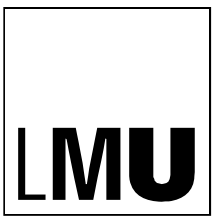
SOFTWARE ENGINEERING

Elite Graduate Program

Master's Thesis

Compositional Analysis for Exposing Vulnerabilities – A Symbolic Execution Approach

Thomas Hutzelmann



Universität
Augsburg
University

TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

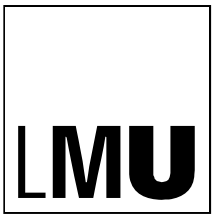
SOFTWARE ENGINEERING

Elite Graduate Program

Master's Thesis

Compositional Analysis for Exposing Vulnerabilities – A Symbolic Execution Approach

Autor:	Thomas Hutzelmann
Matrikelnummer:	1396618
Beginn der Arbeit:	21. Juni 2016
Abgabe der Arbeit:	21. Dezember 2016
Erstgutachter:	Prof. Dr. Alexander Pretschner
Zweitgutachter:	Prof. Dr. Alexander Knapp
Betreuer:	M.Sc. Saahil Ognawala



Institut für Software & Systems Engineering

Universitätsstraße 6a D-86135 Augsburg

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 21. Dezember 2016 Thomas Hutzelmann

Acknowledgments

I would like to express my great appreciation to Prof. Pretschner for the helpful feedback, the valuable suggestions, and his very precise critiques. I am particularly grateful for the assistance given by Prof. Knapp. No matter, what problems I was facing, he always had a sympathetic ear for me. This thesis would not have been possible without Saahil Ognawala. Our numerous discussions have sharpened my deep understanding of the topic and have led to some unique ideas. Finally, I want to express my gratitude for being part of the Software Engineering Elite Graduate Program. It is a great study program and offers a unique insight into a variety of topics.

Abstract

This thesis deepens the research about MACKE, a framework for compositional program analysis based on symbolic execution with the KLEE engine.

MACKE decomposes the analyzed program into small units, each containing one function from the program with all its dependencies. These units are analyzed with KLEE in order expose vulnerabilities inside the function. Then, MACKE matches all discovered problems with the same reason and uses additional KLEE runs aiming to propagate the problem to the entry point of the program. Finally, all results are refined to an error report for the complete program.

This thesis is divided into two main parts: The first part revisits the algorithms inside MACKE and discusses improvements covering several weak points in the old implementation. Among others, it refines the unit generation, simplifies the error matching and introduces a new search strategy for KLEE. The second part uses the new implementation to analyze 16 different real world projects. With the generated data, it shows the efficacy of the proposed changes and compares the performance with pure KLEE analysis. Thereby, several benefits for the uncovered vulnerabilities and the overall program coverage are illustrated.

Contents

1. Introduction

In today's software landscape, bugs inside the programs are still a serious and widening problem. In a study from 2002 [62], the National Institute of Standards and Technology estimates, that the annual costs of software failures for the US economy sum up to 60 billion dollars. With formal proofs of software correctness being limited to smaller research projects, exhaustive testing is the only way to assure a high quality of the implementations. Although the NIST assumes that an improved testing infrastructure might save one-third of these costs, writing tests is still a manual task, that is done more or less intensively in parallel with software development. But what happens, if these hand crafted test cases are not enough or if a piece of legacy software has no test cases at all?

The Heartbleed bug inside OpenSSL [22] is a good example for such a bug. OpenSSL is open source library responsible for secure and encrypted connections to millions of websites on the internet. Despite the strong security policy, at the end of 2011, the developers added a buffer overflow vulnerability to the code base, that remained undiscovered for more than two years, until March 2014. Even worse, using the correct input allows a potential attacker to extract the complete private key from the server. This bug is a severe example, but it emphasizes the requirement of an automatic way for test generation, that covers most parts of the program and especially the corner cases that are easily missed with tests written by humans. Among others, random testing [21] and fuzzing [60] are prominent representatives of automated black-box testing methods.

A promising white-box strategy aiming to close this gap in testing is symbolic execution [34]. Instead of executing one branch from the start to the end of the program, it analyses the instructions step by step and reasons about multiple paths through the program at the same time: Each instruction becomes a constraint in a logic formula, each branching decision (e.g. after a conditional statement) creates orthogonal clones that follows different paths through the program and finally a constraint solver is used to remove all unfeasible paths and to generate example input for the feasible paths. In theory, this approach could cover all possible paths through the program, but unfortunately, it suffers from two major problems.

First of all, not all legal expressions inside the program can be transformed to reasonable formal representations. The cryptographic functions inside OpenSSL are a good example for this problem. A smaller snippet of such a function is depicted in Listing 1.1. It calculates the md5 [54] fingerprint of a given string and compares it with another input given by the caller. Although it is an old and very weak hash function, if both inputs are considered as symbolic variables, this is an almost impossible challenge for the constraint solver, as hash functions are explicitly designed to be too complex for a complete mathematical analysis.

```
1 bool check_hash(char* message, char hash[16]) {
2   char check[16];
3   md5sum(message, check);
4   return memcmp(hash, check, 16*sizeof(char)) == 0;
5 }
```

Listing 1.1: Small program breaking pure symbolic execution

A simple workaround for such problems is the so-called concolic execution. It mixes concrete inputs with symbolic variables and uses a compiled version of the program, that can actually be executed, whenever it is needed. For the small sample program, it is possible to make only the `hash` symbolic and use a concrete variable for the `message`. Calculating the md5 hash of a concrete variable is simply done by normal execution and the remaining symbolic variable suffices to express all possible outcomes of the program, i.e. to find a string, that is identical to the previously calculated hash and another one that is different.

This mix of variable types allows progress through parts that cannot be analyzed with pure symbolic variables. Furthermore, it goes along without an explicit model for the underlying system, e.g. all file operations, hence all these operations can just be executed normally and the analyses can continue afterward.

Unfortunately, the second problem with symbolic execution, the state space explosion [63], has no such easy workaround. Exploring all paths through bigger programs contains a ton of branching decisions and each of them forces another clone of the program state. Just a single loop, that can be iterated infinitely, generates an endless amount of states, which can never be fully explored. Even without infinite loops, the total number of required states is exponential within the analyzed program size. With the 460k lines of code in OpenSSL, a complete representation of all occurring states is far beyond the capabilities of today's computers and even if a complete representation would be available, an SMT-solver certainly cannot show or refute their feasibility within an affordable amount of time.

The main topic of this thesis is MACKE, an algorithm that decomposes the analyzed program into smaller parts, which hugely reduces the state space explosion problem. Each of these components is analyzed with concolic execution in order to uncover potential errors in the memory handling, e.g. invalid pointer dereferences or buffer overflows. Afterward, these discovered errors are subject to distinct static and dynamic analyses in order to determine their impact on the program and to escalate the effect up to the entry point of the program. This can potentially show, that the error can be accessed by an attacker with certain capabilities, which classify the underlying error as a vulnerability with a corresponding severity score.

This thesis is structured in two main parts: The first part explains the algorithm and suggests several improvements to the original implementation. The second part evaluates the results of the algorithm on 16 real-world open-source projects and discusses the benefits for the overall runtime, the revealed vulnerabilities, and the achieved coverage. Finally, it discusses some related approaches and gives an outlook to future work.

Before starting with the discussion in the following chapters, just two minor notes: When talking about the suggestions for MACKE, there are two different artifacts both named MACKE: the original framework suggested [46] and implemented [45] by Ognawala et al. on the one side, and the implementation with all the changes suggested in this thesis on the other side. In order to have a short, unique name for both of them, the former one is called "old MACKE" and the latter one "new MACKE" correspondingly. In cases, where this prefix is missing, the surrounding statement applies for both.

Finally, please note, that this thesis sometimes uses the word "I" and that this exclusively refers to the author himself. The design of algorithms is a big, recurring topic in this thesis and – although the pros and cons of the different approaches are discussed – choosing one particular strategy is always an arguable decision someone has to make. For this thesis, these decisions are taken by the author. If, after this thesis, an alternative strategy turns out to be more suitable, I am totally the one to blame for these decisions. In order to emphasize this fact, the word "I" is used in such situations.

2. Overview of the Framework

MACKE – the abbreviation for "Modular And Compositional analysis with KLEE Engine" – is a framework for dynamic white box analysis and test case generation of arbitrary C programs. It decomposes the program under analysis into smaller components and executes them concolically with the KLEE [10] engine. The findings of each separate analysis are matched and combined statically and afterward, several additional KLEE runs are started in order to refine the error report and to escalate the errors to previously unaffected parts of the program. Finally, it combines all these loose findings into a single, more meaningful error report.

Besides the compilation to LLVM bitcode, the analysis of a project with MACKE is split into three distinct phases that are depicted in Figure 2.1 and are discussed consecutively in the following sections.

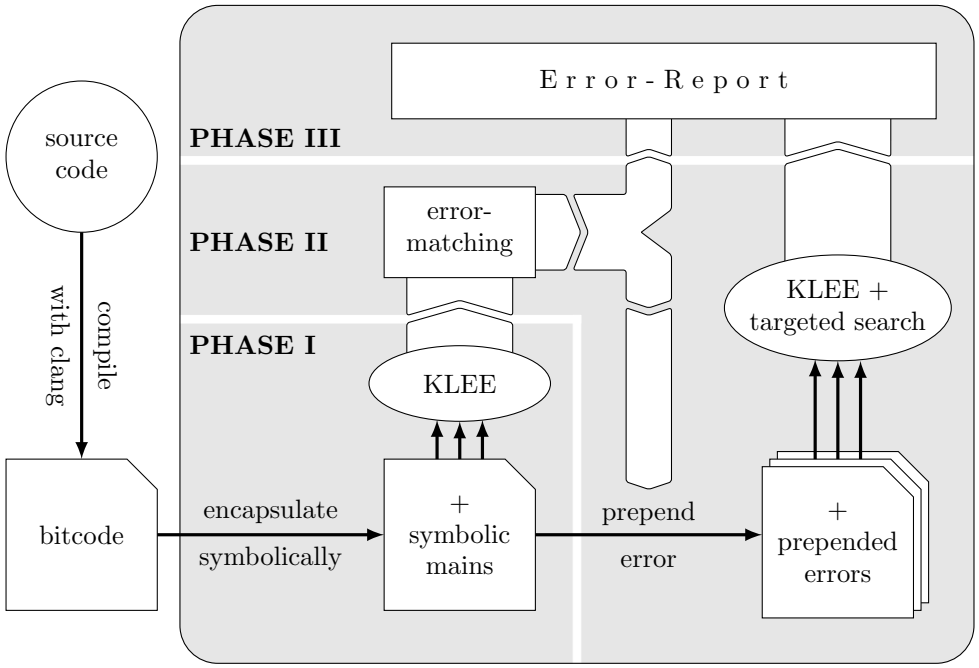


Figure 2.1.: Internal structure of MACKE

For a better understanding, the detailed explanations are accompanied by a small example program. The program takes an arbitrary integer as an input and checks, if it divisible by 6 for smaller numbers or divisible by 30 for bigger numbers. The core function of this program is listed in Listing 2.1, the remaining non-listed functions are basically one line checks, that redirect the divisibility checks to other functions. A complete call graph of the example program is depicted in Figure 2.2. I am aware of this program not being very practical and extensively designed, but its key value is a handy call graph, that is better understandable than naming all nodes generically.

```
1 int example(int n) {
2     return (n < 1000) ? divisible_by_6(n)
3                       : divisible_by_30(n);
4 }
```

Listing 2.1: Excerpt from a small example program

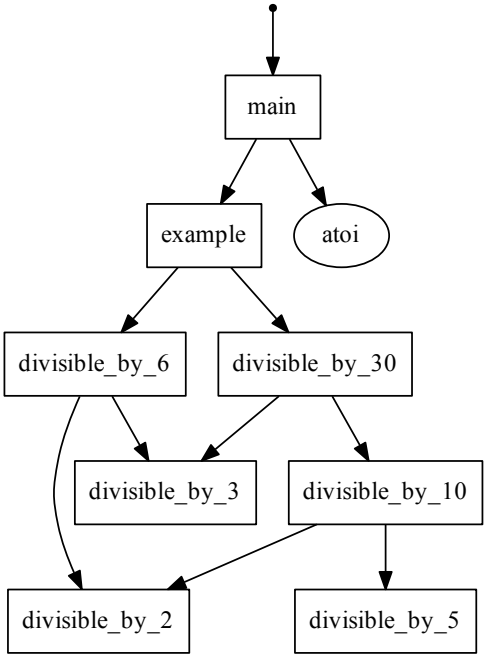


Figure 2.2.: Full call graph of the example program

Before starting with the explanations, please assume, that the program under analysis has already been compiled successfully into its bitcode representation. This process is very technical and discussed later. Thereby, this walk-through can directly start with the more important parts in the beginning of phase one.

2.1. Phase I: Isolated analysis

Straightforward symbolic or concolic execution starts the analysis of a program with its main function and tries to unfold the paths through the program into deeper parts of the program. Unfortunately, the state explosion problem causes this approach to fail early without reaching most of the deeper parts in the program.

Hence, MACKE decomposes the program into smaller components and analyses each of them separately with KLEE. A smaller program contains fewer instructions resulting in a majorly reduced number of program states. This makes the components easier analyzable and has the potential for a much higher coverage.

In theory, each arbitrary chunk of code can be analyzed with this approach, but MACKE only builds one component per function. A function defines a clear interface for its input variables and return value. Furthermore, the call graph forms a suitable map to reveal the structure of the program and correlations between the functions. In addition, most programmers design functions to have a high cohesion inside a function and low coupling with other functions. All these make functions the perfect candidate for smaller subcomponents of the program. The only task remaining for MACKE is to add new main functions into the program, that mark all function inputs as symbolic.

The example program consists of nine different functions. One of them, `atio`, is an external function, which does not form a separate component. Seven out of the remaining eight components are depicted in Figure 2.3. Obviously, the components for the deeper parts of the program, especially for all leaf nodes in the call graph, are relatively small, whereas higher level functions require the code of all called functions, which makes the corresponding component relatively big. In extreme, this is the case for the eighth component – the main function – where the "component" represents the whole program.

Assuming, that there is an erroneous instruction in a leaf node of the call graph, it is quite likely, that MACKE discovers it in phase one. However, each level higher in the call graph decreases the chance for the discovery, although the error might still affect the function. Hence, MACKE uses the second phase, to handle this problem specifically.

2.2. Phase II: Building error chains

All the errors discovered in the separate functions of the program during the analysis in phase one can be split into two groups: Errors inside functions, that have been shown to be also reachable from a caller to this function, i.e. two errors reported in two adjacent functions caused by the same instruction, and errors, where a caller has not been able to trigger the error (yet). Figuratively, MACKE's error matching tries to build chains through caller-callee relations, starting in the function containing the erroneous instruction up to the point, where all callers are no longer affected by the error – ideally up to the main function. The second type of errors forms the ends of these chains.

All ends of all error chains are a topic for further investigation in phase two. In front of the last affected function, MACKE prepends an error summary, that can report the error without analyzing the complete function again. Then it starts additional KLEE runs from all callers, that uses a special targeted search strategy, aiming directly for the shortest path to the error summary. This directed execution reaches parts that were not covered in phase one and hopefully can trigger the summary of the error. If thereby a new function can be shown to be affected, the analysis continues with all callers of this function and so on. All the information about the error chains is used for the error report in phase three.

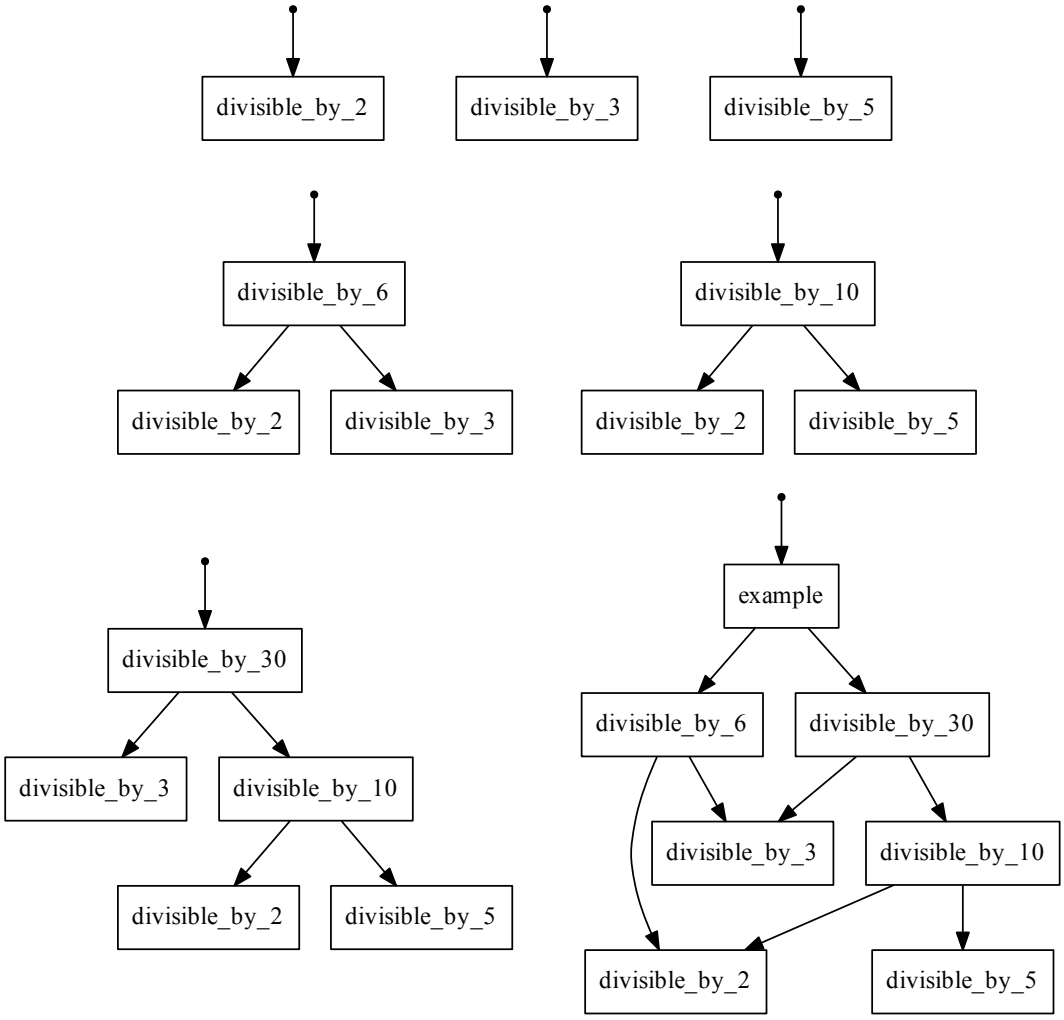


Figure 2.3.: Isolated components of the example program in phase I

For the small example program, assume, that there is only one erroneous instruction residing in `divisible_by_5`. Additionally assume, that after phase one, KLEE has discovered two errors: The one in `divisible_by_5` and another in one of its callers, `divisible_by_10`. With error matching, MACKE can reason, that both errors are caused by the same erroneous instruction and builds a chain with both functions as chain segments. Using this information, no additional search is required starting in `divisible_by_10`, but with `divisible_by_30` not being affected, MACKE generates an error summary for the lower function and starts an additional KLEE run from `divisible_by_30`. If this targeted concolic execution is successful, it causes another run from `example`, and if being successful again, another run from main.

The steps of these error propagation are depicted in Figure 2.4. Especially, the call graph starting with the `example` function is interesting, because, although it uses the error summary, the remaining call graph is still quite big. This emphasizes the requirement of a targeted search strategy. With the information, where the error might occur from phase one, only one branch of the if-statement (see Listing 2.1) is relevant. Hence, the call graph might be bigger, but the other branch incorporating most of the call graph is never executed during the second analysis.

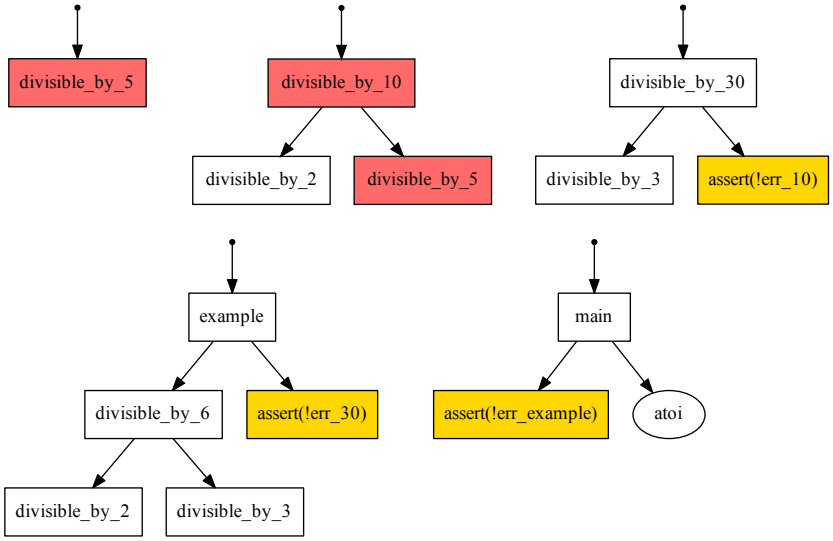


Figure 2.4.: Possible error propagation in phase II

2.3. Phase III: Generating a final error report

The previous two phases normally start hundreds of distinct KLEE runs during the analysis of an average program, that report dozens of vulnerable instructions. The sheer amount of data tends to be overwhelming for the user, so MACKE takes a final phase to compose an interactive error report summarizing all separate analyses.

In addition to a navigation within the call graph of the program, MACKE offers a ranking mechanism, that calculates scores for each vulnerability. Thereby the user gets an ordered priority list, denoting the vulnerabilities that should be inspected and probably fixed first. The score considers among others, the length of the caller-callee chains that are affected by the error, the remaining distance to the main function and the frequency of multiple errors inside the same function. The weighting of the different factors can also be adjusted by the user.

Although this third phase is not covered explicitly in this thesis, it is part of the MACKE framework and remains usable in the new implementation.

3. Methodology and Implementation

After having given an overview of the internal structure of MACKE in the previous chapter, this chapter revisits most of the processing steps and suggests improvements for them. Here, each section shares almost the same structure: In the beginning, it briefly describes the problem and how it is solved in old MACKE. Then, it lists disadvantages of this approach and finally, it discusses suitable improvements that are implemented in new MACKE. At the end, this chapter gives a step-by-step example of an analysis and lists all the intermediate program modifications that were applied by new MACKE.

3.1. Generating bitcodes from existing projects

MACKE's as well as KLEE's analysis work on LLVM bitcode level. Therefore, any project that should be analyzed, must provide their executables in bitcode form. The Makefiles of some modern LLVM based projects are quite likely supporting such a build, but the majority of old projects does not provide any way to accomplish this modern build type. Nevertheless, especially these old projects are the most interesting ones for the evaluation, so it is very important to find a suitable way to obtain bitcode equivalents of these code bases.

