedures main and f 4
2.2. Model of Listing 2.1 5
2.3. Example of type inference 6

5.1. Changed event callback 19

6.1. Compilation process 24

7.1. Bug with incorrect scheduler function 31
7.2. False positive with Clang 32
7.3. Division by Zero bug 35

Listings

2.1. Sample program 4

3.1. Double free bug 9
3.2. Wrong struct size 10

4.1. Coverity `malloc()` model 12
4.2. Different struct sizes 13

5.1. False positive 20
5.2. Double free bug 20
5.3. Scheduler model for GNUnet 21
5.4. GNUnet scheduler macro function 21

6.1. Sample `Makefile.am`[16] 24
6.2. Makefile.am variables 25
6.3. Control flow in Makefile.am 26
6.4. Automake result 26
6.5. After `configure.sh` when `TEST` is true 26
6.6. Libevent macro 28
6.7. Tor header "src/or/addressmap.h" 28
6.8. GNUnet struct 29

7.1. Libevent patch in `event.c` 33
7.2. Libevent sample program 34

A.1. Aggregation script 41