3.1.1. Requirements

Before looking at the existing approaches and making a decision which one to choose, this subsection lists all the requirements for a good bitcode builder.

(1) Minimal human interaction Quick modifications to unknown source code run the risk of unintentional side effects and may introduce new errors. Hence, they should be avoided by design. Furthermore, complicated manual setups increase the amount of required work and thereby will impede other developers from including MACKE into their daily toolchain. Due to this, the amount of human interaction should be kept as low as reasonably possible. **Priority: High**

(2) Ability to modify the used compiler flags Normally, project builders for production environments use command line flags (e.g. `-O3`) to optimize the binaries and reduce their size. Although KLEE can analyze most optimized bitcode files, from my experience it sometimes crashes the analysis. Hence, the code should be built without any optimizations. Additionally, for more human readable results, the bitcode file should be compiled with debug information (i.e., using the `-g` flag). **Priority: Medium**

(3) Support of real-world projects with large code base The potential field of application for MACKE increases with the number of projects that can be built as bitcode. Therefore, the builder should be capable of handling all kinds and sizes of real-world code bases and their internal building mechanisms. **Priority: Medium**

(4) Generate executables as well as libraries MACKE can not only analyze executables with a main function as an entry point but additionally, it can analyze libraries that cannot be run standalone. Therefore, it would be nice, if the bitcode builder also emits bitcode equivalents to static and shared libraries. **Priority: Low**

3.1.2. Existing approaches

The problem of building bitcode out of existing projects affects all KLEE related research (see Section 6.2 and Section 6.6). Nevertheless, I am not aware of any publication covering this topic explicitly. During my research, I have found the approaches listed in this subsection.

klee-clang [44] This is the official approach, suggested by the KLEE project. It is a small script that forms a wrapper around the clang compiler. When a code base is built with make, this wrapper is used as a replacement for the regular compiler (i.e. `make CC=/.../klee-clang`). Internally, it modifies the flags of the command [Req(2)] and starts clang to compile or link the demanded file. This does not require any further interaction with the user [Req(1)]. Sadly, the script does not generate any bitcode for libraries [Req(4)]. Additionally, it does not support commands that require compilation and linking at the same time (e.g. `clang f1.c f2.c -o prog`), which is a big limit for the supported source code bases [Req(3)].

Manually modify the Makefile This method was used by the authors of old MACKE. It should support all types of projects [Req(3)], can consider each file separately [Req(2)] and thereby handles libraries properly [Req(4)]. On the downside, it requires the biggest possible human workload [Req(1)], which makes this method unaffordable for bigger projects or code bases that require a more complex build process. Unfortunately, I fear, that this strategy is used for most of the research about KLEE.

Gold link with LLVM gold plugin [49] This relatively new approach provides a plugin for the clang compiler and linker. With this plugin both programs also emit bitcode equivalents for all the binaries, they build. Once the corresponding shell variables are set, this does not require any extra human interaction [Req(1)]. On the downside, there is no way to remove flags from the commands [Req(2)] and static libraries (i.e. `.a`-files), which are not composed with clang, are not supported [Req(4)]. In practice not all Makefiles reference the standard shell variables (e.g. `CC`), but instead directly name the instance, they want to use (e.g. `GCC`). Then, this approach fails, inducing a few incompatible projects [Req(3)].

With none of the above approaches satisfying all requirements, I decided to develop a different approach for building bitcode out of existing projects.

3.1.3. A new approach – make+llvm

The previous approaches mostly consist of wrappers around one or two selected types of build commands and keep the rest of the commands untouched, although some of the ignored commands could be used to meet the missed requirements (e.g. wrapping `ar` commands allows to generate bitcode equivalents of static libraries).

For this reason, I decide to build make+llvm, a wrapper around the Makefile itself, handling all the relevant commands separately. The structure of the complete wrapper is depicted in Figure 3.1 and described below in detail.

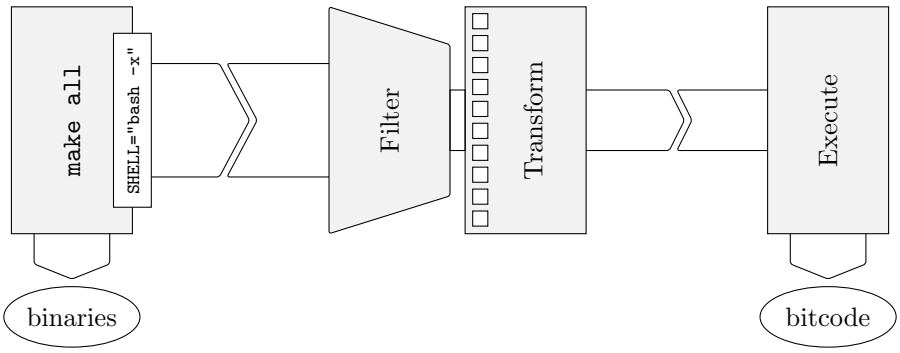


Figure 3.1.: Internal structure of make+llvm

Run normal `make` and record all commands

As the first step, make+llvm runs make as usual but adds `SHELL="bash -x"` as a flag to it. Thereby, the shell used by make emits all commands to standard output before executing them. This output is recorded by make+llvm. Please note, that inside these commands, all variables – locals as well as globals – are substituted with their actual content. Furthermore, concatenated commands (e.g. `cmd1 && cmd2` or all sort of loops) are untangled automatically by the shell. In other words, this record is a serialized list of atomic commands, that can be executed standalone without any further information. Executing the complete list again exactly replays the entire build.

Filter and transform the recording

For the sake of just building bitcode, the recorded list contains a lot of superfluous commands, e.g. all the tests from if-clauses or the preparation of local language files. Hence, the next step of make+llvm simply removes all commands, that do not have any direct bitcode reference. Then, all the remaining commands are adapted to generate bitcode equivalents. For example make+llvm strips most flags off compiler commands and adds the flags for debugging information and zero optimization; the commands for packaging static libraries are rewritten to use the llvm-linker instead and to generate bitcode with the same content linked inside the library. Make+llvm includes a transformation logic that is capable of handling the most common commands. Additionally, the different transformers are implemented in a very modular manner and thereby can easily be supplemented or assimilated to unknown build commands.

Execute the transformed commands

As the last step, make+llvm just executes all previously transformed commands. Assuming, that all transformation are correct and complete, this generates equivalents for all binaries, that were originally generated by the Makefile.

Verification of the requirements

This new strategy matches the requirements listed in Section 3.1.1 better than all previously discussed existing approaches. Once installed, make+llvm just replaces the normal make command, i.e. `./configure, make` becomes `./configure CC=clang, make+llvm` – the rest happens automatically [Req(1)]. The transformer for the compile commands can add and remove all demanded flags [Req(2)] and it has its own transformation logic for generating static libraries [Req(4)], that are missed by all previous approaches. All the projects that are referenced in the evaluation part of this thesis are built successfully with make+llvm. They are chosen from a variety of different authors and cover several build paradigms [Req(3)].

3.2. Adapt the architecture for LLVM bitcode operations

The biggest change this thesis suggests for the implementation of MACKE, regards the interaction with and generation of the bitcode files required for the analyses with KLEE. As discussed in the previous chapter, MACKE relies heavily on modified bitcode files of the analyzed program: All functions are encapsulated symbolically and potentially each uncovered error may be prepended to the function containing it. For bigger programs, this sums up to thousands of slightly different bitcode files. This section describes the way old MACKE generates these modifications, discusses the requirements on a better architecture and finally explains the design in new MACKE.

3.2.1. The design of old MACKE

The details, how old MACKE handle these bitcode modifications, is missing inside Figure 2.1. Inside the figure, the arrows suggest, that the different bitcode files at the bottom line are directly generated from a previously generated bitcode file, but old MACKE has no mechanism to directly modify bitcode. In order to generate a new bitcode file, MACKE considers the original source code of the program, applies the changes inside the C code and recompiles the project again to get the required bitcode file. This detail is left out in the figure for more clarity and for consistency with new MACKE, that does not take these detours.

This design causes a lot of problems in old MACKE: First of all, implementing the modifications in a manner that fits all types of programs, is quite complex. The syntax of C is very flexible and there are multiple ways to declare a function (e.g. the split between header and source files or the usage of function-like DEFINE macros). Old MACKE uses libClang [5] to interact with the source code, but there is no way to assert that it correctly covers all valid C constructs. More importantly, this approach hugely increases the runtime. Although advanced Makefiles are designed to cache several intermediate compilation steps, recompilation remains a quite costly task.

In order to circumvent these problems, I build all interactions with the program directly with LLVM bitcode operations. Thereby, the source code is no longer required, after the program has been built with make+llvm once. Furthermore, the bitcode syntax is relatively simple. An operation can handle all possible programs if it can handle all explicitly named subtypes of the occurring operands, which can be asserted quite easily. Finally, bitcode modifications are much faster than parsing the C code, applying the modification and recompiling the project again.

3.2.2. Requirements for the new architecture

New MACKE greatly benefits from the switch to LLVM bitcode, but there are more requirements, that need to be considered for the new architecture.

(1) Use the same LLVM version as KLEE MACKE uses KLEE to analyze each unit. So the generated units must be compatible with the LLVM version used by KLEE or all these analyses will fail because of syntactic incompatibility. At the time of writing this thesis, KLEE uses LLVM 3.4.1, that was released in May 2014, although LLVM 3.9.0 was released in September 2016. In other words, the new architecture must support a very outdated version of LLVM. **Priority: Critical**

(2) Support major LLVM version updates With the outdated version of LLVM inside KLEE, it is quite likely that KLEE skips several versions and will update LLVM to a current release soon. Certainly, this update will require some changes in MACKE's codebase, but it should definitely not cause a complete rewrite. Hence, all LLVM dependent code should be clearly separated. **Priority: High**

(3) Implement main parts of MACKE in Python Old MACKE was completely written in Python because this language is widely known by many programmers. Additionally, it allows a pretty simple and fast implementation of many program patterns, e.g. the implementation of a parallelized worker pool (see Section 3.8) is already present as a built-in Python library. Therefore, the main parts of MACKE should remain using Python. At least, the main Python code should be able to directly interact with all other parts. **Priority: Medium**

(4) Access to the full LLVM API The bitcode changes required by MACKE are quite intense: Locate specific instructions, build new functions, extract and modify the call graph – to just name a few of them. Hence, access to a big part of the API is helpful, when implementing them. **Priority: Medium**

3.2.3. Possible approaches

There are multiple ways for interacting with LLVM bitcode. The most promising ones are discussed below in more detail.

Use a Python plugin

For the purpose of implementing everything directly inside Python [Req(3)], using a Python plugin is the most natural approach. According to the Python Package Index, two packages are available.

llvmpy [40] provides quite an elaborate interface to the native LLVM API [Req(4)]. Sadly, at the time of this thesis, it only supports LLVM 3.2, which does not fit KLEE [Req(1)]. Furthermore, the authors declare this project as deprecated in favor of llvmlite. That disqualifies this approach.

llvmlite [16] is a lightweight Python library for building a JIT-compiler based on LLVM. Although being the official successor of llvmpy, it does not directly interact with the LLVM API but rebuilds equivalent functionality inside Python. This restricts the provided API [Req(4)]. Additionally, the earliest supported LLVM version is 3.5, which is too recent for KLEE [Req(1)]. Furthermore, llvmlite binds its versions to the respectively current LLVM version. Hence, using its current version forces the usage of the current version of LLVM, which collides with KLEE's update progress [Req(2)].

Directly calling the LLVM C++ API from Python

A direct call to a C or C++ function with Python code is a quite common idiom and there exist several setups for it [51, 50, 4, 1]. Thereby, it is possible to directly use the complete LLVM C++ API [Req(4)] inside Python [Req(3)]. The called C++ code is completely controlled by the user, so every required LLVM version can be used [Req(1)] and updated when necessary [Req(2)]. Although this approach satisfies all requirements, it also requires a bunch of boilerplate code and ends up in a more complex build process.

Write LLVM opt passes

LLVM provides an interface [38] to write new passes for its optimizer, that can perform all types of operations on LLVM bitcode [Req(4)]. This interface is part of the core LLVM project since version 1.0 [Req(1)] and will certainly be supported in future versions [Req(2)]. Once implemented, the new pass becomes an external program, that can be run standalone. Python supports calls to external programs, but a direct interaction with the LLVM API is not possible with this approach [Req(3)].

3.2.4. Final architecture for new MACKE

The last two approaches satisfy all requirements but draw different lines between Python and C++ code. For me, writing LLVM passes separates more clearly between all low-level bitcode operations in pure C++ and all coordination logic in pure Python code. Therefore, I decided to choose the third approach and to implement the following three distinct LLVM opt passes and use them through a tidily defined interface:

Extract call graph uses already available functions in LLVM to build the call graph of the program and to detect cycles and strongly connected components. This information is then transformed to JSON code, that can easily be parsed inside Python. This is used as a foundation for MACKE's task scheduling and later error composition. For more details, see Section 3.4 and Section 3.8.

Encapsulate symbolically takes an existing function and adds a function to the bitcode that introduces several symbolic variables as arguments for a final call to the encapsulated function. The new function has no arguments, so it can be used as an entry point to the program, i.e. as the main function. This is used in the beginning of phase one in MACKE. For more details, see Section 3.3.

Prepend errors takes a previously generated KLEE error report from phase one and matchs the referenced variables with the variables used by the corresponding function inside the bitcode. Then, it introduces a new function with the same signature that checks, if a possible assignment of variables can trigger this error and, if not, calls the function normally. All calls to the original function inside the program are replaced with calls to the new function. This results in a control flow graph, where the error checks are prepended to the original function. For more details, see Section 3.5

Please note, that all three passes are implemented in a separate code base and, although being heavily used by MACKE, they can also be executed without any interaction with it and not only in a specific situation during the analysis. Thereby, they can be tested and updated separately without even touching any other code inside the rest of MACKE and hopefully can be reused in future projects.

After this overview of the three fundamental bitcode operations, the following sections give more details about their implementation and discuss several considerations of challenges discovered during the evaluation.

3.3. Symbolic encapsulation

The introduction of new entry points for symbolic analysis of separated functions is not discussed explicitly in the paper describing old MACKE's implementation. Nevertheless, this is quite a complex task due to the variety of different function signatures, that can be used inside C programs. Especially pointer variables require a special construct that is discussed in this section. Please note, that the following examples only show the encapsulation of one variable. If a function uses more variables, the presented strategies are simply replicated for all parameters.

3.3.1. Non-pointer variables

Variables, that are no pointers (i.e. directly referenced variables) are the easiest case for symbolic encapsulation. An example of such a function is depicted in Listing 3.1. In order to make the argument symbolic, MACKE creates a new variable of exactly the same type and marks the memory used by it as symbolic inside KLEE.

```
1 void foo(int i);
2
3 void macke_foo_main() {
4   int i;
5   klee_make_symbolic(&i, sizeof(i), "i");
6   foo(i);
7 }
```

Listing 3.1: Symbolic encapsulation of a non-pointer variable

3.3.2. Single pointer variables

For pointer type variables, the encapsulation becomes a little trickier. A very basic approach is depicted in Listing 3.2. It is very similar to the non-pointer variables, but additionally to the variable declaration, new memory is allocated for the pointer and, instead of the pointer, the memory referenced by this pointer is marked as symbolic. Thereby, the generated constraints and test cases correctly describe the referenced memory. This strategy is used in the implementation of old MACKE.

```
1 void foo(int* i);
2
3 void macke_foo_main() {
4   int* i = malloc(sizeof(int));
5   klee_make_symbolic(i, sizeof(*i), "i");
6   foo(i);
7 }
```

Listing 3.2: Problematic symbolic encapsulation of a pointer variable

This strategy has a major problem: All pointer variables refer to exactly one element. If this pointer is used as an array inside the program, it has only the size one. This prevents KLEE from reaching any part of the program, that needs arrays with two or more elements. For circumvention, this basic strategy needs an improvement.

Ideally, KLEE should support symbolic arrays, where it does not only store constraints for the actual content, but also for the size of the array. Thereby, this situation could be solved perfectly, but unfortunately, KLEE has no symbolic arrays. So new MACKE relies on a small hack for these variables.

The strategy in new MACKE is depicted in Listing 3.3. Since it is impossible to use all possible array sizes, new MACKE chooses several arbitrary sizes for the array and analyses the function with each of them in different KLEE forks. The forking inside KLEE is triggered by introducing a new symbolic integer (line 4) and a switch-case statement based on this variable (line 6). When KLEE reaches the switch, it forks into each case separately. Inside each case, a different array size is chosen (line 8 – 12). This size is then used for a similar pattern as shown in Listing 3.2 (line 14 – 16).

```
1  void foo(int* i);
2
3  void macke_foo_main() {
4    int fork_i = klee_int("fork_i");
5    int symsize_i;
6    switch(fork_i) {
7      default:
8      case   1: symsize_i =   1 * sizeof(int); break;
9      case   2: symsize_i =   2 * sizeof(int); break;
10     case   4: symsize_i =   4 * sizeof(int); break;
11     case  16: symsize_i =  16 * sizeof(int); break;
12     case 128: symsize_i = 128 * sizeof(int); break;
13   }
14   int* i = malloc(symsize_i);
15   klee_make_symbolic(i, symsize_i, "i");
16   foo(i);
17 }
```

Listing 3.3: Better symbolic encapsulation of a pointer variable

As a consequence of this construction, KLEE must push several forks on the same paths through the analyzed function, which multiplies the required runtime, although all internal caches can be reused on all forks. After some trails with several sizes, I have decided to use 1, 2, 4, 16, and 128 as sizes for the symbolic arrays, because they seem to be a good trade-off between being complex enough for all program paths and the required runtime. Please note, that choosing only one huge size is not sufficient because it is no superset of all smaller sizes. Buffer overflows only occur on accesses beyond the end of the array, which becomes the more unlikely the bigger the array is.

3.3.3. Nested pointer variables

In order to support the full range of C function variables, MACKE must be able to handle nested pointers (e.g. pointers to pointers to variables). Again, there is no symbolic type for these variables and, sadly, the trick for single pointer variables does not scale for these variables. Theoretically, the fork strategy can be nested to support arbitrary levels of indirection, but the runtime for the analysis with KLEE explodes. Hence, new MACKE excludes functions with a multi-pointer argument from the analysis and handles them, like if no error occurs during the analysis. Adding symbolic arrays with a varying size and support for nested pointers is beyond the scope of this thesis.

Finally, a small note on old MACKE: Its implementation does not exclude multi-pointer functions explicitly. Instead, it only makes the first pointer layer symbolic and its content is used for the addresses, where the second pointers point to. So all its content just refers to arbitrary memory addresses, but neither this memory is make symbolic nor is the content of these pointed areas stored in the report. Hence, in spite of being executable, the generated KLEE report is utter nonsense without the actual memory content.

3.4. Matching errors

With using multiple KLEE runs on separate components of the analyzed program, MACKE faces a lot of non-correlated error reports. However several of them might be caused by the same vulnerable instruction and form error chains through the call graph of the program. In order to identify these chains, MACKE needs a matching mechanism on KLEE's error reports. This is explicitly needed after phase one, but also the results from the targeted search in phase two require some considerations. Old MACKE suggests approaches for the matching in both situations, but new MACKE uses a simpler and more reliable implementation that is presented in this section.

3.4.1. Match errors from phase one

After phase one is completed, MACKE starts the static compositional analysis in order to rule out all errors that have already been shown to be triggerable by the corresponding parent function. Therefore, old MACKE looks at the reported stack trace given by each KLEE error report and searches for endings with identical function names [46, section 2.2.2, figure 1].

Unfortunately, this strategy misses some matching errors, when the compiler is allowed to apply optimizations to the analyzed programs, as is the case with KLEE's `-optimize` flag used inside MACKE. For example, consider a parent function of a leaf node in the call graph. When this function is analyzed separately, dead code elimination removes most of the program – basically everything except the two functions – and inside the remaining code, the leaf function is probably only called once, so it is quite likely, that this call is inlined by the compiler. When the analysis of the optimized program triggers an error inside this inlined function, i.e., a matching error is hit, the stack does not exhibit the inlined call and old MACKE cannot match these errors. This schema has the biggest impact on parents of all leafs, nevertheless, it can occur on all newly generated units.

New MACKE, therefore, does not use the stack for error matching, but instead directly looks at the reported filename and line number, that causes the errors. Candidates for matching errors refer to the same location in the source code. Then new MACKE uses the call graph of the unoptimized program to reconstruct the path and identify matching errors.

This strategy was not applicable inside old MACKE because it modifies the source code (e.g. it adds some additional lines) and recompiles it again for each analysis. So the same instruction from the same file might get another filename or line number. New MACKE directly modifies the bitcode and therefore does not interfere with filenames and line numbers.

3.4.2. Match errors from phase two

The errors found in phase two require a different type of matching. During the targeted search for an error summary, KLEE might find some other errors in the program, that are not correlated to the prepended error. These errors are no continuation of the originally investigated error chain. Hence, MACKE needs a way to identify errors triggered by the summary and handle the other ones differently.

Furthermore, it is possible, that multiple different error chains have identical caller-callee chain segments. As a basic strategy, MACKE could use multiple targeted search runs starting from the same entry point, aiming for the same function calls, but checking different error summaries. Obviously, using multiple runs for the same caller-callee pair wastes a lot of execution time, because KLEE must find the identical way to the calls again and again in all executions only to check a different error summary. With a procedure to distinguish between different error sources, all relevant summaries can be put inside the program location at once. No matter, if KLEE can trigger one or all errors, the execution and path discovery in a single run continues until KLEE cannot find any new paths or the run time is exceeded, but each path is only explored once.

Old MACKE uses a slight modification of the basic strategy described above. First, it removes all previous assert statements inside the analyzed program. Then it adds a single assert statement checking the error summary from the end of a single error chain and starts a targeted search for this assert statement. If KLEE reports an assert error during the analyses, it can only be caused by a satisfiable error summary, and, because old MACKE only analyzes a single error at a time, that error must be a continuation of the analyzed error chain. All in all, a very simple matching strategy, but it requires hard restrictions on the execution.

On that account, new MACKE suggests an approach enabling a fast and unambiguous error matching in phase two, by using one of KLEE's internal functions: `klee_report_error()`. If an error summary can be satisfied, it is used to report a new distinct type of error called MACKE error. Thereby, all confusion with other errors, that might have been inside the program previously is avoided. Additionally, a unique identifier for the corresponding error chain is added to the generated error report. Hence, multiple error summaries can be used simultaneously and be correctly appropriated afterward. This strategy modifies the program very deliberately, does not require any restrictions on the execution schedule and still enables a trustworthy error matching.

3.5. Prepend error summaries to functions

When the summary of an error should be prepended to a function, the matching with the original error, as it is discussed in Section 3.4.2, is not the only challenge that must be considered in the implementation. An excerpt from a problematic source code is depicted in Listing 3.4 and used as an example in the next subsections.

In the following, this section names several challenges, discusses their implications and describes how new MACKE handles them.

```
1 int target(int i) {
2   /* Contains an error for some i */
3 }
4
5 int source(int n) {
6   int t = (n < 42) ? target(n) : 0;
7   return (t > 0) ? target(t) : 0;
8 }
```

Listing 3.4: Example code including several difficulties

3.5.1. Challenge: Multiple calls to the same function

The example code contains two functions: one called `target`, that contains an error that KLEE can trigger with some, but not all assignments to `i`; the other function called `source` calls `target` twice in different locations with different variables. For the sake of the argument, please assume, that KLEE cannot find any error in `source`.

In cases like this, it might be possible, that the error in the called function cannot be triggered by all calls in all parent functions. With the example, this can happen, when the error in `target` is only triggered by very high values of `i`, that cannot be passed in the first call. For the purpose of uncovering such errors, MACKE must go past the first call and look at the second one without any influences to the state space during the analysis and the error check.

Programs containing such constructs cannot be analyzed with old MACKE because it replaces the calls to the target function with an assert statement, that checks if the error occurs [46, Listing 2]. So old MACKE replaces both calls with an assert statement. But this is problematic in cases like the example code, where the second call uses the result of the first call. For being still compilable and sticking with the same execution semantics, at least the original function calls must be left in place in addition to adding the assert statements checking the error summary in the correct position.

In order to analyze such programs correctly, new MACKE modifies the source code in a more gentle way: It introduces a new function with the same signature that just checks for the error with the summary and, if it is not satisfiable, calls the original function normally. All calls to the old function are replaced with calls to this new function before the analysis is started. In short, MACKE has prepended the error to the function.

With this structure, the program is executed normally, as long as the error summary is not satisfiable. In other words, new MACKE passes the first call in the example normally and can look for the second call. Please note that the modification is done on LLVM bitcode level, but if it would be done on C code level, the resulting program would look similar to Listing 3.5.

```
1  int macke_error_target(int i) {
2    if (/* Error summary can be satisfied */) {
3      klee_report_error(/* ID for the error */);
4    } else {
5      return target(i);
6    }
7  }
8
9  int target(int i) {
10   /* Contains an error for some i */
11 }
12
13 int source(int n) {
14   int t = (n < 42) ? macke_error_target(n) : 0;
15   return (t > 0) ? macke_error_target(t) : 0;
16 }
```

Listing 3.5: Example code with prepended error

3.5.2. Challenge: No interference through the error checks

Sadly, this way to prepend the errors silently introduces another problem: Originally, the call to `target` is only constrained by the analyzed program, but the prepended error check adds the constraint, that the error summary must be unsatisfied with the used variables in the following program.

First of all, this increases the runtime required for a complete analysis, because all following queries to the SAT-solver issued by KLEE now contain all variables mentioned by the error summary. Thereby, they must be considered while finding possible solutions, although they do not add any valuable information to the execution state.

This interference causes even worse problems when more than one error is prepended at the same time. This happens each time, when a function has a pointer argument (see Section 3.3) or contains more than one error. Then, the order of the error checks matters, because there is no guarantee that all occurring errors are independent. Imagine the worst case, that the second error that is checked, is an implication of the error checked first. In this case, the second error can never be triggered.

For old MACKE, this is not a big problem, because it only investigates one prepended error for exactly one call at a time, but new MACKE needs a mechanism to prevent this influence. Therefore, MACKE adds code similar to Listing 3.3, that generates multiple KLEE forks of the current execution state. Then, one fork is used for each error summary and, no matter whether the check was successful or not, the execution stops afterward.

Finally, one last fork is used to continue with the normal execution of the program. Each fork has its own state space and – except caching – does not interfere with any other fork, so the normal execution continues without any alteration. The resulting code for the example with two error summaries is depicted in Listing 3.6.

```
1  int macke_error_target(int i) {
2    int n = klee_int("symfork");
3    switch(n) {
4      case 1:
5        if (/* Error summary 1 can be satisfied */) {
6          klee_report_error(/* ID for error 1 */);
7        } else {
8          klee_silent_exit();
9        }
10       break;
11     case 2:
12       /* .. */
13     default:
14       return target(i);
15       break;
16   }
17 }
```

Listing 3.6: Example code with the forking mechanism

3.5.3. Challenge: Correct sizes for `memcmp`

Additionally to all previously explained challenges, the biggest problem in old MACKE during phase two lies inside the error summary itself. Old MACKE uses `memcmp` to check if the memory content of the erroneous inputs can be built with the current execution state. This `memcmp` requires the size of the memory, that should be compared as an explicit parameter. In order to determine this size, old MACKE simply uses the C operator `sizeof`, but this does not work for all types of variables, especially for array types. According to the C standard [32, section 6.5.3.4], "[t]he sizeof operator shall not be applied to an expression that has [...] an incomplete type". Roughly speaking, for all arrays, whose sizes are not explicitly known at compile time, old MACKE relies on undefined behavior. For example, the small program in Listing 3.7 seems to return the size of the array containing all program arguments – in other words the same number as stored in `argc` – but depending on the compiler, this program just returns a constant value (e.g. 8) instead.

Luckily, KLEE records all memory sizes internally and offers a function, that covers exactly this problem: `klee_get_obj_size()`. This function uses all information that is available at runtime to determine the size of the element referenced by an arbitrary pointer. Therefore, new MACKE uses a double-tracked strategy: For all non-pointer values, it just compares the content of the value with the normal == operator, and for pointer variables, it checks, if the value has the correct object size and then it compares the actual content.

```
1 int main(int argc, char** argv) {
2   return sizeof(argv);
3 }
```

Listing 3.7: A problematic usage of `sizeof`

3.6. Step by step modification of an example program

With all the previously discussed improvements, the modifications of the source code look quite different compared to old MACKE, despite the fact, that they are fulfilling almost the same purpose. In order to emphasize the differences, this sections revisits the code example from the old MACKE paper [46, Listing 1 and 2] and shows, what modification new MACKE applies to the same analyzed source code.

Please note, that the following code excerpts are written in C code, but new MACKE works on pure LLVM bitcode. So this code is only a reconstruction of the analysis for the purpose of easier visualization, nevertheless, the result of a compilation of this code *should be very close* to the actual bitcode used by new MACKE.

```
1  int mask_b(int* b, int n) {
2    b[n++] = 1; /* potential buffer overflow */
3    return n;
4  }
5  int main(int argc, char** argv) {
6    int i, n=0, b[4] = {0,0,0,0};
7    for (i = 0; i < argc; i++) {
8      if (*argv[i] == 'b') {
9        n = mask_b(b, n);
10     }
11   }
12 }
```

Listing 3.8: Example program before the analysis

Listing 3.8 shows almost the same code presented in the old MACKE paper. Only a few lines of code that do not contain any relevant instructions are omitted, in order to reduce the overall code size and thereby increase readability. The program processes all its arguments and generates a bitmask for inputs starting with `'b'`. Notably, a buffer overflow occurs, if more than four arguments of the program starts with `'b'`

In phase one, MACKE looks inside the code and encapsulates all suitable functions symbolically. In the example program, only `mask_b` can be encapsulated. So MACKE adds a new function `macke_mask_b_main` to the program and leaves the rest unchanged as depicted in Listing 3.9. This new function composes three steps: The first step (line 2 – 4) makes `n`, a simple integer variable, symbolic. The second step (line 6 – 19) makes `b`, a pointer to an integer, symbolic. As depicted, the effort for pointer variables is much higher than for non-pointer variables. Finally, these new symbolic variables are used in a third step to call the encapsulated function.

```
1  void macke_mask_b_main() {
2    // Introduce a symbolic n
3    int n;
4    klee_make_symbolic(&n, sizeof(n), "n");
5
6    // Introduce a symbolic b
7    int fork_b = klee_int("fork_b");
8    int symsize_b;
9    // Fork into five different KLEE execution states
10   switch(fork_b) {
11     default:
12     case   1: symsize_b =   1 * sizeof(int); break;
13     case   2: symsize_b =   2 * sizeof(int); break;
14     case   4: symsize_b =   4 * sizeof(int); break;
15     case  16: symsize_b =  16 * sizeof(int); break;
16     case 128: symsize_b = 128 * sizeof(int); break;
17   }
18   int* b = malloc(symsize_b);
19   klee_make_symbolic(b, symsize_b, "b");
20
21   // Call the encapsulated function
22   mask_b(b, n);
23 }
24
25 int mask_b(int* b, int n) { /* unchanged */ }
26 int main(int argc, char** argv) { /* unchanged */ }
```

Listing 3.9: Example program during phase one

This new function is used as an entry point for a distinct KLEE analysis focusing on `mask_b`. This KLEE run will uncover the buffer overflow inside the function and because the symbolic pointer variables introduce five forks with different array sizes, KLEE will report five different test cases for this vulnerable instruction. Probably, `n` is bigger than the chosen size of `b` and `b` is filled with zeros, because this variable content is unconstrained.

For this program, phase one has two possible outcomes: Either the analysis of the main function also reveals the buffer overflow or the erroneous instruction has not been covered from main and the error remains hidden. In the latter case, the call from `main` to `mask_b` is a candidate for phase two, otherwise, the execution would already end here early because main is already the entry point of the program and has no further callers. In order to explain, what happens in phase two, please assume, that it has not been revealed from main. Furthermore, assume for simplicity, that in the reported test case `b` is filled completely with zeros and `n` is exactly one greater than the allocated size of `b`.

For phase two, the test cases found by phase one are used to build new functions. In the example, depicted in Listing 3.10, there is one of these functions, called `macke_error_mask_b` and it contains all test cases causing errors generated for `mask_b` during phase one. A call to this new function replaces all calls to `mask_b` in the program, i.e. in `main` (line 38) and `macke_mask_b_main` (line 30). Therefore, the error checks (line 4 – 22) are executed before any code from `mask_b`. If none of the error summaries can be satisfied, the function is called normally (line 24). In other words, assuming that no error occurs, `mask_b` and `macke_error_mask_b` are semantically equivalent.

In phase two MACKE starts a KLEE run from main with a targeted search for a call to `mask_b`. During this call, KLEE has to pass the for loop inside main (line 36) four times in order to consecutively increase the value of `n` to 5. During the fifth iteration of the for loop, KLEE can satisfy the third error summary (line 13 – 17) and reports an error with `klee_report_error`.

Finally, please note, that all detours (e.g. a terminating `while(true)`-loop at the end of main) inside the example program in MACKE's introductory paper, although they are omitted in this example, are no problem for MACKE. The KLEE runs in phase two stops, if the targeted search notices, that the execution state cannot reach any call of the target function anymore. So this terminating loop is never executed. The targeted search mechanism is discussed in more detail in the following section.

```
1  int macke_error_mask_b(int* b, int n) {
2    int s = klee_int("symfork");
3    switch(s) {
4      case 1:
5        if (n == 2 && klee_get_obj_size(b) == 1 &&
6            b[0] == 0) {
7          klee_report_error(/* ID for error 1 */);
8        } else {
9          klee_silent_exit();
10       }
11       break;
12     /* other cases */
13     case 3:
14     if (n == 5 && klee_get_obj_size(b) == 4 &&
15         b[0] == 0 && b[1] == 0 &&
16         b[2] == 0 && b[3] == 0) {
17       klee_report_error(/* ID for error 3 */);
18     } else {
19       klee_silent_exit();
20     }
21     break;
22     /* other cases */
23     default:
24       return mask_b(b, n);
25       break;
26   }
27 }
28 void macke_mask_b_main() {
29   /* ... introduce the symbolic variables ... */
30   macke_error_mask_b(b, n);  // changed call
31 }
32
33 int mask_b(int* b, int n) { /* unchanged */ }
34 int main(int argc, char** argv) {
35   int i, n=0, b[4] = {0,0,0,0};
36   for (i = 0; i < argc; i++) {
37     if (*argv[i] == 'b') {
38       n = macke_error_mask_b(b, n);  // changed
39     }
40   }
41 }
```

Listing 3.10: Example program during phase two

3.7. Targeted search

For the analysis in phase two, MACKE starts further KLEE runs, that aim to find the previously injected error summaries. On each forking decision during the execution, KLEE can choose which path to analyze next. Per default, it uses a search strategy that chooses the path that is most likely to reach a previously uncovered instruction. This strategy is suitable for phase one, but in phase two, MACKE is only interested in directed paths to the erroneous function that is the subject of the current analysis, so it requires a different search strategy determining the shortest path to the target.

3.7.1. Distance metric in old MACKE

Old MACKE uses a simple metric in order to determine which execution state should be explored next. It looks at the debug information inside the bitcode and extracts the file name and line number, where the target error summary is defined. Then, it compares each eligible execution state and chooses the state with the lowest difference between the line number of its position and the line number of the target, if the state is inside the same file. If none of the states shares the target's filename, a random state is selected.

This metric can be calculated very easily, but it is not very precise because it does not consider the internal structure of the program at all (e.g. the call graph or the control flow graph). Furthermore, there is no way to tell, if the target is still reachable from an execution. So the search cannot stop early if the summary is unreachable. Ideally, it analyzes the paths to the error summary first, but then it continues till the whole program is analyzed or the timeout is reached.

3.7.2. Distance metric in new MACKE

The need for a distance measure inside a program is neither new nor limited to MACKE. As mentioned in the beginning of this section, KLEE also uses a search strategy based on the minimum distance to an uncovered instruction. As another example, OTTER [39] – another framework for symbolic execution – searches the nearest usage of a symbolic variable. Although all these algorithms need a distance measure inside the source code, none of them discusses the search strategy in detail or elaborates on the implementation. Furthermore, their code is not reusable for MACKE, because it applies some special optimizations for the corresponding target and it gives no hints for its correctness (e.g. test cases). Therefore, I decided to build a generic version for targeted search and explicitly name all problems occurring during the implementation.

A search algorithm for a generic target takes a bitcode program and an execution state as an input and returns the shortest distance. An execution state is a pair of an instruction inside the program and a stack of all previously passed call statements, where the corresponding return statement has not been executed yet. In order to concretize the generic search algorithm, the user must provide two additional functions: A function, that takes an instruction and checks, if the given instruction is the target (e.g. has the

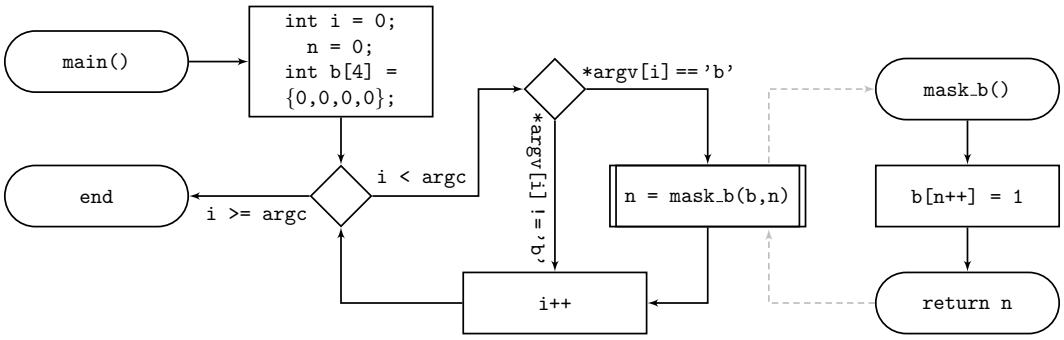


Figure 3.2.: Flowchart showing the control flow graph of the example program

instruction already been covered earlier?) and another function that takes an instruction and returns the (positive) distance it takes to pass the given instruction. Please note that there can be multiple instructions targets inside the programs.

For example, consider once again the small program in Listing 3.8. The corresponding control flow graph is depicted as a flowchart in Figure 3.2. In order to keep it simple, regard all increment operations (i.e. the ++ operator) as the target of the search and assume that each passed edge in the flowchart increases the distance by one. Then an exemplary task for the targeted search algorithm is to find the shortest distance from the `main()`-node to one of the two nodes, where the increment operator is used. Without looping, there are two eligible paths: going through the initialization block, taking the right (`i < argc`) and then the downwards (`*argv[i] != 'b'`) decision branch to end in `i++` or going through the initialization block, taking the right decision branch twice (`i < argc` and `*argv[i] == 'b'`), calling the sub-graph of `mask_b` to end in `n++`. The targeted search is looking for the shortest path, so the first route with a distance of four is returned before the second route with a distance of six. If additional routes are required, the algorithm returns alternating extensions of these two paths with circles through the first decision node.

With this initial situation, the following circumstances are challenging for an appropriate implementation.

Challenges for the implementation

The target is unreachable When calculating the distance to the target, there is no guarantee that the target is actually reachable from the current execution state. In this case, the search should return infinity – or at least the maximum integer value – but without any optimization, it is quite expensive to determine that the target is unreachable, because a straightforward implementation needs to visit each reachable instruction and verify that it is not the target, before it knows, that the target is unreachable.

Cycles in the control flow Every for or while loop inside the program introduces a cycle in the control flow graph. Exploring this part of the program must visit each instruction in the cycle once, but must also avoid looping infinitely through the cycle.

The stack is required Considering the code in Listing 3.11, the target is reachable from the entry of `foo`, only the call to `mid` has to be passed, but the target is unreachable from the entry of `bar`, even if `mid` is passed. But what about an execution state directly inside `mid`? If it has an empty stack, the target is unreachable, because it is not present in `mid`. But if the state inside `mid` is a consecutive state coming from `foo`, the call from `foo` is on the stack and the control flow can reach the target. Whereas, if the state inside `mid` is a consecutive state coming from `bar`, returning to this function is not helpful at all and the target is still unreachable. Obviously, the stack of the execution state is relevant for the reachability.

```
1  void foo() {
2    mid();
3    // <- target
4  }
5
6  void mid() {
7    // <- position of the current execution state
8    return;
9  }
10
11 void bar() {
12   mid();
13   /* no target here */
14 }
```

Listing 3.11: Small program forcing consideration of the stack

Cycles in the call graph The introduction of the stack also introduces another problem: How to deal with recursions? Similar to the cycles in the control flow graph, a recursive function must be considered once, but may not introduce infinite loops to the program exploration.

A suitable implementation

My implementation is an adapted Dijkstra's algorithm, that contains some basic sort of cycle detection. I have chosen this algorithm because it fits very well to the generic distance measure. Additionally, it has a better performance than depth-first search, if the target is quite close to the starting point of the search, which is quite common in the use case of MACKE. The complete algorithm is depicted in detail in Listing 3.12 and Listing 3.13.

A small note for all readers, that are not very familiar with LLVM bitcode: Everything inside the program is bundled into a module; a module contains multiple functions – each C function becomes a function; each function contains multiple basic blocks – similar to the branches in a control flow graph; finally each basic contains multiple instructions – comparable with assembler instructions.

```
1  uint searchMinimalDistance(ExecutionState state) {
2    PriorityQueue queue = {};
3
4    // Add the start to the search queue
5    // Distance == Priority ==  0
6    queue.push({state, 0});
7
8    while (!searchqueue.empty()) {
9      // Check, if we already hit the target
10     if (isTheTarget(queue.top())) {
11       return queue.top()->distance;
12     }
13     doSingleSearchIteration(&queue);
14   }
15   // Empty search queue and still not found
16   // -> target not reachable
17   return -1;
18 }
```

Listing 3.12: Body for the generic search algorithm

3.7.3. Specialization for new MACKE

In order to concretize this generic algorithm for the usage inside new MACKE, the two missing functions need to be defined.

As a target, I define the call instruction to the function, where MACKE has extracted and prepended the error summary. With the error summary in front of the marked call, all KLEE states that check, if the error summary can be satisfied, can still reach the target. Thereby, every instruction inside the erroneous functions cannot reach the target and the execution can be stopped earlier if the function is not recursive or is called in later instructions inside the program.

As a distance measure, I decided to count the "correct" decisions, that must be taken to reach the target. Without any suitable cost estimation for each instruction, forking during the KLEE analysis has the biggest impact on the overall computation costs. KLEE forks, whenever a basic block has multiple successors. Hence, this measure is implemented by looking at the end of each fully explored basic block and checking the number of possible successor blocks. If there is more than one successor, this is considered as a correct decision KLEE has to take, because KLEE has to continue the exploration with the best path to the target.

Please note, that the above algorithm does not use any sort of caching or shortcut to simplify the distance calculation, especially for situations, where the target is not (or no longer) reachable. Nevertheless, MACKE only calculates distances from a function to a direct callee – so the distance is quite short – and KLEE stops the execution on states that cannot reach the target. So I deem this slowness to be acceptable, in favor of keeping the implementation clear and well tested. Building a bullet-proof and perfectly optimized search strategy is beyond the scope of this thesis.

```
19 void doSingleSearchIteration(PriorityQueue* queue) {
20   ExecutionState curr = queue->pop()->state;
21
22   if (isa<CallInst>(curr.instruction)) {
23     // If call, increase stack and add to search queue
24
25     // Extract the called function
26     Function* called = curr.instruction->getCalledFunction();
27
28     // Check if the function is an external call
29     if (called && !called->isIntrinsic() && !called->empty()) {
30       // Avoid recursions
31       if (!curr.alreadyInsideStackOf(curr))) {
32         // Add the current call instruction to the stack and enqueue
33         queue->push({{called->entry(), curr.stack + curr.instruction},
34             curr.distance + distanceToPass(curr.instruction)});
35       }
36     } else {
37       // Just skip the called function and continue
38       queue->push({{succ(curr.instruction), curr.stack},
39           curr.distance + distanceToPass(curr.instruction)});
40     }
41   } else if (isa<ReturnInst>(curr.instruction)) {
42     // If return, add last entry from stack
43
44     // Check, if we have any point to return to
45     if (!curr.stack.empty()) {
46       // Go back to top entry of the stack
47       queue->push({{curr.stack.head(), curr.stack.tail()},
48           curr.distance + distanceToPass(curr.instruction)});
49     }
50     // Otherwise, this branch ends here
51   } else if (isa<TerminatorInst>(curr.instruction)) {
52     // If terminal instruction of a basic block, add all successor
53
54     // Get access to the current block
55     BasicBlock* currblock = curr.instruction->getParent();
56
57     // Iterate over all the successors
58     for (BasicBlock* nextblock : currblock->successors) {
59       // And add their first instruction to the search queue
60       if (!queue->wasAddedBefore({nextblock->entry(), curr.stack})) {
61         queue->push({{nextblock->entry(), curr.stack},
62             curr.distance + distanceToPass(curr.instruction)});
63       }
64     }
65   } else {
66     // All other instructions just add their successor
67     queue->push({{succ(curr.instruction), curr.stack},
68         curr.distance + distanceToPass(curr.instruction)});
69   }
```

Listing 3.13: Inside the loop of the generic search algorithm

3.8. Scheduling and parallelization

The old MACKE implementation [45] is not parallelized at all and the paper about MACKE only states, that phase one "may be parallelized efficiently" [46, subsection 2.2.1]. In fact, both phases can be parallelized at least partially. This section discusses the scheduling algorithms used in the new implementation. For an evaluation of their effectiveness please have a look at Section 4.6.

Both algorithms only aim to run multiple KLEE analyses at the same time, because the majority of the MACKE runtime is spent inside KLEE. All LLVM bitcode operations could be parallelized as well if multiple copies of almost the same bitcode file are used. I renounced this because it heavily increases the storage needed for an analysis, with only a very small benefit for the run-time. The portion on the run-time of error matching and composing the error reports is hardly notable and therefore, they should not be parallelized at all.

3.8.1. Parallelization in phase one

None of the KLEE-runs in phase one interfere in any way with another KLEE analysis. Therefore, theoretically, all of them can be started at the same time and run in parallel. However, most of the today's computers do not have enough cores for this degree of parallelization. Furthermore, the run-time of the different KLEE-runs varies hugely: The deeper and smaller functions finish within a few seconds, whereas the higher and bigger functions quite certainly hit the timeout of two minutes.

In order to accommodate these requirements, MACKE creates a pool of parallel worker threads – ideally one per core – and puts all the KLEE-runs into a synchronized job queue. Each worker takes the next job out of the queue and executes the demanded KLEE analysis. After his analysis finishes (or reaches the timeout), the worker stores the results and continues with the next job until the queue is empty. For more determinism, the queue of KLEE-runs is sorted with the same inverted topological order taken from the call graph, i.e., starting with the leaf nodes up to the node of the main function, before the execution begins.

3.8.2. Scheduling in phase two

In phase two, the situation is more complicated. Each KLEE-run requires the errors summary that is prepended to the analyzed call and this summary might come from another analysis in phase two. In other words, before a call can be completely analyzed, the analysis of all deeper calls in the call graph must be completed.

An alternative strategy

It is a valid strategy to simply ignore errors found by phase two at first. With this assumption, the first steps in phase two can be parallelized as explained in phase one, but then, several analyses must be run again with the errors previously found in phase two. If any of these new error summaries can be satisfied, this new errors must be analyzed again. In the worst case, a single error must be propagated from a leaf node to the entry function, requiring dozens of analyses to be run again. If there are more than one of these errors spread over the program, this results in a cascade of repeated analyses. Therefore, I decided to implement a scheduler, that needs more synchronization but analyzes each call just once.

MACKE's approach

The MACKE scheduling for phase two relies similar to phase one on a pool of workers and a queue of KLEE-runs, but this time, the queue is split into several groups that must be analyzed completely before the processing of the next group in the queue can begin.

While explaining the basic idea about grouping, please assume, that the call graph of the program does not contain any cycles, i.e., the program does not use recursive function calls. MACKE looks at the call graph and puts all calls to leaf nodes into one group and adds it to the queue. Then the leaf nodes are removed from the call graph and MACKE continues with the calls to the leaf nodes in the modified graph until the whole call graph is scheduled. This strategy does not require any call to be analyzed twice because each caller of a function is enqueued in a later group and thereby all deeper calls are completely analyzed before their result is used by a higher function.

If there are cycles in the call graph, this simple strategy faces a problem: At some point, there is no remaining leaf in the graph but not all nodes – especially the ones inside the cycle – are scheduled. In this situation MACKE identifies the deepest cycle in the graph and schedules all these nodes in the cycle differently: for each node of the cycle, the calls are enqueued in a separate group containing only calls from this single node and afterward, the whole cycle is removed from the call graph. Afterward, the scheduling can continue normally.

This scheduler has a drawback: It puts each cycle node only once into the queue and even worse – except for the last node in the cycle – while not all of its callers are completely analyzed. This prevents possible errors from propagating through the complete cycle. Although this is a rare corner case, this might hide some errors in the analyzed program.

Nevertheless, cycles are also a problem for the alternative strategy presented earlier. If an error can go through the cycle once, it might be the case, that it can go through it multiple times or even infinitely often. Therefore, also the alternative scheduler is missing a mechanism to prevent this infinite propagation. In order to keep the scheduling mechanism simple, I have decided to exclude any further cycle analysis and accept the minor risk to miss such rare kind of error.

Potential for further optimization

Both presented schedulers for phase two do not consider, if a function is affected by any error, that should be prepended, at all. If there is nothing to analyze for a scheduled call, MACKE just skips the execution and continues with the next job from the queue. This forces the chosen scheduler to build more groups than are required to satisfy the requirement to complete the analysis of all callees, before the caller gets analyzed. For the sake of having a textual representation in the following example, a short syntax for these schedules is defined as follows: (a, b) represents a call from a to b, that should be analyzed, $\{(\ldots),\ \ldots\}$ represents a group of tasks, that can be independently analyzed in parallel, and finally $[\{\ldots\},\ \ldots]$ represents a list of task groups, denoting their order of execution.

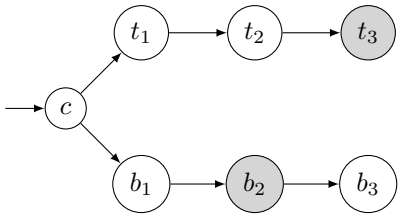


Figure 3.3.: Example control flow graph with sub-optimal parallelization

With this notation in mind, consider the call graph depicted in Figure 3.3, and assume, that t_3 and b_2 trigger an error in phase one. MACKE would schedule phase two as $[\{(t_2, t_3), (b_2, b_3)\}, \{(t_1, t_2), (b_1, b_2)\}, \{(c, t_1), (c, b_1)\}]$. If no error gets propagated, only one KLEE-run is executed at once. But with the knowledge of all previously occurring errors before and after each group, the scheduler could form less groups, that do more work in parallel. For example, it can see, that (b_2, b_3) does not need to be scheduled because it has no errors to propagate. With this information, the scheduler can put (t_2, t_3) and (b_1, b_2) into the same group. So, if none of them propagates an error, an optimal schedule for this situation is $[\{(t_2, t_3), (b_1, b_2)\}]$, which executes two KLEE-runs in parallel. Defining a scheduler that reaches a maximum degree of parallelization in all situations, is beyond the scope of this thesis, but please note, that there is some potential for further optimizations.

4. Evaluation

After implementing all previously discussed improvements to the algorithms, this chapter focuses on the evaluation of the aspired benefits. After a closer look at the 16 programs, that I have selected for the evaluation, and the setup used for the evaluation, the following sections discusses a lot of numbers correlating with different properties of the algorithm: A general overview identifying outlier programs, a direct comparison with old MACKE, an analysis of the realized degree of parallelization, a closer look at the discovered vulnerabilities and finally a detailed examination of the achieved coverage.

4.1. Analyzed programs

For the evaluation, I have selected 16 different open-source C programs from a variety of application fields. In addition to the older versions used for comparison with old MACKE, I have chosen for each program the most current version available at the time of this thesis. Most of the programs can be clustered in one of the following five categories:

Compression algorithms

A part of the selected programs are implementations of compression algorithms. `bzip2` creates .bz files, `lz4` creates .xz files, `zopfli` is an alternative way to generate .gz files. Although the main application for `tar` is concatenating multiple files to one .tar file, it also includes several compression algorithms (e.g. zip) to compress the output file subsequently.

Reading and parsing file contents

Some of the programs read content from a file and interact with it. `diff` takes two files and compares them line by line. `grep` prints all lines, that match a pattern given by the user. `less` reads the file partially and allows the user to navigate inside the file.

Server

The collection of analyzed programs also contains some servers. `goahead` is a complete, embedded web server. `ngircd` is a server for IRC-based chats. `libuv` is the asynchronous I/O library behind `node.js` and thereby used internally by many web servers.

Stream modification

A couple of programs are designed for stream manipulation. `sed` can filter and transform the input as requested by a pattern given by the user. `jq` manipulates JSON data and combines it into new JSON output described by user input.

Code generators

A few programs are generators for source code. `bison` generates a parser for a given context-free grammar. `flex` generates a scanner for lexical analysis of source code.

Miscellaneous

Finally, there are two programs that do not fit into the above categories. `bc` is a math library for arbitrary precision calculations. `coreutils` is a collection of very basic command line tools on UNIX-like operation systems for text, file and shell interaction.

Program sizes

The analyzed programs consist of medium-scale programs (like `zopfli`) and large-scale programs (like `tar`, which is around ten times bigger). Table 4.1 gives details about the size of each program, denoting their lines of code and defined functions.

Program	LOC	Functions	Program	LOC	Functions
bc 1.06	3.5 *k*	129	grep 2.25	8.0 *k*	461
bison 3.0.4	15.3 *k*	1076	grep SIR	3.6 *k*	119
bzip2 1.0.6	3.3 *k*	108	jq 1.5	8.9 *k*	572
coreutils 6.10	38.7 *k*	1400	less 481	7.9 *k*	459
coreutils 8.25	42.6 *k*	1683	libuv 1.9.1	5.9 *k*	479
diff 3.4	7.8 *k*	391	lz4 r131	4.7 *k*	205
flex 2.6.0	6.5 *k*	260	ngircd 23	9.0 *k*	574
flex SIR	3.8 *k*	144	sed 4.2.2	3.2 *k*	213
goahead 3.6.3	17.3 *k*	958	tar 1.29	20.7 *k*	1214
			zopfli 1.0.1	1.9 *k*	103

Table 4.1.: Size of the analyzed programs

4.2. Definitions

This section defines all the terms used during the evaluation. Furthermore, it explains how the corresponding figures are counted or calculated. Please note, that some of them are also used by the original MACKE paper [46], but are defined differently. This is pointed out inside the definition of the affected term, where you also find the reason for the adaptation.

Lines of Code (LOC) is the number of lines inside all files considered during the compilation (i.e. `.c` and `.h` files) that have at least one corresponding instruction inside the LLVM bitcode. In order to count these lines, the analyzed program is compiled with debug information enabled and without any optimization (i.e. using `-g -O0`). Then, while iterating over all instructions inside the resulting bitcode file, each referenced file and line number inside the debug information is counted once.

This number is way lower than in old MACKE. For the old measure, all lines inside the code are counted (i.e. `wc -l *.c`). This also counts empty lines, as well as comments, that are both excluded in the new measure. The new measure is expected to offer a more proportional comparison between different programs because it only considers code that actually represents some functionality in the analyzed program.

Function refers to the number of functions that have a signature as well as an executable body inside the bitcode file. Functions with an explicit inline attribute have no distinct signature inside the bitcode and are therefore not considered for this measure.

Coverage is the proportion of LOC covered by at least one test case generated by KLEE and all LOC present in any KLEE run. The input for KLEE is a completely non-optimized bitcode file, that is later optimized by KLEE before the actual analysis takes place. This means that dead code gets eliminated and explicit constructions with multiple generic instructions may be combined to fewer, more specific instructions. Thereby, some code is optimized away and cannot be reached by a KLEE run, because it is no longer part of the analyzed program. This code is included in LOC but is not considered for the coverage. The proportion is measured by combining all run.istats files of the triggered KLEE runs, that denotes which line of code was visited how often inside the corresponding KLEE run. All LOC that are not present in any of these files, are considered as *optimized* away, all the remaining lines are considered as *present*. All LOC that are visited at least once by at least one KLEE run are considered as *covered*. The fraction *covered* divided by *present* denotes the overall coverage.

Function coverage is a generalization of the coverage based on LOC. Most of LOC represented in the bitcode (except for some global constants) belongs to one function. This allows a projection on all functions of a program. A function with at least one covered LOC, is considered as covered. A function with no covered LOC, but at least one uncovered LOC, is considered as uncovered. Finally, a function with no covered or uncovered LOC, i.e. only with LOC that are optimized away, is considered as removed. Removed functions are most likely functions that were inlined by the compiler and/or constants progageted through the code. The ratio of covered functions is defined by covered functions divided by the total number of functions in the program.

KLEE error is an error reported by KLEE during a single analysis that is caused by one of the following reasons: invalid pointer usage (`.ptr.err`), a call to free of previously not allocated storage (`.free.err`), a division by zero (`.div.err`) or a violated assert statement in the code (`.assert.err`).

Vulnerable Instruction (Vuln.Inst.) is a line of code in a specific source file of the analyzed program that is the reason for at least one KLEE error. Multiple KLEE errors can correspond to a single vulnerable instruction, because with different entry points in the program, the same error can be triggered in different KLEE runs. This term is the adequate representation of the errors found by MACKE if all term should be reduced to just one error definition. The vulnerable instructions are collected by a deeper look into all KLEE errors, which denote the line and file causing the error.

Error chain is a path through the transposed call graph just visiting functions that are affected by the same vulnerable instruction, and that is not a subsequence of a longer chain. More formally, this can be expressed as follows: $[f_0, f_1, \ldots, f_n]_{VI}$ is an error chain of a vulnerable instruction VI, if f_i is called by f_{i+1}, each f_i is affected by VI, f_0 does not call any function affected by VI, and none of the callers to f_n are affected by VI. The same path can be listed multiple times, if it can be caused by different vulnerable instructions. In order to keep the chain length finite, possible cycles inside the call graph are only taken once.

$|chain| > 1$ is the number of error chains that are spread at least over two functions. This is measured by looking at the list of all error chains and counting all of them that cover more than one function. This number is called 1-level-up in old MACKE.

$chain \prec P2$ is the number of error chains that end with an error found by phase two of MACKE. That means that the last KLEE error inside the chain has been prepended to the next function in the chain and a targeted search run of this function has triggered the error by its summary. Multiple successes in phase two inside the same chain are not counted twice. This is measured by looking at the list of all error chains and counting all of them that end with an error found by phase two.

main exploit is the number of vulnerable instructions that can be triggered by a generated input from the main function of the program. This is counted by looking at all affected functions for each vulnerable instruction and counting the occurrences of the main function.

library exploit is the number of vulnerable instructions that can be triggered by a generated input from a library function. All functions that do not have any callers and are not the main function are considered as library functions. This is measured by counting all vulnerable instructions that affect at least one function that is qualified as a library function. For example, all functions forming an API are library functions because the analyzed code intentionally does not contain any information about their usage.

runtime is the amount of time passed on the clock from the start to the end of the program. This number does not consider any parallelism. It is calculated by the difference between the time, when the program was started and when the program has finished its calculations. The abbreviation “min” denotes the unit of time and stands for minutes.

runt.P1 and runt.P2 denote the runtime, but only taking phase 1 or phase 2 into account for their measurement.

DOP1 and DOP2 are the degree of parallelism during the execution in phase 1 or phase 2. It is calculated as the ratio of the sum of the runtime for all single threaded KLEE runs and the runtime of the MACKE phase.

4.3. Setup for the tests

All tests were run on a quad core machine with 2.5 GHz per core and 32 GB of memory, running a 64bit Linux 4.4.0. Only one MACKE run was executed at once, using at most 4 threads with in total 4 GB memory usage for its parallelized tasks. With KLEE being single threaded, four different experiments were run in parallel for the best utilization of the test machine, without cutting the resources required by a single run.

The experiments with KLEE vary a little in their results, because there is some randomness included in the calculation, e.g. the underlying SAT solver does not always return the same counterexamples, the KLEE search strategy chooses the next instruction based on an adapted random distribution [10, section 3.4], and all this fills the internal caches differently, which affects the overall run-time. MACKE starts hundreds of KLEE runs and therefore its results vary even more.

In order to anticipate the threat to the validity of the results that is caused by this variety, I decided to repeat all experiments five times and report the results for all five runs in detail in the tables in this chapter. This aims to keep the size of the tables small and handy, while no correlation between the different numbers is hidden inside averages or deviations, even if it is not covered by any analysis in this thesis. The lines are given in the same order arbitrary order, I have received them during the experiments. Please note, that this chapter gives multiple tables about different aspect about the analysis of the same program and all these tables use the same order. Thereby, the nth row in the tables of the same program belongs to the same run of MACKE or KLEE respectively.

For MACKE, I decided to keep the old 2 minutes per function time limit, but for KLEE, I dropped the old, constant 2 hours per program time limit. Assigning a time limit per function scales well with the size of the program under test, whereas a static time limit favors small programs and penalizes bigger programs. In order to build a fair competition with KLEE, I measure the longest run-time MACKE takes to analyze the program under test and use this as a time limit for KLEE on the same program. I calculate the run-time as the difference between the end and start time on the clock, ignoring the degree of parallelization inside MACKE. On the one hand, the four parallel threads seem to be a big advantage for MACKE, but on the other hand, MACKE often considers the same instruction multiple times for different components. So overall this should give an equal chance to both programs.

For details, how to obtain the required source code, please see Appendix A.

Table 4.2.: Results for bc 1.06 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	**max** $\|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	28.2 %	65	39	6	16	5	47 *min*		16.5 %	0	47 *min*
	27.9 %	66	38	6	16	5	45 *min*		15.6 %	0	47 *min*
	27.6 %	66	37	6	16	5	46 *min*		16.0 %	0	47 *min*
	28.0 %	66	38	5	13	5	46 *min*		15.6 %	0	47 *min*
	28.0 %	65	39	6	16	5	46 *min*		17.2 %	0	47 *min*
⌀	27.9 %	65.6	38.2	5.8	15.4	5.0	45.9	⌀	16.2 %	0.0	46.7
σ	0.2 %	0.5	0.8	0.4	1.3	0.0	0.8	σ	0.7 %	0.0	0.2

Table 4.3.: Results for bison 3.0.4 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	**max** $\|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	41.9 %	452	489	9	69	1	704 *min*		9.9 %	0	708 *min*
	41.9 %	450	490	9	73	1	705 *min*		9.2 %	0	708 *min*
	41.8 %	455	495	9	65	1	708 *min*		9.9 %	0	708 *min*
	41.4 %	451	507	9	66	1	701 *min*		13.9 %	0	708 *min*
	41.8 %	451	491	9	67	1	696 *min*		9.9 %	0	708 *min*
⌀	41.8 %	451.8	494.4	9.0	68.0	1.0	702.9	⌀	10.6 %	0.0	708.2
σ	0.2 %	1.9	7.4	0.0	3.2	0.0	4.5	σ	1.9 %	0.0	0.1

Table 4.4.: Results for bzip2 1.0.6 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	**max** $\|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	37.5 %	66	21	3	4	0	44 *min*		8.9 %	0	49 *min*
	37.8 %	65	19	3	4	0	49 *min*		9.2 %	0	49 *min*
	37.9 %	70	21	3	5	0	45 *min*		8.9 %	0	49 *min*
	38.1 %	71	20	3	4	0	49 *min*		8.9 %	0	49 *min*
	39.6 %	76	21	3	4	0	47 *min*		9.2 %	0	49 *min*
⌀	38.2 %	69.6	20.4	3.0	4.2	0.0	46.7	⌀	9.0 %	0.0	49.1
σ	0.8 %	4.4	0.9	0.0	0.4	0.0	2.1	σ	0.2 %	0.0	0.1

Table 4.5.: Results for coreutils 6.10 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	$\max \|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	51.6 %	177	5	2	0	52	455 *min*		50.4 %	28	476 *min*
	51.0 %	176	5	2	0	52	452 *min*		50.7 %	28	476 *min*
	51.1 %	175	5	2	0	52	452 *min*		50.2 %	28	474 *min*
	51.2 %	172	5	2	0	51	452 *min*		50.7 %	28	475 *min*
	50.9 %	175	5	2	0	52	450 *min*		50.7 %	28	474 *min*
⌀	51.2 %	175.0	5.0	2.0	0.0	51.8	452.3	⌀	50.6 %	28.0	474.8
σ	0.3 %	1.9	0.0	0.0	0.0	0.4	1.7	σ	0.2 %	0.0	0.8

Table 4.6.: Results for coreutils 8.25 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	$\max \|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	50.5 %	158	8	2	0	36	529 *min*		51.5 %	4	556 *min*
	50.7 %	157	9	2	0	38	531 *min*		51.2 %	4	554 *min*
	51.0 %	159	8	2	0	37	527 *min*		51.8 %	4	551 *min*
	51.0 %	157	8	2	0	37	526 *min*		52.4 %	4	545 *min*
	50.6 %	162	8	2	0	37	527 *min*		51.1 %	4	556 *min*
⌀	50.8 %	158.6	8.2	2.0	0.0	37.0	528.3	⌀	51.6 %	4.0	552.6
σ	0.2 %	2.1	0.4	0.0	0.0	0.7	2.1	σ	0.5 %	0.0	4.7

Table 4.7.: Results for diff 3.4 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	$\max \|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	57.0 %	258	173	6	6	1	205 *min*		18.2 %	1	210 *min*
	58.1 %	258	174	6	7	1	209 *min*		18.4 %	1	146 *min*
	58.0 %	253	175	6	6	1	209 *min*		17.6 %	1	210 *min*
	57.8 %	253	176	6	8	1	206 *min*		17.6 %	1	210 *min*
	57.4 %	252	175	6	6	1	205 *min*		17.6 %	1	162 *min*
⌀	57.7 %	254.8	174.6	6.0	6.6	1.0	206.9	⌀	17.9 %	1.0	187.5
σ	0.5 %	2.9	1.1	0.0	0.9	0.0	2.3	σ	0.4 %	0.0	31.0

Table 4.8.: Results for flex 2.6.0 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	$\max \|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	34.5 %	120	61	4	8	0	120 *min*		9.1 %	0	142 *min*
	34.5 %	120	69	4	7	0	120 *min*		8.8 %	0	157 *min*
	34.4 %	120	63	4	8	0	119 *min*		8.8 %	0	157 *min*
	34.3 %	117	57	4	8	0	119 *min*		8.8 %	0	157 *min*
	34.4 %	117	59	4	7	0	119 *min*		8.8 %	0	157 *min*
⌀	34.4 %	118.8	61.8	4.0	7.6	0.0	119.6	⌀	8.9 %	0.0	153.7
σ	0.1 %	1.6	4.6	0.0	0.5	0.0	0.7	σ	0.1 %	0.0	6.6

Table 4.9.: Results for flex SIR orig.v0 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	$\max \|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	30.2 %	95	53	7	8	5	104 *min*		15.4 %	3	105 *min*
	29.9 %	94	65	9	7	5	104 *min*		21.1 %	3	105 *min*
	30.1 %	98	74	9	17	5	104 *min*		17.2 %	3	105 *min*
	30.5 %	94	66	9	8	5	105 *min*		20.5 %	3	106 *min*
	30.0 %	94	65	9	8	5	104 *min*		16.6 %	3	107 *min*
⌀	30.1 %	95.0	64.6	8.6	9.6	5.0	104.4	⌀	18.2 %	3.0	105.8
σ	0.2 %	1.7	7.5	0.9	4.2	0.0	0.4	σ	2.5 %	0.0	1.1

Table 4.10.: Results for goahead 3.6.3 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	$\max \|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	45.8 %	507	675	8	31	0	1012 *min*		0.0 %	0	0 *min*
	46.2 %	505	665	11	30	0	1008 *min*		0.0 %	0	0 *min*
	45.6 %	503	654	8	23	0	1009 *min*		0.0 %	0	0 *min*
	45.3 %	510	650	11	21	0	1008 *min*		0.0 %	0	0 *min*
	45.4 %	507	674	11	39	0	1006 *min*		0.0 %	0	0 *min*
⌀	45.7 %	506.4	663.6	9.8	28.8	0.0	1008.8	⌀	0.0 %	0.0	0.1
σ	0.4 %	2.6	11.4	1.6	7.2	0.0	2.2	σ	0.0 %	0.0	0.0

Table 4.11.: Results for grep 2.25 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	$\max \|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	56.7 %	271	138	5	5	0	234 *min*		42.7 %	0	240 *min*
	56.3 %	265	133	5	6	0	237 *min*		40.6 %	0	238 *min*
	56.7 %	268	135	5	10	0	232 *min*		42.2 %	0	239 *min*
	56.4 %	266	135	5	9	0	235 *min*		43.6 %	0	238 *min*
	55.9 %	270	137	5	8	0	235 *min*		38.0 %	0	228 *min*
⌀	56.4 %	268.0	135.6	5.0	7.6	0.0	234.6	⌀	41.4 %	0.0	236.5
σ	0.3 %	2.5	1.9	0.0	2.1	0.0	2.1	σ	2.2 %	0.0	4.9

Table 4.12.: Results for grep SIR orig.v0 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	$\max \|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	58.5 %	125	20	4	0	2	60 *min*		48.7 %	2	60 *min*
	57.6 %	116	18	4	0	2	59 *min*		52.8 %	2	60 *min*
	57.8 %	118	18	4	0	2	59 *min*		53.4 %	2	60 *min*
	57.8 %	121	19	4	0	2	60 *min*		53.4 %	2	60 *min*
	57.4 %	124	19	4	0	2	60 *min*		46.8 %	2	60 *min*
⌀	57.8 %	120.8	18.8	4.0	0.0	2.0	59.5	⌀	51.0 %	2.0	60.1
σ	0.4 %	3.8	0.8	0.0	0.0	0.0	0.5	σ	3.1 %	0.0	0.1

Table 4.13.: Results for jq 1.5 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	$\max \|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	12.5 %	0	0	0	0	0	15 *min*		19.5 %	0	15 *min*
	12.5 %	0	0	0	0	0	15 *min*		18.9 %	0	15 *min*
	12.5 %	0	0	0	0	0	15 *min*		19.4 %	0	15 *min*
	12.5 %	0	0	0	0	0	15 *min*		19.3 %	0	15 *min*
	12.6 %	0	0	0	0	0	15 *min*		19.4 %	0	15 *min*
⌀	12.5 %	0.0	0.0	0.0	0.0	0.0	14.8	⌀	19.3 %	0.0	15.1
σ	0.0 %	0.0	0.0	0.0	0.0	0.0	0.1	σ	0.3 %	0.0	0.0

Table 4.14.: Results for less 481 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	**max** $\|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	45.0 %	148	96	4	12	1	106 *min*		2.4 %	1	0 *min*
	44.4 %	145	94	4	13	1	103 *min*		2.4 %	1	0 *min*
	43.6 %	146	97	4	13	1	104 *min*		2.4 %	1	0 *min*
	44.7 %	150	98	4	12	1	103 *min*		2.4 %	1	0 *min*
	44.0 %	150	96	4	13	1	104 *min*		2.4 %	1	0 *min*
⌀	44.3 %	147.8	96.2	4.0	12.6	1.0	104.0	⌀	2.4 %	1.0	0.1
σ	0.6 %	2.3	1.5	0.0	0.5	0.0	1.2	σ	0.0 %	0.0	0.0

Table 4.15.: Results for lz4 r131 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	**max** $\|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	70.6 %	85	134	4	2	1	96 *min*		0.0 %	1	7 *min*
	71.2 %	86	132	4	1	1	97 *min*		0.0 %	1	7 *min*
	70.1 %	89	135	4	4	1	96 *min*		0.0 %	1	8 *min*
	70.3 %	86	133	4	3	1	97 *min*		0.0 %	1	0 *min*
	67.6 %	84	133	4	1	1	100 *min*		14.6 %	1	13 *min*
⌀	70.0 %	86.0	133.4	4.0	2.2	1.0	97.4	⌀	2.9 %	1.0	6.9
σ	1.4 %	1.9	1.1	0.0	1.3	0.0	1.9	σ	6.5 %	0.0	4.4

Table 4.16.: Results for ngircd 23 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	**max** $\|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	41.6 %	165	571	6	2	1	494 *min*		0.0 %	0	1 *min*
	41.4 %	163	519	6	2	1	493 *min*		0.0 %	0	5 *min*
	42.7 %	165	618	6	3	1	492 *min*		0.0 %	0	4 *min*
	41.1 %	165	614	6	2	1	495 *min*		0.0 %	0	0 *min*
	42.5 %	168	540	6	3	1	498 *min*		0.0 %	0	0 *min*
⌀	41.9 %	165.2	572.4	6.0	2.4	1.0	494.5	⌀	0.0 %	0.0	2.1
σ	0.7 %	1.8	43.9	0.0	0.5	0.0	2.0	σ	0.0 %	0.0	2.1

Table 4.17.: Results for sed 4.2.2 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	**max** $\|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	64.2 %	117	103	5	10	0	113 *min*		0.0 %	0	6 *min*
	63.6 %	112	100	5	10	0	117 *min*		0.0 %	0	5 *min*
	65.4 %	116	97	5	5	0	116 *min*		0.0 %	0	5 *min*
	66.0 %	115	101	5	9	0	114 *min*		65.8 %	0	118 *min*
	64.0 %	118	103	5	8	0	113 *min*		65.8 %	0	117 *min*
⌀	64.6 %	115.6	100.8	5.0	8.4	0.0	114.4	⌀	26.3 %	0.0	50.1
σ	1.0 %	2.3	2.5	0.0	2.1	0.0	1.8	σ	36.0 %	0.0	61.5

Table 4.18.: Results for tar 1.29 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	**max** $\|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	52.5 %	580	598	8	10	1	932 *min*		0.2 %	1	0 *min*
	52.8 %	580	604	11	15	1	918 *min*		0.2 %	1	0 *min*
	52.6 %	602	599	8	11	1	926 *min*		0.2 %	1	0 *min*
	52.1 %	590	584	11	10	1	927 *min*		0.2 %	1	0 *min*
	52.6 %	577	603	8	14	1	928 *min*		0.2 %	1	0 *min*
⌀	52.5 %	585.8	597.6	9.2	12.0	1.0	926.1	⌀	0.2 %	1.0	0.2
σ	0.3 %	10.3	8.0	1.6	2.3	0.0	5.2	σ	0.0 %	0.0	0.0

Table 4.19.: Results for zopfli 1.0.1 with MACKE (left) and KLEE (right)

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	**max** $\|chain\|$	$chain \prec P2$	**main exp.**	**runtime**		**Coverage**	**main exp.**	**runtime**
	55.5 %	68	20	4	1	0	70 *min*		8.9 %	0	72 *min*
	56.6 %	66	20	4	1	0	72 *min*		8.9 %	0	72 *min*
	55.2 %	65	20	4	1	0	72 *min*		8.9 %	0	72 *min*
	55.5 %	64	17	4	0	0	71 *min*		8.9 %	0	72 *min*
	57.6 %	66	18	4	0	0	69 *min*		8.9 %	0	72 *min*
⌀	56.1 %	65.8	19.0	4.0	0.6	0.0	70.8	⌀	8.9 %	0.0	72.2
σ	1.0 %	1.5	1.4	0.0	0.5	0.0	1.5	σ	0.0 %	0.0	0.0

Table 4.20.: Results for libuv 1.9.1 with MACKE

	Coverage	**Vuln.Inst.**	$\|chain\| > 1$	$\max \|chain\|$	$chain \prec P2$	**main exp.**	**runtime**
	57.4 %	237	88	5	2	0	182 *min*
	57.8 %	231	84	5	3	0	182 *min*
	56.6 %	240	88	5	3	0	184 *min*
	57.1 %	235	84	5	2	0	185 *min*
	58.3 %	233	85	5	5	0	183 *min*
$\varnothing$	57.4 %	235.2	85.8	5.0	3.0	0.0	183.2
σ	0.7 %	3.5	2.0	0.0	1.2	0.0	1.1

4.4. General overview

Before the detailed discussion of several hypotheses in the following sections, this section gives a general overview of the results of the evaluation. The numbers from the comparison of MACKE with KLEE on the 16 programs are depicted in Table 4.2 to Table 4.20. Please note that two different versions of three programs are analyzed, resulting in different 19 tables each for MACKE and KLEE. The only exception forms `libuv`. It is a library and therefore it cannot be analyzed with KLEE. The first five rows correspond to one of the five repeated runs for the analysis and the last two rows denote the averages and standard deviation of all runs.

Considering the deviation of the five runs on the same program some variations seems to be normal with MACKE, although the results of KLEE vary less for the same program. The multiple KLEE runs inside MACKE probably multiply the variations, especially, with the second phase relying on the results of the runs in phase one.

The numbers for KLEE are peculiar for six programs: for `less` and `tar` KLEE runs regularly, but it terminates early and only generates one test case. Most likely, KLEE is not able to solve a path condition leading to the main parts of the program, but I may also have used unappropriate flags for the analysis of these programs. For `goahead` KLEE is unable to initialize a global symbol and denies the analysis completely. The analysis of `lz4` cuts the memory cap and stops early. All runs for `ngircd` and three of the runs for `sed` report a failed fork for the SMT-solver and stop the analysis. In spite of these problems also affecting some KLEE runs started by MACKE, the main part of its runs are executed free from any kind of defects.

4.5. Comparison with old MACKE

This thesis suggests a lot of minor and major changes to the old MACKE implementation and all of them intend to improve its capabilities. This section revisits the analysis of the old implementation and compares the results with the outcome of the new implementation on the same programs.

All the old results are extracted from [46, table 1]. There, the authors have decided to put the results for all programs into one table, but here, the multiple repeats of one analysis can be represented more clearly in separate tables for MACKE and KLEE and for each program. See Table 4.21 up to Table 4.24 for MACKE and Table 4.25 up to Table 4.28 for KLEE. The first five data rows represent results of the new implementation, the last line cites the corresponding numbers from the MACKE paper.

First of all, please note that the column names of these cited tables are a bit different to the ones in the old MACKE paper: The column "1-level-up" is named $|chain| > 1$. Furthermore, the KLEE tables are missing the vulnerable instruction column, because each vulnerable instruction in KLEE is also a main exploit, so these two numbers are always equal. Finally, the column $chain \prec P2$ is not present in the paper, but after a request, the authors of old MACKE have provided this number for better comparability.

Considering the data for KLEE, the perfect reproducibility of the results seems to be limited. For `grep` and `coreutils`, the coverage seems to be almost comparable. It is slightly higher, but this is likely caused by a moderately longer time frame for the analysis and the more recent version of KLEE. Unfortunately, the paper does not give an exact time limit for `coreutils` nor the exact KLEE version used. Nevertheless, this does not explain the big difference on `bzip2` and `flex`. For `bzip2`, the achieved coverage is doubled and for `flex`, it even increases by the factor three. I have no reasonable explanation for these differences, probably the minor changes in KLEE are more beneficial for these programs. Nevertheless, it is an indication that the comparisons of these two programs are less meaningful than the other two. Furthermore, KLEE finds more main exploits as reported in the paper. This is quite likely caused by different flags used for the KLEE analysis, especially turning also the program name into a symbolic variable.

Considering the data for new MACKE, the overall coverage is similar to old MACKE. For `coreutils`, the coverage achieved with old and new MACKE are almost identical. On `grep`, the achieved coverage is slightly better than with old MACKE. For the remaining two programs – `bzip2` and `flex` – with a big aberration on the KLEE runs discussed earlier, the coverage achieved with MACKE also differs. On `bzip2`, the coverage is lower, but on `flex`, it is increased. With both tools using the same time limits, this is probably caused by the more complicated array logic in new MACKE. This logic requires more execution time, but is capable of covering more complicated paths in the program. On the one hand, if the simple arrays in old MACKE are enough to cover most of the paths, this slows down the analysis and fewer paths are covered at the end of the time limit. On the other hand, if the program requires the more complex array logic, these paths can not be covered with old MACKE.

Table 4.21.: Comparison with old MACKE on bzip2 1.0.6

Coverage	Vuln.Inst.	$\|chain\| > 1$	$chain \prec P2$	main exploit
37.5 %	66	21	4	0
37.8 %	65	19	4	0
37.9 %	70	21	5	0
38.1 %	71	20	4	0
39.6 %	76	21	4	0
53 %	106	16	0	0

Table 4.22.: Comparison with old MACKE on grep SIR orig v0

Coverage	Vuln.Inst.	$\|chain\| > 1$	$chain \prec P2$	main exploit
58.5 %	125	20	0	2
57.6 %	116	18	0	2
57.8 %	118	18	0	2
57.8 %	121	19	0	2
57.4 %	124	19	0	2
54 %	114	12	0	0

Table 4.23.: Comparison with old MACKE on flex SIR orig v0

Coverage	Vuln.Inst.	$\|chain\| > 1$	$chain \prec P2$	main exploit
30.2 %	95	53	8	5
29.9 %	94	65	7	5
30.1 %	98	74	17	5
30.5 %	94	66	8	5
30.0 %	94	65	8	5
21 %	75	9	0	1

Table 4.24.: Comparison with old MACKE on coreutils 6.10

Coverage	Vuln.Inst.	$\|chain\| > 1$	$chain \prec P2$	main exploit
51.6 %	177	5	0	52
51.0 %	176	5	0	52
51.1 %	175	5	0	52
51.2 %	172	5	0	51
50.9 %	175	5	0	52
51 %	240	27	0	21

Table 4.25.: Comparison with old KLEE on bzip2 1.0.6

Coverage	main exploit
8.9 %	0
9.2 %	0
8.9 %	0
8.9 %	0
9.2 %	0
5 %	0

Table 4.26.: Comparison with old KLEE on grep SIR orig v0

Coverage	main exploit
48.7 %	2
52.8 %	2
53.4 %	2
53.4 %	2
46.8 %	2
44 %	0

Table 4.27.: Comparison with old KLEE on flex SIR orig v0

Coverage	main exploit
15.4 %	3
21.1 %	3
17.2 %	3
20.5 %	3
16.6 %	3
7 %	1

Table 4.28.: Comparison with old KLEE on coreutils 6.10

Coverage	main exploit
50.4 %	28
50.7 %	28
50.2 %	28
50.7 %	28
50.7 %	28
43 %	20

The more complex logic is also beneficial for the error chains. For all programs, except `coreutils`, new MACKE finds more error chains through more than one function. Additionally, new MACKE is able to propagate errors in phase two, which is not possible at all with old MACKE. The number of uncovered vulnerabilities kind of correlates with the achieved coverage: lower coverage with less vulnerable instructions in `bzip2`, higher coverage with more vulnerable instructions in `flex`. Again, the numbers of `coreutils` form an exception. Supposedly, the reason for this outlier is not MACKE, but a different counting strategy: `coreutils` consists of 98 standalone programs sharing parts of their codes and their distinct results are merged for the tables. New MACKE actually merges the elements, so the same vulnerable instruction inside a library used by multiple programs is just counted once, but old MACKE seems to simply add the numbers for vulnerable instructions and chains.

Finally, a remark on the discovered main exploits. The paper about old MACKE reports an error inside touch.c, that was uncovered by old MACKE, but not by KLEE alone. In these test results, there are two main exploits for `touch.c` but they are found by both, new MACKE and KLEE. With no additional information about the exploit in the paper, either the vulnerability is no longer discovered by new MACKE or KLEE is capable of exposing it as well. Nevertheless, new MACKE reports three main exploits for `flex` and 24 errors for `coreutils`, that are not discovered with KLEE. Details on these exploits are discussed in Section 4.8.1. Furthermore, new MACKE as well as KLEE report errors for `grep`, `flex` and `coreutils` that neither have been revealed by old MACKE nor have been reached by the KLEE runs in the paper. But without further information about the evaluation in the paper reasoning about this is not expedient.

All in all, new MACKE requires a slightly longer run-time for some programs than old MACKE to achieve the same degree of coverage in the analyzed code, but on the covered instructions, it is capable of building more and longer error chains. I consider the resulting reports to have a higher quality and thereby the suggested changes to MACKE implemented in the new version as beneficial, although they cost a little extra run-time.

Table 4.29.: Runtime on bc 1.06

	runt.P1	**runt.P2**	**DOP1**	**DOP2**
	16 *min*	31 *min*	2.96	1.64
	16 *min*	29 *min*	2.77	1.68
	15 *min*	31 *min*	2.83	1.64
	15 *min*	31 *min*	2.94	1.64
	15 *min*	31 *min*	2.99	1.63
⌀	15.4	30.6	2.90	1.65
σ	0.5	0.9	0.09	0.02

Table 4.30.: Runtime on bison 3.0.4

	runt.P1	**runt.P2**	**DOP1**	**DOP2**
	275 *min*	430 *min*	2.59	1.41
	276 *min*	429 *min*	2.54	1.45
	276 *min*	432 *min*	2.54	1.41
	276 *min*	424 *min*	2.52	1.42
	277 *min*	420 *min*	2.50	1.46
⌀	276.0	427.0	2.54	1.43
σ	0.7	4.9	0.03	0.02

Table 4.31.: Runtime on bzip2 1.0.6

	runt.P1	**runt.P2**	**DOP1**	**DOP2**
	19 *min*	25 *min*	2.57	1.38
	19 *min*	29 *min*	2.58	1.25
	18 *min*	27 *min*	2.94	1.36
	20 *min*	29 *min*	2.69	1.26
	19 *min*	27 *min*	2.70	1.29
⌀	19.0	27.4	2.69	1.31
σ	0.7	1.7	0.15	0.06

4.6. Runtime and degree of parallelization

A big benefit of the compositional structure in MACKE is that many of the distinct components are independent of each other and hence can be analyzed in parallel. This section takes a deeper look at the influence of the availability of multiple cores on the overall run-time, that is achieved in both phases during the analysis of the 16 evaluated programs. As discussed in Section 3.8, the parallelism of phase one is more flawless than in phase two. The following discussion substantiates this assumption with some measurements and numbers.

The run-time of phase one and phase two as well as the degree of parallelism (DOP) for each program is depicted in Table 4.29 to Table 4.47.

Considering the numbers inside the tables, it is apparent that the run-time behavior depends hugely on the analyzed program. The smaller programs like `bzip2`, `bc` and `zopfli` require around an hour for the complete analysis, whereas bigger programs like `bison`, `tar` and `goahead` twelve to 16 hours. The amount of time spent in phase two depends on the number of exposed vulnerable instructions in phase one. In the extreme case, phase one does not reveal any vulnerable instructions – like for `jq` – and phase two has nothing to do. In the other extreme case, phase one results in several hundred error chains – like for `goahead` and `ngircd` – and phase two takes around three times longer than phase one. Most notably, phase two of new MACKE requires a lot more time than in old MACKE. The old implementation requires less than five minutes [46, chapter 3], whereas phase two in the new implementation frequently requires more time than phase one. This is probably the consequence of the longer and more numerous error chains detected by the new matching system (see Section 3.4).

Now on the degree of parallelism: With the four cores available on the test machine, perfect parallelism has a DOP of 4.0, a full workload single-threaded execution of KLEE has a DOP of 1. Please note, that both numbers are the theoretical optimum. Furthermore, only the time spent inside KLEE is used as a numerator for this fraction, assuming that all bitcode operations and synchronization overhead are negligible. This assumption does not apply, when the majority of KLEE runs completes fast, while the corresponding bitcode operations require a constant chunk of time. This happens for phase two of `flex SIR`, resulting in a DOP of only 0.58.

As expected, for a single program the DOP for phase one is better than for phase two in all analyzed programs. Nevertheless, the DOP depends on the analyzed program. It varies from 2.64 (`tar`) to 3.56 (`zopfli`) for phase one and from 0.58 (`flex SIR`) to 2.54 (`ngircd`). These ranges do not include both versions of `coreutils` since it is only a collection of around hundred small programs. Although in total they form the biggest analyzed code base, each separate MACKE run analyzes a comparably small program.

Table 4.32.: Runtime on coreutils 6.10

	runt.P1	runt.P2	DOP1	DOP2
	304 *min*	150 *min*	1.34	0.89
	302 *min*	150 *min*	1.33	0.89
	301 *min*	150 *min*	1.35	0.89
	301 *min*	150 *min*	1.32	0.89
	299 *min*	150 *min*	1.37	0.89
⌀	301.4	150.0	1.34	0.89
σ	1.8	0.0	0.02	0.00

Table 4.33.: Runtime on coreutils 8.25

	runt.P1	runt.P2	DOP1	DOP2
	328 *min*	201 *min*	1.19	0.92
	329 *min*	201 *min*	1.15	0.91
	326 *min*	200 *min*	1.20	0.92
	325 *min*	201 *min*	1.19	0.92
	325 *min*	201 *min*	1.19	0.92
⌀	326.6	200.8	1.18	0.92
σ	1.8	0.4	0.02	0.00

Table 4.34.: Runtime on diff 3.4

	runt.P1	runt.P2	DOP1	DOP2
	83 *min*	121 *min*	3.17	1.23
	84 *min*	125 *min*	3.19	1.22
	83 *min*	126 *min*	3.18	1.23
	84 *min*	123 *min*	3.22	1.22
	83 *min*	122 *min*	3.23	1.24
⌀	83.4	123.4	3.20	1.23
σ	0.5	2.1	0.03	0.01

Table 4.35.: Runtime on flex 2.6.0

	runt.P1	runt.P2	DOP1	DOP2
	44 *min*	77 *min*	2.96	0.92
	43 *min*	77 *min*	2.98	1.01
	43 *min*	76 *min*	2.88	1.02
	43 *min*	76 *min*	3.02	0.93
	43 *min*	76 *min*	3.00	1.00
⌀	43.2	76.4	2.97	0.98
σ	0.4	0.5	0.06	0.05

Table 4.36.: Runtime on flex SIR orig.v0

	runt.P1	runt.P2	DOP1	DOP2
	22 *min*	82 *min*	3.48	0.60
	22 *min*	82 *min*	3.39	0.56
	22 *min*	83 *min*	3.50	0.60
	23 *min*	82 *min*	3.27	0.59
	23 *min*	81 *min*	3.35	0.58
⌀	22.4	82.0	3.40	0.58
σ	0.5	0.7	0.10	0.02

Table 4.37.: Runtime on goahead 3.6.3

	runt.P1	runt.P2	DOP1	DOP2
	281 *min*	731 *min*	2.68	1.75
	280 *min*	728 *min*	2.69	1.72
	281 *min*	729 *min*	2.65	1.69
	282 *min*	726 *min*	2.60	1.70
	280 *min*	726 *min*	2.57	1.73
⌀	280.8	728.0	2.64	1.72
σ	0.8	2.1	0.05	0.02

Table 4.38.: Runtime on grep 2.25

	runt.P1	runt.P2	DOP1	DOP2
	91 *min*	143 *min*	3.08	1.48
	91 *min*	147 *min*	2.92	1.48
	91 *min*	141 *min*	3.03	1.55
	92 *min*	144 *min*	2.92	1.39
	92 *min*	143 *min*	2.86	1.50
⌀	91.4	143.6	2.96	1.48
σ	0.5	2.2	0.09	0.06

Table 4.39.: Runtime on grep SIR orig.v0

	runt.P1	runt.P2	DOP1	DOP2
	22 *min*	38 *min*	3.02	1.05
	22 *min*	38 *min*	3.38	1.14
	22 *min*	37 *min*	3.10	1.34
	22 *min*	38 *min*	3.18	1.13
	22 *min*	38 *min*	3.35	1.12
⌀	22.0	37.8	3.21	1.16
σ	0.0	0.4	0.16	0.11

Table 4.40.: Runtime on jq 1.5

	runt.P1	runt.P2	DOP1	DOP2
	15 *min*	0 *min*	0.14	0.00
	14 *min*	0 *min*	0.14	0.00
	14 *min*	0 *min*	0.14	0.00
	14 *min*	0 *min*	0.14	0.00
	14 *min*	0 *min*	0.14	0.00
⌀	14.2	0.0	0.14	0.00
σ	0.4	0.0	0.00	0.00

Table 4.41.: Runtime on less 481

	runt.P1	runt.P2	DOP1	DOP2
	51 *min*	55 *min*	2.97	1.22
	51 *min*	52 *min*	2.96	1.34
	50 *min*	54 *min*	3.04	1.21
	50 *min*	52 *min*	2.84	1.29
	51 *min*	54 *min*	2.83	1.23
⌀	50.6	53.4	2.93	1.26
σ	0.5	1.3	0.09	0.05

Table 4.42.: Runtime on libuv 1.9.1

	runt.P1	runt.P2	DOP1	DOP2
	98 *min*	84 *min*	2.89	1.21
	99 *min*	83 *min*	2.99	1.23
	99 *min*	85 *min*	2.88	1.23
	98 *min*	87 *min*	2.90	1.27
	98 *min*	84 *min*	2.99	1.25
⌀	98.4	84.6	2.93	1.24
σ	0.5	1.5	0.05	0.02

Table 4.43.: Runtime on lz4 r131

	runt.P1	runt.P2	DOP1	DOP2
	36 *min*	60 *min*	3.44	1.38
	37 *min*	61 *min*	3.35	1.48
	36 *min*	59 *min*	3.42	1.55
	36 *min*	61 *min*	3.37	1.60
	36 *min*	64 *min*	3.40	1.47
⌀	36.2	61.0	3.40	1.50
σ	0.4	1.9	0.04	0.09

Table 4.44.: Runtime on ngircd 23

	runt.P1	runt.P2	DOP1	DOP2
	132 *min*	362 *min*	3.27	2.41
	132 *min*	361 *min*	3.33	2.40
	133 *min*	359 *min*	3.21	2.44
	133 *min*	362 *min*	3.31	2.41
	134 *min*	364 *min*	3.21	2.38
⌀	132.8	361.6	3.27	2.41
σ	0.8	1.8	0.05	0.02

Table 4.45.: Runtime on sed 4.2.2

	runt.P1	runt.P2	DOP1	DOP2
	43 *min*	70 *min*	3.15	1.43
	42 *min*	74 *min*	3.34	1.41
	43 *min*	73 *min*	3.22	1.47
	43 *min*	71 *min*	3.29	1.49
	43 *min*	70 *min*	3.34	1.39
⌀	42.8	71.6	3.27	1.44
σ	0.4	1.8	0.08	0.04

Table 4.46.: Runtime on tar 1.29

	runt.P1	runt.P2	DOP1	DOP2
	336 *min*	595 *min*	2.77	1.16
	336 *min*	581 *min*	2.82	1.21
	338 *min*	589 *min*	2.79	1.18
	336 *min*	591 *min*	2.79	1.18
	338 *min*	590 *min*	2.76	1.17
⌀	336.8	589.2	2.79	1.18
σ	1.1	5.1	0.03	0.02

Table 4.47.: Runtime on zopfli 1.0.1

	runt.P1	runt.P2	DOP1	DOP2
	25 *min*	45 *min*	3.47	1.55
	24 *min*	48 *min*	3.59	1.61
	24 *min*	48 *min*	3.48	1.50
	24 *min*	47 *min*	3.58	1.47
	24 *min*	45 *min*	3.69	1.52
⌀	24.2	46.6	3.56	1.53
σ	0.4	1.5	0.09	0.05

With the small collection of programs in the evaluation, the correlation between DOP and program size seems to be the following: Bigger programs require more costly bitcode operations, which decreases the DOP in both phases. Nevertheless, bigger programs have a bigger call graph and therefore the scheduler for phase two forms bigger independent groups with more parallelism, which increases the DOP in phase two; in particular, since the bitcode operations for phase one in the current implementation are single-threaded but are parallelized in phase two. So, roughly speaking, DOP1 is indirectly proportional and DOP2 is directly proportional to the size of the analyzed program.

4.7. Hypothesis: MACKE covers more and deeper parts of the program than pure KLEE

The decomposition of the analyzed program performed by MACKE has a main purpose: It aims to increase the overall coverage in the program achievable in the same time and potentially uncover more vulnerabilities. This section discusses the hypothesis that MACKE actually covers more parts of the program and especially deeper parts of the program compared to the symbolic execution of the whole program with KLEE.

A first impression of the achieved coverage is given by the general analysis in Section 4.4. The tables there depict side by side the achieved overall coverage by MACKE and KLEE on the same program. For 11 out of 18 programs, MACKE obtains more than twice the coverage than KLEE – namely, on `bison`, `bzip2`, `diff`, `flex`, `goahead`, `less`, `lz4`, `ngircd`, `sed`, `tar` and `zopfli`. For another 4 programs, – namely, `bc`, `flex SIR`, `grep` and `grep SIR` – the coverage with MACKE is considerably higher. For two programs – namely, `coreutils 6` and `coreutils 8`, the coverage by MACKE and by KLEE are comparable. MACKE is slightly better on `coreutils 6` and slightly worse on `coreutils 8` than KLEE. For only 1 program, `jq`, MACKE's coverage is worse than the coverage by KLEE.

4.7.1. Coverage on function level

So, only according to the overall coverage, the results looks quite promising: For 15 out of 18 programs, the coverage is improved, for 3 programs the additional effort was not beneficial. But for a more precise judgment, the overall coverage is a too general measure. Hence, Table 4.48 to Table 4.66 show the coverage on function level for each of the programs.

The function coverage underlines the first impression. For all programs, except `jq`, MACKE covers more than 60% of all functions, whereas KLEE is not able to cover more than 60% in each program. Even on both versions of `coreutils`, where the overall coverage was similar, MACKE is able to cover more functions than KLEE. Only `jq` seems to be problematic for MACKE, but the function coverage reveals the reason.

During phase one of MACKE, each function inside the program is encapsulated symbolically and analyzed in isolation. Assuming, that each function can be encapsulated, 100% of all functions should by covered by the analyses of MACKE and none should

be completely uncovered or optimized away. However, this is not the case for any program, because not all functions can be encapsulated. As discussed in Section 3.3.3, multi-pointer arguments can not be handled by MACKE. Therefore, all functions with such an argument can only be covered by phase two or remain uncovered. For `jq`, most of the functions use a pointer to on an internal state struct, which cannot be handled by the symbolic encapsulation. KLEE's analysis starts from main and initializes these structs there with concrete variables. So these functions can be analyzed flawlessly.

4.7.2. Coverage on LOC level

Nevertheless, the previous analyses are not fine grained enough to explain the outcome on coreutils. In order to fill this gap, I use a modified type of histogram for the source code of the analyzed programs that depicts the achieved coverage of MACKE and KLEE side by side and gives further inside about the distribution of the covered lines.

Each diagram consists of two mirrored histograms, one for KLEE on the left side and for MACKE on the right side. The y-axis denotes the depth inside the program. Each LOC corresponds to one function and this function has a minimal distance from the main function in the call graph of the program. For example, a depth of three is assigned to all LOC inside a function that is three calls away from the main function. A depth of zero directly belongs to code inside the main function. The x-axis denotes the number of LOC that resides the corresponding number of calls away from the main function. The resulting area is colored in green if the corresponding LOC was covered, red if it was not covered and white if the LOC was optimized away by the compiler. If MACKE and KLEE achieve the same coverage on the program, both histograms are symmetric. Any aberration of this denotes differences in the achieved coverage.

Before starting with the discussion, please note two special details: Firstly, there is a separate bar above the y-axis denoting a depth of `?`. It represents LOC where it is impossible to calculate the distance to the main function, e.g. when a function is included in the source code, but never used by the program – then its code is completely optimized away – or the function is only called via function pointers and thus does not have direct links inside the call graph.

Secondly, each program has only one picture, although its analysis is repeated five times. Before drawing these pictures, I took a majority voting on all five individual results: If more than two runs cover a line, it is considered as covered; alike for uncovered and removed lines. The five individual pictures would look almost identical, and thus I have applied this small trick to save some space. Only for programs – namely, `lz4` and `sed` – where some KLEE runs crashes, these crashes are excluded from the voting on KLEE. Otherwise, the KLEE side of these graphs would be completely empty, which does not add any information for the discussion.

Table 4.48.: Function coverage for bc 1.06 with MACKE (left) and KLEE (right)

Covered	Uncovered	Ratio	Removed	Covered	Uncovered	Ratio	Removed
87	39	67%	3	33	83	25%	13
86	39	66%	4	30	86	23%	13
86	39	66%	4	31	85	24%	13
87	39	67%	3	30	86	23%	13
87	39	67%	3	34	82	26%	13
⌀ 86.6	39.0	67.1	3.4	⌀ 31.6	84.4	24.5	13.0
σ 0.5	0.0	0.4	0.5	σ 1.8	1.8	1.4	0.0

Table 4.49.: Function coverage for bison 3.0.4 with MACKE (left) and KLEE (right)

Covered	Uncovered	Ratio	Removed	Covered	Uncovered	Ratio	Removed
713	231	66%	132	106	735	9%	235
714	229	66%	133	96	745	8%	235
705	239	65%	132	106	735	9%	235
703	240	65%	133	142	699	13%	235
714	228	66%	134	106	735	9%	235
⌀ 709.8	233.4	66.0	132.8	⌀ 111.2	729.8	10.3	235.0
σ 5.4	5.7	0.5	0.8	σ 17.8	17.8	1.6	0.0

Table 4.50.: Function coverage for bzip2 1.0.6 with MACKE (left) and KLEE (right)

Covered	Uncovered	Ratio	Removed	Covered	Uncovered	Ratio	Removed
85	17	78%	6	20	72	18%	16
85	17	78%	6	20	72	18%	16
85	17	78%	6	20	72	18%	16
85	17	78%	6	20	72	18%	16
85	17	78%	6	20	72	18%	16
⌀ 85.0	17.0	78.7	6.0	⌀ 20.0	72.0	18.5	16.0
σ 0.0	0.0	0.0	0.0	σ 0.0	0.0	0.0	0.0

Table 4.51.: Function coverage for coreutils 6.10 with MACKE (left) and KLEE (right)

Covered	Uncovered	Ratio	Removed	Covered	Uncovered	Ratio	Removed
908	408	64%	84	796	468	56%	136
904	412	64%	84	794	470	56%	136
903	413	64%	84	791	473	56%	136
908	408	64%	84	795	469	56%	136
904	412	64%	84	794	470	56%	136
⌀ 905.4	410.6	64.7	84.0	⌀ 794.0	470.0	56.7	136.0
σ 2.4	2.4	0.2	0.0	σ 1.9	1.9	0.1	0.0

Table 4.52.: Function coverage for coreutils 8.25 with MACKE (left) and KLEE (right)

Covered	Uncovered	Ratio	Removed	Covered	Uncovered	Ratio	Removed
1037	508	61%	138	890	546	52%	247
1041	504	61%	138	878	558	52%	247
1044	501	62%	138	893	543	53%	247
1046	499	62%	138	873	529	51%	281
1041	504	61%	138	881	555	52%	247
⌀ 1041.8	503.2	61.9	138.0	⌀ 883.0	546.2	52.5	253.8
σ 3.4	3.4	0.2	0.0	σ 8.3	11.4	0.5	15.2

Table 4.53.: Function coverage for diff 3.4 with MACKE (left) and KLEE (right)

	Covered	Uncovered	Ratio	Removed		Covered	Uncovered	Ratio	Removed
	278	24	71%	89		52	135	13%	204
	276	25	70%	90		53	134	13%	204
	278	23	71%	90		48	139	12%	204
	278	23	71%	90		48	139	12%	204
	278	24	71%	89		49	138	12%	204
⌀	277.6	23.8	71.0	89.6	⌀	50.0	137.0	12.8	204.0
σ	0.9	0.8	0.2	0.5	σ	2.3	2.3	0.6	0.0

Table 4.54.: Function coverage for flex 2.6.0 with MACKE (left) and KLEE (right)

	Covered	Uncovered	Ratio	Removed		Covered	Uncovered	Ratio	Removed
	188	19	72%	53		41	176	15%	43
	192	18	73%	50		40	177	15%	43
	190	19	73%	51		40	177	15%	43
	190	19	73%	51		40	177	15%	43
	190	20	73%	50		40	177	15%	43
⌀	190.0	19.0	73.1	51.0	⌀	40.2	176.8	15.5	43.0
σ	1.4	0.7	0.5	1.2	σ	0.4	0.4	0.2	0.0

Table 4.55.: Function coverage for flex SIR orig.v0 with MACKE (left) and KLEE (right)

	Covered	Uncovered	Ratio	Removed		Covered	Uncovered	Ratio	Removed
	122	20	84%	2		38	100	26%	6
	120	22	83%	2		50	88	34%	6
	121	21	84%	2		40	98	27%	6
	122	20	84%	2		49	89	34%	6
	122	20	84%	2		39	99	27%	6
⌀	121.4	20.6	84.3	2.0	⌀	43.2	94.8	30.0	6.0
σ	0.9	0.9	0.6	0.0	σ	5.8	5.8	4.0	0.0

Table 4.56.: Function coverage for goahead 3.6.3 with MACKE (left) and KLEE (right)

	Covered	Uncovered	Ratio	Removed		Covered	Uncovered	Ratio	Removed
	718	165	74%	75		0	0	0%	958
	721	161	75%	76		0	0	0%	958
	717	163	74%	78		0	0	0%	958
	722	161	75%	75		0	0	0%	958
	716	162	74%	80		0	0	0%	958
⌀	718.8	162.4	75.0	76.8	⌀	0.0	0.0	0.0	958.0
σ	2.6	1.7	0.3	2.2	σ	0.0	0.0	0.0	0.0

Table 4.57.: Function coverage for grep 2.25 with MACKE (left) and KLEE (right)

	Covered	Uncovered	Ratio	Removed		Covered	Uncovered	Ratio	Removed
	339	52	73%	70		147	164	31%	150
	337	53	73%	71		145	166	31%	150
	338	52	73%	71		146	165	31%	150
	338	52	73%	71		161	150	34%	150
	336	55	72%	70		132	179	28%	150
⌀	337.6	52.8	73.2	70.6	⌀	146.2	164.8	31.7	150.0
σ	1.1	1.3	0.2	0.5	σ	10.3	10.3	2.2	0.0

Table 4.58.: Function coverage for grep SIR orig.v0 with MACKE (left) and KLEE (right)

Covered	Uncovered	Ratio	Removed	Covered	Uncovered	Ratio	Removed
108	9	90%	2	69	41	57%	9
109	9	91%	1	75	35	63%	9
106	11	89%	2	75	35	63%	9
106	11	89%	2	76	34	63%	9
108	10	90%	1	67	43	56%	9
⌀ 107.4	10.0	90.3	1.6	⌀ 72.4	37.6	60.8	9.0
σ 1.3	1.0	1.1	0.5	σ 4.1	4.1	3.4	0.0

Table 4.59.: Function coverage for jq 1.5 with MACKE (left) and KLEE (right)

Covered	Uncovered	Ratio	Removed	Covered	Uncovered	Ratio	Removed
118	412	20%	42	182	347	31%	43
118	412	20%	42	181	348	31%	43
118	412	20%	42	182	347	31%	43
118	412	20%	42	181	348	31%	43
118	412	20%	42	182	347	31%	43
⌀ 118.0	412.0	20.6	42.0	⌀ 181.6	347.4	31.7	43.0
σ 0.0	0.0	0.0	0.0	σ 0.5	0.5	0.1	0.0

Table 4.60.: Function coverage for less 481 with MACKE (left) and KLEE (right)

Covered	Uncovered	Ratio	Removed	Covered	Uncovered	Ratio	Removed
382	65	83%	12	13	417	2%	29
381	66	83%	12	13	417	2%	29
381	66	83%	12	13	417	2%	29
381	66	83%	12	13	417	2%	29
380	67	82%	12	13	417	2%	29
⌀ 381.0	66.0	83.0	12.0	⌀ 13.0	417.0	2.8	29.0
σ 0.7	0.7	0.2	0.0	σ 0.0	0.0	0.0	0.0

Table 4.61.: Function coverage for lz4 r131 with MACKE (left) and KLEE (right)

Covered	Uncovered	Ratio	Removed	Covered	Uncovered	Ratio	Removed
145	16	70%	44	0	0	0%	205
144	16	70%	45	0	0	0%	205
144	16	70%	45	0	0	0%	205
143	17	69%	45	0	0	0%	205
144	16	70%	45	47	80	22%	78
⌀ 144.0	16.2	70.2	44.8	⌀ 9.4	16.0	4.6	179.6
σ 0.7	0.4	0.3	0.4	σ 21.0	35.8	10.3	56.8

Table 4.62.: Function coverage for ngircd 23 with MACKE (left) and KLEE (right)

Covered	Uncovered	Ratio	Removed	Covered	Uncovered	Ratio	Removed
432	88	75%	54	0	0	0%	574
438	83	76%	53	0	0	0%	574
435	84	75%	55	0	0	0%	574
437	84	76%	53	0	0	0%	574
437	84	76%	53	0	0	0%	574
⌀ 435.8	84.6	75.9	53.6	⌀ 0.0	0.0	0.0	574.0
σ 2.4	1.9	0.4	0.9	σ 0.0	0.0	0.0	0.0

Table 4.63.: Function coverage for sed 4.2.2 with MACKE (left) and KLEE (right)

	Covered	Uncovered	Ratio	Removed		Covered	Uncovered	Ratio	Removed
	156	8	73%	49		0	0	0%	213
	157	7	73%	49		0	0	0%	213
	158	6	74%	49		0	0	0%	213
	157	7	73%	49		100	17	46%	96
	158	6	74%	49		101	16	47%	96
⌀	157.2	6.8	73.8	49.0	⌀	40.2	6.6	18.9	166.2
σ	0.8	0.8	0.4	0.0	σ	55.0	9.0	25.8	64.1

Table 4.64.: Function coverage for tar 1.29 with MACKE (left) and KLEE (right)

	Covered	Uncovered	Ratio	Removed		Covered	Uncovered	Ratio	Removed
	1022	115	84%	77		5	992	0%	217
	1025	112	84%	77		5	992	0%	217
	1028	110	84%	76		5	992	0%	217
	1026	113	84%	75		5	992	0%	217
	1019	119	83%	76		5	992	0%	217
⌀	1024.0	113.8	84.3	76.2	⌀	5.0	992.0	0.4	217.0
σ	3.5	3.4	0.3	0.8	σ	0.0	0.0	0.0	0.0

Table 4.65.: Function coverage for zopfli 1.0.1 with MACKE (left) and KLEE (right)

	Covered	Uncovered	Ratio	Removed		Covered	Uncovered	Ratio	Removed
	83	16	80%	4		16	83	15%	4
	84	15	81%	4		16	83	15%	4
	84	15	81%	4		16	83	15%	4
	84	15	81%	4		16	83	15%	4
	85	14	82%	4		16	83	15%	4
⌀	84.0	15.0	81.6	4.0	⌀	16.0	83.0	15.5	4.0
σ	0.7	0.7	0.7	0.0	σ	0.0	0.0	0.0	0.0

Table 4.66.: Function coverage for libuv 1.9.1 with MACKE

	Covered	Uncovered	Ratio	Removed
	338	22	73%	102
	340	24	73%	98
	335	25	72%	102
	337	22	72%	103
	337	24	72%	101
⌀	337.4	23.4	73.0	101.2
σ	1.8	1.3	0.4	1.9

The coverages for all programs are depicted in Figure 4.1 to Figure 4.18. Unfortunately, the histograms for `goahead`, `grep SIR` and `less` are almost completely empty. This is caused by function pointers that are introduced into the main function by the compiler. They break the call graph, so the complete program is assigned to depth `?`. For `ngircd`, the KLEE runs are all stopped by crashes, so the histogram shows all LOC as removed as a fallback.

The histograms confirm the findings from the previous tables. In general, except for `jq`, MACKE's side is much greener than KLEE's side. But they also reveal a correlation between the higher coverage of MACKE and the structure of the program's call graph. For programs where the histogram forms a pyramide, i.e. where most LOC have a very small distance to main, MACKE only has a small advantage over KLEE. This is especially the case for both `coreutils`, which have multiple main functions. For programs where the histogram forms a bulb, MACKE's coverage is superior. The best example for this is `zopfli`: KLEE left all deep parts of the program uncovered, whereas MACKE covers them adequately.

Considering all these data, I think the hypothesis, that MACKE covers more and deeper parts of the program, can be accepted. MACKE has some problems with pointers and, of course, the results vary depending on the analyzed program, but in general, MACKE achieves better coverage than pure KLEE.

Figure 4.1.: Coverage graph bc 1.06

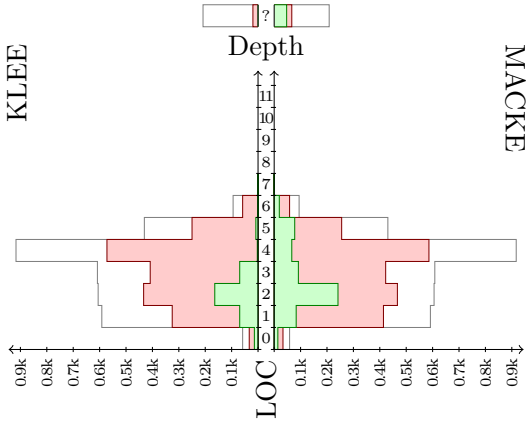


Figure 4.2.: Coverage graph bison 3.0.4

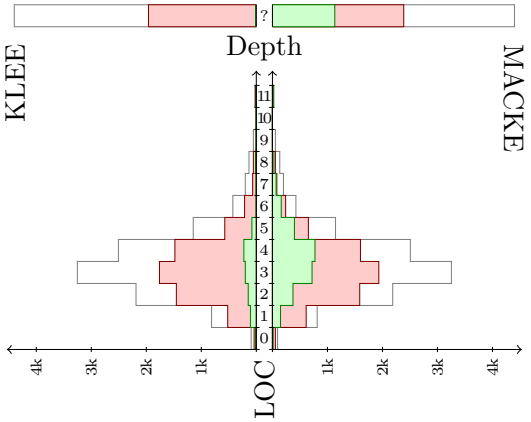


Figure 4.3.: Coverage graph bzip2 1.0.6

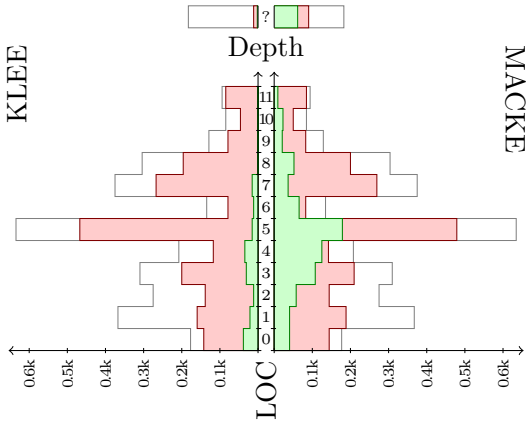


Figure 4.4.: Coverage graph coreutils 6.10

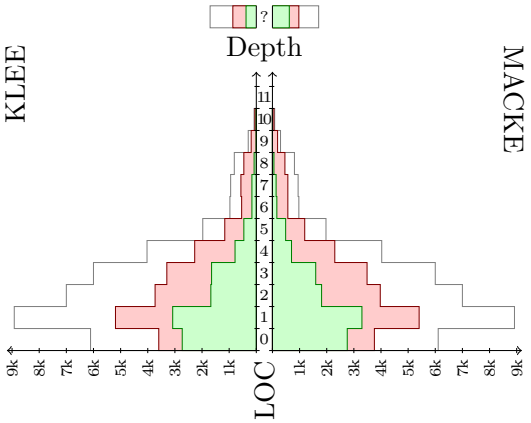


Figure 4.5.: Coverage graph coreutils 8.25

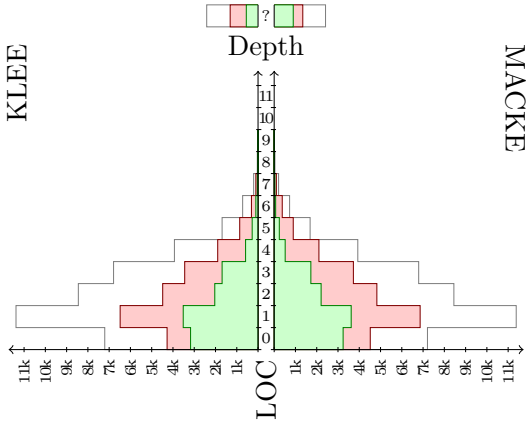


Figure 4.6.: Coverage graph diff 3.4

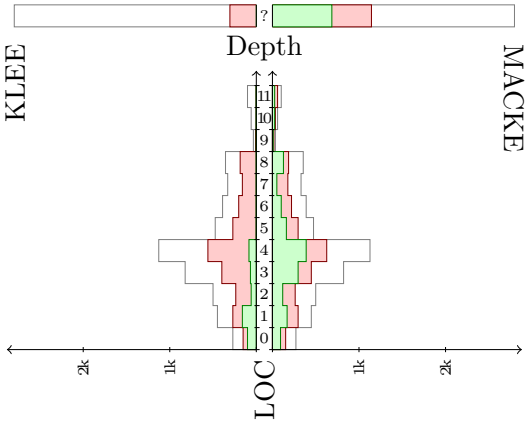


Figure 4.7.: Coverage graph flex 2.6.0

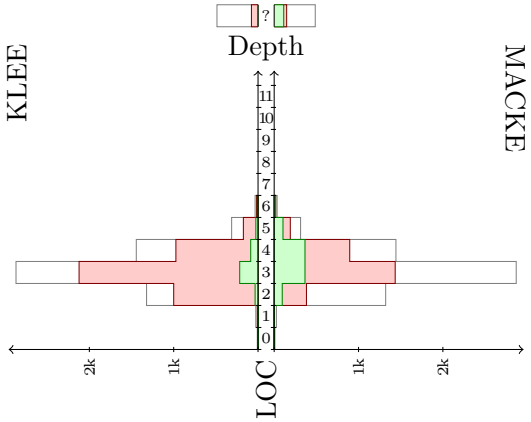


Figure 4.8.: Coverage graph flex SIR

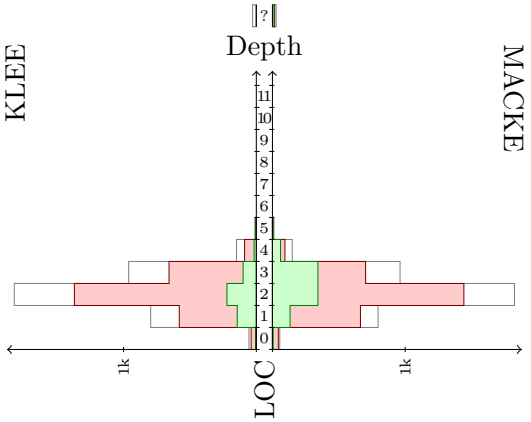


Figure 4.9.: Coverage graph goahead 3.6.3

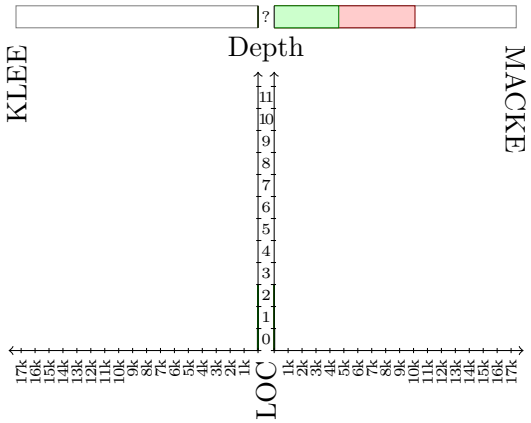


Figure 4.10.: Coverage graph grep 2.25

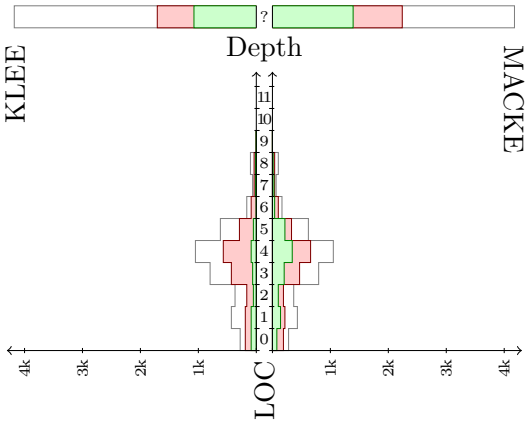


Figure 4.11.: Coverage graph grep SIR

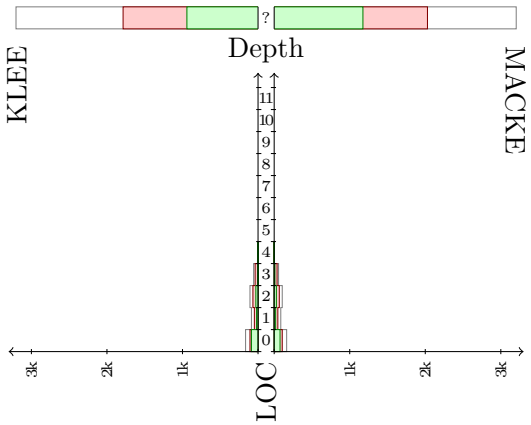


Figure 4.12.: Coverage graph jq 1.5

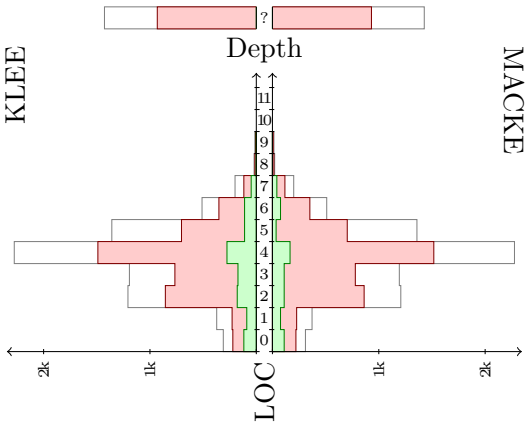


Figure 4.13.: Coverage graph less 481

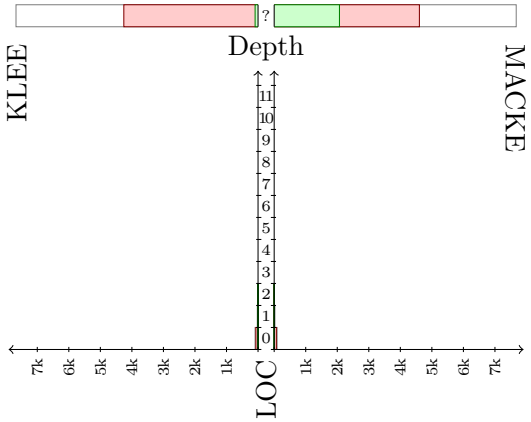


Figure 4.14.: Coverage graph lz4 r131

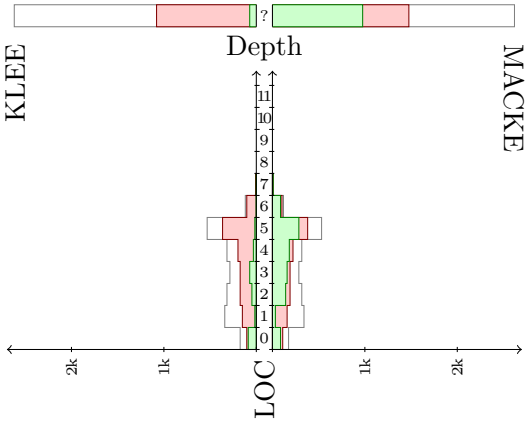


Figure 4.15.: Coverage graph ngircd 23

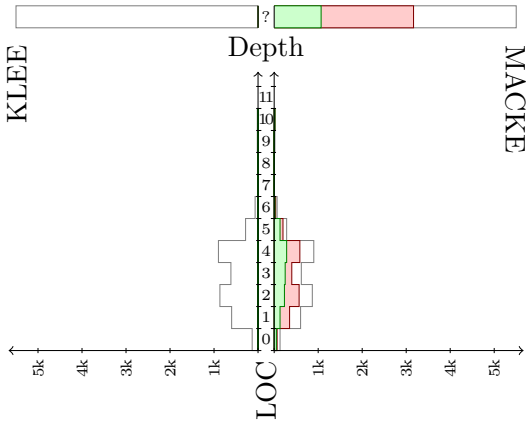


Figure 4.16.: Coverage graph sed 4.2.2

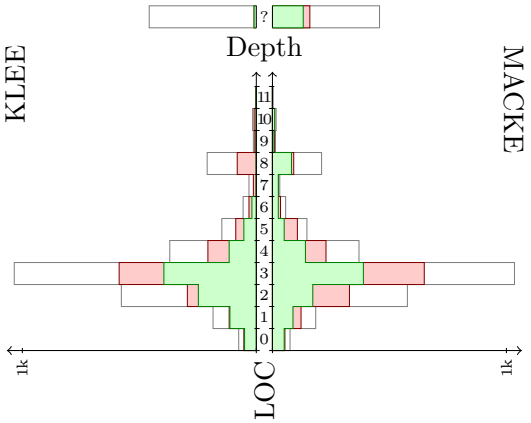


Figure 4.17.: Coverage graph tar 1.29

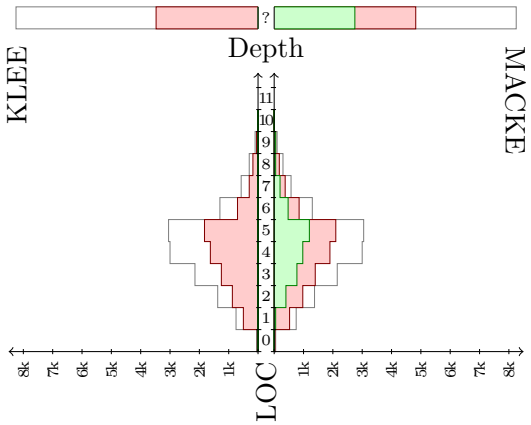
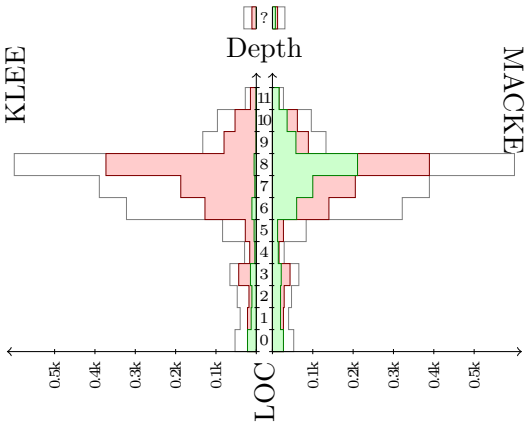


Figure 4.18.: Coverage graph zopfli 1.0.1



4.8. Hypothesis: MACKE finds more errors than KLEE

Even more important than the achieved coverage of the analyses are the revealed vulnerabilities. Especially main exploits, i.e. vulnerable instructions that have been shown reachable from the main function are an extreme valuable feedback for the developers. In order to be revealed, a vulnerable instruction must be covered during the analysis. With the higher coverage showed in Section 4.7 MACKE should be able to find more errors than KLEE. Therefore, this section focuses on the reported errors of both tools and discusses the hypothesis that MACKE finds more errors than KLEE within the same time frame.

All library errors reported by MACKE, i.e. vulnerable instructions where MACKE is not able to find an input for the main function, have no corresponding element in KLEE's error report. With KLEE starting the analysis from the main function each revealed vulnerable instruction is automatically a main exploit. So all these errors are a clear advantage for MACKE. However, a direct comparison on the main exploits is more instructive.

Not all analyzed programs contain main exploits, but at least all main exploits revealed by KLEE should also be revealed by MACKE. This assumption is investigated in Table 4.67. The table revisits all main exploits reported during the evaluation and classifies them into three groups: Errors that are only reported by MACKE but not by KLEE, errors that are reported by both tools, and finally errors that are only reported by KLEE but not by MACKE. Please note that the vulnerabilities counted here are the same for all five runs of all programs with KLEE and MACKE – except for both versions of `coreutils`. These two exceptions are caused by a few main exploits that are not reported by all MACKE runs. The underlying vulnerable instructions are not reported by KLEE. Hence, they make no difference for the discussed hypothesis and for simplicity, I selected the runs with the most reported main exploits for the row about both `coreutils`. Thereby, none of the five repetitions must be depicted explicitly and the resulting table becomes smaller.

The data in the table clearly shows a superiority for MACKE: it is able to reveal all 41 main exploits too that are reported by KLEE, i.e. no main exploit is uniquely discovered by KLEE. On the other hand, MACKE is able to reveal 67 additional main exploits that are not covered by KLEE. With more than the doubled amount of reported main exploits through compositional analysis and without missing any error that is revealed by whole program analysis, this table strongly supports the hypothesis.

Nevertheless, please note that these differences in the covered errors are only caused by the limited run-time and not by the overall inability of both tools to discover the errors. In theory and with an infinite execution time, both tools would cover all paths in the programs and would reveal exactly the same errors. So MACKE's advantage over KLEE is only caused by spending the limited time on more important parts of the program and not by an improved error detection.

Table 4.67.: Amount of main exploits found by MACKE and/or KLEE

Program	MACKE only	both	KLEE only
bc 1.06	5	0	0
bison 3.0.4	1	0	0
bzip2 1.0.6	0	0	0
coreutils 6.10	24	28	0
coreutils 8.25	34	4	0
diff 3.4	0	1	0
flex 2.6.0	0	0	0
flex SIR	2	3	0
goahead 3.6.3	0	0	0
grep 2.25	0	0	0
grep SIR	0	2	0
jq 1.5	0	0	0
less 481	0	1	0
lz4 r131	0	1	0
ngircd 23	1	0	0
sed 4.2.2	0	0	0
tar 1.29	0	1	0
zopfli 1.0.1	0	0	0
in total	67	41	0

4.8.1. A closer look at the vulnerable instructions

Looking at the plain numbers gives a good overview, but when talking about exploits and vulnerable instructions it is also important to look at the reasons for the reported errors inside the program. MACKE reports more than 3750 vulnerable instructions and 108 unique main exploits in the 19 evaluated programs. Unfortunately, these reported errors are too numerous for a deeper analysis by one person and within the scope of this thesis. Furthermore, I am no domain expert in any of the analyzed tools, so I decided to focus on all the 19 main exploits outside the two `coreutils` bundles in order to give at least a small impression about the underlying vulnerabilities.

Overall, almost all main exploits are caused by two different reasons: wrong assumptions about the size of `argv` and access to uninitialized global variables in low-level functions that are falsely propagated by MACKE.

The first reason is a small detail about the `argv` variable that is used by the `main` function of each C program to receive the command line arguments given to the program. Normally, the first entry in this array denotes the name of the program and all following entries contain an argument of the program. The number of entries in this array is given in the variable `argc`. However, the C standard does not state these strict requirements to the variable. In [32, section 5.1.2.2.1] it only says "The value of `argc` shall be nonnegative" and "`argv[argc]` shall be a null pointer". Hence, a null pointer is a valid assignment to `argv` as well as empty, just null-terminated strings for all parameters and even the program name. Therefore, an imprudent access to these variables may result in an out-of-bound pointer access. KLEE and MACKE stick closely to the standard and reveal all types of these inappropriate usages.

The latter reason only affects MACKE. When it detects an error in a lower level function, it generates a summary of this error and tries to propagate it up to the main function with additional analyses. In the current implementation, this summary only considers the arguments of the function, but not the global variables that can also be associated with the error. Thereby, the used summary might not be an exact representation of all erroneous inputs, but also include some false positives. A more detailed example and some suggestions for possible solutions are discussed in Section 5.2. Anyway, an automatic mechanism to avoid these false positives is not present in the current implementation, so I just manually flag them in the following discussion.

Finally, before starting the deeper investigation of each program, please note, that to the best of my analysis capabilities, I may claim that none of the reported errors is a critical vulnerability. Nevertheless, they are a valuable feedback for the developer, because they can easily be missed by manual code inspection. Even if they only cause program crashes in rare corner cases, they should be fixed inside the program.

`bc` in total contains five different out-of-bound pointer accesses. Two of them lie four calls deep, where the program reads some command line arguments and cannot handle a null-pointer as `argv`. Two false positive errors lie two calls deep in the program, where the low-level function reads from an uninitialized global buffer. Another false positive error lies three calls deep, where the low-level function removes a function from an undefined global function registry. All five errors are uniquely discovered by MACKE.

`bison` contains a false positive out-of-bound pointer access. It is one call deep, where the program access a global hash table that is not initialized during low-level analysis. It is only reported by MACKE.

`diff` contains a violated assert statement. It lies one call deep, where the program initializes the comparison of symlinks and asserts compatible link types. I assume that these symlink types are not supported correctly by KLEE. This is revealed by both tools.

flex SIR in total contains five different out-of-bound pointer accesses. Two of them lie directly inside the main function, where the program fails to read an empty `argv`. Another one lies three calls deep, where the program fails to read from an empty input buffer. Finally, the remaining two errors are false positives, occurring four calls deep, where the low-level function accesses a potentially uninitialized global buffer. Except for the two false positives, all these errors are revealed by both tools.

grep SIR contains one out-of-bound pointer access and one memory error. Both are two calls deep and trace down into a KLEE library providing system functionality. However, there is a comment above this erroneous code that symbolic variables are not supported at this point. So I assume, these are false positive errors that affect both tools.

less contains an out-of-bound pointer access. It lies one call deep, where the program reads information about the currently used terminal. I assume that KLEE does not emulate any existing terminal and the available information has not the expected format. This is revealed by both tools.

lz4 contains an out-of-bound pointer access. It lies directly inside the main function, where the program creates a copy of the program name from the command line arguments that is not expected to be an empty string. This is revealed by both tools.

ngircd contains an out-of-bound pointer access on the program arguments. It lies six calls deep, where a potentially empty value can be written to the logfile. It is uniquely revealed by MACKE.

tar contains an out-of-bound pointer access on the program arguments. It lies one call deep, where the program reads the pointer to the input file that might not be present in the arguments. It is revealed by both tools.

After removing the six false positives from the nine main exploits uniquely revealed by MACKE, there are still three "real" vulnerabilities remaining that are not reported by KLEE. So, after all, the advantage is smaller than initially indicated by the pure numbers, but it is still existing. Especially, with MACKE correctly detecting all errors reported by KLEE, I think the hypothesis that MACKE finds more errors than KLEE can be accepted.

5. Threats to Validity

This chapter discusses several threats to the validity of the findings presented in this thesis. For each threat, the context is shortly explained and afterward, each section gives an expectation about its possible impact.

5.1. Error summary only by example

The biggest threat to validity lies inside the error summaries that are prepended in phase two of MACKE. Although the concept is designed to handle very generic shortcuts for checking, whether the error can be triggered with the current assignment of the (symbolic) variables, the implementation of KLEE left only very little space for composing these summaries.

In the current implementation, MACKE just takes the concrete test case generated by KLEE and adds a "summary", that only checks, if this concrete instance can be realized within the variable space of the current path condition. In other words: Instead of a representation in propositional logic, MACKE only encodes an example, that satisfies the condition for the error.

As an alternative, KLEE also emits the constraints for program path, that leads to the error inside the program. Regrettably, these outputs are a blown up SMT queries on bit vectors, that correlate with the symbolic variables inside the program. Fair enough, KLEE focuses on correctness and not on prettiness, but these queries are not designed for backward matching with the original variables and their types. Furthermore, even if the matching would be possible, there is no way to transform this SAT query back into bitcode that can be used as an identical error summary inside the program. This approach can certainly be implemented, but it requires heavy work on the KLEE code. Therefore, I dismissed this approach and stick with the simpler summary by example.

Without any restriction to the current implementation inside KLEE, there are several other approaches that are discussed in Section 6.5. All of these strategies require big changes inside KLEE and some of them even a complete rewrite of KLEE's test case generation. This lies beyond the scope of this thesis; nevertheless, it has great potentials for further improvements to MACKE.

In addition, MACKE can be used as a test case generator for an arbitrary summary strategy. After phase one, it matches all errors from the same reason and only continues the analysis for errors that are not present in at least one caller. Analyzing the matched errors does not add any benefit because KLEE already shows their reachability, but this situation is ideal for testing the error summary: These errors inside the callee can be triggered from the caller, i.e. the erroneous input is not sanitized, and KLEE is able to find them without any summarization. Therefore, each summary that is better than the original, complete program code, must be able to trigger the error earlier. This generates dozens of different test cases for each analyzed program.

Applying these generated test cases to the currently used summary-by-example strategy emphasizes the requirement of a more suitable error summary: In the experiments, only a few percent of the matched errors can be reconstructed. Therefore, I expect longer error chains and possibly more propagated main vulnerabilities when using an appropriate summary. A more complex summary might also cause a longer run time in phase two, which widens the time frame used for comparison with pure KLEE. Hence, KLEE might overall perform a bit better. Anyhow, the source code coverage with MACKE is achieved in phase one and remains unchanged.

5.2. False positives

Another big problem during the evaluation is the huge portion of false positives, that are reported as errors and pollute the error reports.

For this purpose, KLEE includes `klee-replay`, a tool, that is designed to replay test cases generated by KLEE and to check, if the supposed errors can actually be triggered in the real program. In theory, this tool can be included as an intermediate step into MACKE and filter out all false positives. Unfortunately, this tool seems to be broken and I am not able to verify even handcrafted errors with it.

For the evaluation, I have inspected all main vulnerabilities manually and have tried to figure out, why they are reported as errors. This procedure is error-prone, but the reason for the false positives seems to be the usage of global variables, which are bound outside the currently analyzed function.

For example, consider the small program in Listing 5.1. When MACKE examines `checkmyglobal` separately, there is no input variable, hence no symbolic variable. When the assert statement in line 4 is executed, the assert fails, because `myglobal` has not been set earlier, especially not to 42. Therefore, phase one of MACKE registers this as a vulnerable instruction, when `true` can be satisfied at the function call. On phase two, MACKE revisits the call to `checkmyglobal` in line 9 and prepends this trivial error summary. It can easily be satisfied and MACKE builds an error chain from the assert statement, that can be triggered from `main`. Nevertheless, for the program in total, this assert statement can never be violated, because `myglobal` is set to 42 before the only call in line 8.

This is a very basic example, that can easily be circumvented, but finding a universal solution is beyond the scope of this thesis. Future work can introduce some sort of data flow analysis algorithm [53], which can avoid this source for false positives.

No matter, which way is chosen to remove the false positives from the error reports, it will reduce the number of main exploits and the length of the detected error chains. In the worst case, these numbers highly overestimate the capabilities of MACKE. However, the filtering will increase the quality of the remaining detected errors and I deem it arguable, that an instruction that can cause errors with the “wrong” global variables – despite not happening in the current program – is not considered as some sort of vulnerable instruction.

```
1  static int myglobal;
2
3  void checkmyglobal() {
4    assert(myglobal == 42);
5  }
6
7  int main(int argc, char** argv) {
8    myglobal = 42;
9    checkmyglobal();
10   return 0;
11 }
```

Listing 5.1: Example code reporting causing a false positive error

5.3. Bad support for pointer variables

For the symbolic encapsulation, MACKE needs suitable symbolic equivalents for all variable types used by any function inside the program. As discussed in Section 3.3, this task is easy for flat variables, complicated for single pointer variables and almost impossible for nested-pointer variables.

With the current MACKE implementation ignoring functions with nested-pointer arguments, I expect an increase on the overall coverage, with a good chance to uncover new vulnerable instructions. A more appropriate implementation for single pointers might also have some benefits for the coverage and detected vulnerable instructions. Especially on functions with multiple single pointer variables, this causes exponential state explosions in the current implementation.

The results for pure KLEE depend on the changed run time caused by the new symbolic encapsulation. Longer runs might potentially improve the results, whereas shorter runs might have negative effects.

5.4. Function pointers breaking call graphs

With the current design of MACKE, the scheduling of phase two depends on the call graph of the program. Function calls, that are not represented in the call graph, are no candidates for error chain construction. Therefore, missing links in the call graph potentially reduce the number and length of error chains and can even hide main exploits, because the corresponding analysis is never started.

Not supporting function pointers is an intentional restriction to the scope of this thesis. Precise function pointer analysis is a different research topic [43] and assuming, that function pointers are used rarely inside the programs, the resulting inaccuracy is acceptable. Nevertheless, the coverage analysis (see Section 4.7.2) has revealed, that the compiler is also allowed to introduce new function pointers (e.g. for casting the type of a function).

A more precise call graph might be beneficial for the error chains discovered by MACKE. However, this depends widely on the frequency of function pointers in the analyzed program and their position in the call graph. If the program completely avoids the usage of such pointers, the analysis is not affected at all.

5.5. Errors inside KLEE

Every step inside MACKE relies heavily on KLEE and the quality of its reports. On the good side, all improvements to KLEE also improve MACKE, but each error in KLEE also affects MACKE. KLEE is a huge project and there have already been several errors that affect existing KLEE related research [55].

The impact of these errors can hardly be predicted. MACKE is strongly correlated to KLEE, so I assume, that bugs affect both similarly, but it might also be possible, that a bugfix only improves the results from one of them and the other is degraded poorly. Nevertheless, this might have some impact, because during the 1105 MACKE runs for the evaluation, 291 KLEE runs (0.2 % of all KLEE runs) crashed, reporting several, sometimes quite random error messages.

5.6. Inappropriate selection of KLEE flags

KLEE offers dozens of flags to its users, that widely adjust KLEE's internal behavior. As listed in Appendix B, the selection of flags used for the KLEE runs inside MACKE is deliberated carefully. Nevertheless, MACKE uses almost the same flags for all analyzed programs, although distinct adoption for each program might improve the KLEE analysis.

Both tools, MACKE and KLEE, should profit from more appropriate flags, but I expect KLEE to benefit more. The KLEE runs inside MACKE are quite short compared to a sole run of KLEE. Hence, the flags have more influence on the program exploration there. In summary, both programs might be able to achieve better results and the gap between them should shrink a little.

5.7. Error chains with cycles

As already mentioned in Section 3.8.2, tracing error chains and especially the error propagation with additional KLEE runs in phase two becomes complicated, if the analyzed program uses some sort of recursive function calls. An error chain can potentially cycle several times through the same functions until the propagated error also affects a caller of the cycle.

Although this program structure is used quite rarely inside the analyzed programs, MACKE's current implementation is not able to detect this special type of error correctly. An appropriate implementation should increase the required run time for phase two but has also the potential for longer error chains and perhaps additional main exploits. So overall, I expect slightly better results for KLEE, caused by the longer run time, but MACKE should still be ahead by a good margin.

6. Related Work

This chapter gives an overview of related work to the topics covered in this thesis. Namely, it reports sources of inspiration of the chosen implementation, revisits some general concepts, discusses related strategies for compositional analysis and targeted search, gives alternatives for the error summaries used by MACKE and finally presents some other frameworks built around multiple KLEE runs.

6.1. Direct foundations and inspirations

All work in this section can be considered as intellectual ancestors of MACKE and its algorithms. The corresponding tools and algorithms are directly used inside MACKE. Hence, these are strongly related to this research.

First of all, this thesis is the consequent continuation of the introductory paper for MACKE [46]. This thesis seizes its suggestions and further deepens the research and evaluation.

For the most parts, MACKE relies on KLEE [10] for all tasks considering symbolic execution. Although this tool is quite mature today, there is still a lot of research trying to push concolic execution to its limits: analyzing programs with parallel threads [18, 17], supporting symbolic floating point variables [15, 3] and widening the analysis from C to C++ programs [36]. Unfortunately, as the time of this thesis, none of these optimzations were merged directly into MACKE, so they cannot be used for the evaluation, but I expect them to be beneficial for the results.

One core component of MACKE is the targeted search. The algorithm suggested in this thesis can not only aim for function calls but can also be used as a very basic solution for the line reachability problem: Find a realizable path covering a given line of code. This problem is discussed in more details in [39], which presents two search strategies and a hybrid version. The first strategy is the shortest distance symbolic execution (SDSE), which is almost identical to MACKE's targeted search. During the exploration of the program states, the distance to the target for each possible successor state is calculated with no consideration of the corresponding constraints and the analysis continues with the state having the shortest distance. The second strategy is the call-chain-backward symbolic execution (CCBSE). Starting from the function containing the target line, the analysis continues with all callers of this function until it reaches the starting point. Although MACKE does not use this strategy explicitly, its overall structure follows a similar pattern: After proving that an error is reachable from the entry point of a function in phase one (comparable with SDSE), it walks upwards the call graph until it reaches the entry point of the program in phase two. All intermediate functions that can no longer reach the error are not further examined. However CCBSE does not contain any type of function or error summary, so it is forced to show the feasibility of the lower parts of the program again.

6.2. General concepts

Symbolic execution has been a topic of research for multiple years. Its first conceptual design lies back in the year 1976 [34]. Over the years, several improvements to the basic schema were suggested. For example reuse of standard model checking algorithms [33], mixed symbolic and concrete execution [8] and interleaving symbolic execution with white box fuzzing [31]. Furthermore, KLEE [10] is only one representative of a bunch of other symbolic execution frameworks: EXE, the predecessor of KLEE [9], JPF-SE for JavaCode [2], CUTE with a focus on Unit-Tests for C [56], and even S2E [11], a whole platform for building own analyses based on symbolic techniques.

Similar progress in research has been achieved on the underlying solvers, starting with extensible SAT solvers [23], some special optimizations for symbolic execution [24], extending the logic with arrays and bitvectors to STP solvers [25] and finally concluding into universal solvers [19]. There are so many different solvers with different benefits existing today that it is beneficial to use multiple solvers at once and choose the fastest result, as shown in [47].

Additionally, there exist several approaches to reduce state explosion during symbolic execution: [41] collects all suitable inputs into independent groups taking distinct paths through the program; [35] suggests a heuristic, when two states can be merged during further program exploration without losing paths; [7] identifies paths that have identical side effects and executes just one of them; [12] summarizes big program states into smaller summaries; finally, [57] suggests the usage of a shared storage for all program states that shares variables ranges and the underlying constraints automatically.

6.3. Compositional analysis

More recently, compositional program analysis using symbolic execution has become a topic for new research.

IC-Cut [13] is a framework that automatically generates unit tests for each function inside a program using symbolic execution, similar to MACKE's phase one. As an additional feature, it generates summaries for cheap low-level functions.

DART [29] also decomposes the program into functions and generates unit tests. Its special feature is the environment modeling: It automatically creates an interface for each call to the environment, that simulates all kinds of return values. This allows unit tests of corner cases that would be complicated to simulate in the real environment. MACKE redirects the problem of calls to the environment to KLEE, which just links the code of the standard C library in order to execute it normally and emulates all system calls in a comparable way to DART.

SMART [27] is an extension of the DART framework discussed earlier. Instead of only generating interfaces for calls to the environment, it uses previous runs of lower level functions to build suitable replacements for the calls to these functions in all higher level functions. MACKE could use a similar strategy to improve the coverage in phase one, but for the targeted execution in phase two, the usage of relatively complex summaries for the whole function might result in a bad tradeoff. The targeted search executes only the parts that are required to reach the target, but not necessarily the whole function. Hence, the partial execution of the function might be cheaper than working with a summary of the complete function.

MicroX [28] is a framework that allows stand-alone analysis of arbitrary chunks of codes. All uninitialized input becomes a symbolic variable on the first time it is read. The extraction of the source code and a bigger structure of the analysis is completely left to the user, while MACKE does all this automatically. Nevertheless, this would hugely reduce the overhead required inside MACKE to run a function in isolation and would also allow a more fine-grained analysis.

All the above frameworks do not target specifically errors and paths to potential vulnerabilities in deeper parts of the program. Especially the targeted search mechanism is missing completely.

Unfortunately, none of the above compositional analysis tools is open source or at least publically available. With the very small number of analyzed programs in the corresponding literature, a direct comparison with MACKE is practically impossible. Additionally, this precludes a detailed comparison of the algorithms and their implementation. Regrettably, this makes conjoint improvements of the strategies and reusing the components impossible.

6.4. Targeted search

In addition to the paper used as an inspiration for the implementation of targeted search directed path exploration is also a topic in other publications. [48] compares several generic search strategies and concludes, that a “best-first strategy” achieves the best results. MACKE’s targeted search can be considered as an instance of such a strategy.

[64] uses single branches or condition as a target. It focuses on the situation, where symbolic execution finds a path to the target, but the corresponding constraints are unsatisfiable. Then, it calculates a fitness score for each possible modification of this path to find a path whose variable ranges are closer to the required condition. The search continues with this new path until the path to the target is feasible or all modifications have been proved unfeasible. In contrast to this local optimization strategy, MACKE’s targeted search is used for global navigation to more generic targets. Anyway, situations, where the targeted search selects paths that are unfeasible occurs relatively often. So a fitness score that is merged with the distance measure of MACKE’s targeted search might be able to identify feasible paths to the target earlier.

The search strategy presented by the authors of [59] has a quite similar concept as MACKE. It aims for a call to a target function and tries to escalate smaller paths consecutively to the main function using forward and backward analysis. Most interestingly, they do no static precalculation on the whole program, like MACKE does, but learn caller-callee relations and invariants during the program analysis. They claim that this strategy only analyzes the relevant program parts and nothing additional or in repetition. In phase one, MACKE analyses the whole program and finds multiple vulnerabilities all around the code. Phase two tries to lower the exploration of irrelevant code especially, when multiple errors reside close to each other, e.g. in the same function. This redundancy is not avoided by the learning exploration, so a direct comparison between both strategies – static precalculation and on the fly learning – would be informative. However, the alternative strategy is only implemented for Java programs, whereas MACKE analyzes C programs. Hence, this would require a lot of additional implementation effort.

6.5. Error summary alternatives

The biggest problem with the current implementation of MACKE is that its error summaries are only examples taken from all inputs triggering the error (see Section 5.1). I expect using a more powerful summary to be very beneficial for the results, so this section is dedicated to related work suggesting alternative ways of summaries.

The most obvious approach for summarizing the behavior of functions is presented in [27]. The main idea is that each function can be described by a disjunction of all paths through the function. In turn, a path can be described by a conjunction of its pre- and postcondition. The precondition denotes all variable ranges that lead to the corresponding path, and the postconditions describe the return value and all potential side effects. Unfortunately, their later work [14, 13] documents that, although these summaries are very powerful and accurate, they become too big and expensive for most real world functions and are only beneficial for smaller functions with special properties. From my experience, MACKE's errors are often correlated to bigger array structures that require very complex queries for the post conditions. Similar approaches [37] discusses several techniques for summarizing smaller parts of the program i.e. basic blocks (a unit one level below functions), but these are too fine-grained for MACKE's overall structure and still not implemented and tested for industrial scale programs.

The simplest solution for finding appropriate summaries in complicated situations is redirecting this task to the user. The overall workload easily becomes too big, but mixing it up with some automatic methods makes this approach feasible. [14] uses compositional symbolic execution and fuzzing to prove memory safety of a bigger Windows library. In order to have a sound and complete analysis, they must prevent all timeouts and unsound state space pruning. For each position inside the program, they require the user to find a solution to avoid it, such as to manually inline the function within the concrete context or adding a suitable summary for the problematic code. A similar strategy might

also work for MACKE's error summary. It can identify each error summary that is too complex for automatic summarization and at least report it to the user, e.g. as a warning or as an additionally rating factor in the final error report.

In the situation, where an example is easy, but too simple and the precise model of all error inputs is too complex, a model, that defines most of the erroneous inputs, but not all of them seems to be the best choice. I am not aware of any research about symbolic execution using such a mechanism, but for model checking the basic principles for this are already used successfully. Counter-Example guided abstraction refinement (CEGAR [6]) could be used to build a reduced model for some paths, that causes the error, and a check in MACKE's phase two based on this model should be affordable. Furthermore, the internal SMT logic inside KLEE can be extended with interpolation [58], the underlying principle of this abstraction refinement. It has been shown to be applicable to big benchmark problems from industrial applications [42], so it should scale to the models required for most of MACKE's error summaries. However, KLEE does not even provide an interface to include these additional queries for the SMT solver, so this is only a very theoretical alternative.

6.6. Frameworks around KLEE

Finally, there are a few frameworks built around multiple KLEE runs that also suggest and implement several improvements for the program analysis.

BovInspector [26] has, similar to MACKE, two phases of analysis: In phase one, it searches for vulnerabilities and in phase two, it uses KLEE to check the reachability of the uncovered vulnerabilities. But the internals of both phases differs widely. In phase one, BovInspector uses Fortify, a static code analysis tool, to search only for buffer overflow errors. MACKE uses KLEE also for phase one and is not limited to only one type of error. In phase two, BovInspector uses a special search strategy for KLEE, that solves the same problem as the targeted search of MACKE. Nevertheless, BovInspector calculates the paths to the vulnerable instruction before starting KLEE and KLEE is just following these static, precalculated paths. MACKE's targeted search is more generic and calculates the shortest path to the target dynamically during the execution of KLEE. Moreover, both tools handle their results differently: BovInspector automatically suggests fixes for the revealed buffer overflows, whereas MACKE assigns ratings to all vulnerabilities and leaves the fixes for them completely to the user. Automatic fixes are helpful for quick improvements on the code, but MACKE reports more errors than BovInspector and not only one type of error, so giving automatic fixes is much more complicated and calculating a score is more appropriate. Finally, MACKE does not simply ignore errors that cannot be exploited by the main function, but calculates severity scores for them that are included in the final error report.

UC-KLEE [52] shares the same fundamental idea with MACKE: It identifies path explosion as the main problem of symbolic execution and suggests analyses of smaller components of the program, i.e. each function separately as it is done in MACKE. The approach suffers mainly from the same threats to validity, but the authors propose some solutions for some of them. Their work focuses mainly on underconstrained variables, a special type of symbolic variables that builds its memory structure lazily when the analyzed program first accesses it. This completely solves the problem of nested pointer variables (see Section 5.3). Especially, the improved error reports from KLEE look very promising, as from my experience parsing more complicated information from normal KLEE errors (e.g. structs) is very unintuitive. UC-KLEE has the same problems as MACKE with false positives (see Section 5.2) and function pointers (see Section 5.4). UC-KLEE redirects both problems to the user and offers an interface to manually clarify the behavior of the analysis. However, MACKE is designed to be a fully automated analysis tool conflicting with this approach. Additionally, UC-KLEE offers functionality for comparing two versions of the same program. This is not directly supported by MACKE, but interestingly, for this UC-KLEE uses a search strategy aiming for blocks of code differing between the versions. The authors give no further details about their implementation, but MACKE's targeted search can easily be adapted to reproduce this behavior. Unfortunately, overall UC-KLEE seems to be a very basic framework: It does not combine the findings of the different components and does not use the call graph for any orientation inside the program. Furthermore, it does not construct any type of error chain nor does it try to propagate any error to higher functions. All this is implemented inside MACKE. Finally, they do not evaluate the achieved code coverage and do not compare with KLEE or any other tool.

7. Conclusion

This thesis continues the research on MACKE, an open source framework for compositional symbolic analysis. Instead of starting the analysis of the program from the main function, it decomposes the program into several units, analyzes these units separately, performs reachability analysis and combines all revealed findings to a more comprehensive error report. In order to strengthen the foundation of the tool, this thesis suggests improvements for the several downsides of the old tool: More parallelism, a better matching algorithm, and a precise targeted search. The direct comparison with the old tool shows that these changes amplifies the generated error reports: The coverage is higher, more errors are reported and the resulting error chains are longer.

With the state of the introductory paper about MACKE [46, section 5], the authors limit the validity of their findings to their small collection of 4 open source programs. Although the results may vary between different analyzed programs, this thesis hugely widens the evaluation on 16 different open source programs from various different application fields and confirms the advantages of the compositional approach with only very small restrictions. As discussed extensively in the results section, the compositional analysis achieves more coverage, especially in deeper parts of the program and discovers more vulnerable instructions than pure, whole program symbolic analysis.

However, the biggest achievement of this thesis is not represented in the sheer numbers: Starting with building the analyzed program, the whole analysis process is now completely automated and requires absolutely no human interaction. Each of the four analyzed programs in the paper took several hours of human preparation and result collection with the old implementation. With the current implementation, giving the source code of 16 programs is enough to automatically collect the filled result tables and graphs in this thesis with a month of plain execution time, but no further human work. And all this is open source, so every programmer around the world can start analyzing and improving his source code today.

8. Future Work

Although this thesis covers and deepens a variety of topics, there is still a lot of work left for future research. This final section names some fields for future work and gives some thoughts about each of them.

8.1. Beyond the scope of this thesis

During the work on this thesis, four topics were excluded explicitly, because the required amount of effort lies beyond the scope of the thesis. These topics are the support of multi-pointer variables (see Section 3.3.3), a more parallelized schedule for phase two (see Section 3.8.2), a better summary for errors (see Section 5.1) and an automatic strategy to identify false positives (see Section 5.2). Of course, these should be covered by future work.

8.2. Improve the web interface with domain experts

With all the improvements inside this thesis, MACKE has become a lot more mature, than it has been before. The MACKE paper originally suggests a severity scale for the discovered vulnerabilities and includes a web interface for easy navigation through the findings. These are not covered explicitly in this thesis. However, MACKE is now capable of handling industrial size projects and a survey with domain experts might further improve the usability of MACKE and widen the field of application.

8.3. Automatically apply MACKE on open source projects

Even without the help of an industrial partner, MACKE has an opportunity to expand beyond the scientific environment. Right now, the build and analysis process is highly automated and requires almost no human interaction. With very little effort, it is possible to set up a server that regularly checks several open source projects and archives the corresponding error reports. A deep investigation of the reports might be too expensive, but the difference between the reports of the same project can highlight the newly introduced errors in the program. Their number should be low enough to be analyzed in a matter of minutes and hopefully, the bug reports from this sort of regression test will increase the attention for MACKE.

8.4. A different source for error summaries

In the current setup, MACKE uses KLEE for error generation in phase one as well as for error propagation in phase two. However, symbolic execution is not the only source for error reports, that can be applied on the function level. For example static analysis [26] and fuzzing [30] also uncover potential errors that can be used as a foundation for phase two. Perhaps, these errors are different, more numerous or even a superset of the errors reported by KLEE. Furthermore, this affects the run time of MACKE. Hence, it is quite interesting, how MACKE with fuzzer performs compared to MACKE with KLEE or even MACKE using both as an input for phase two.

8.5. Standardize targeted search

As explained in Section 3.7, targeted search is an algorithm that is not only used by MACKE, but also by a variety of other tools. The implementation for this thesis is very basic, but building an optimized, well-tested library for targeted search seems to be beneficial for a complete area of research.

In summary, thinking about a bigger testbed, introducing some optimizations and cache layers, swapping the search strategy in existing projects and comparing the performance of the new library with the old implementation, has the potential for another thesis.

List of Figures

List of Listings

List of Tables

Bibliography

[1] David Abrahams and Stefan Seefeld. *Boost.Python*. 2016. URL: http://www.boost.org/doc/libs/1_60_0/libs/python/doc/html/index.html (visited on 12/02/2016).

[2] Saswat Anand, Corina S Păsăreanu, and Willem Visser. "JPF–SE: A symbolic execution extension to java pathfinder". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. 2007, pp. 134–138.

[3] Earl T Barr, Thanh Vo, Vu Le, and Zhendong Su. "Automatic detection of floating-point exceptions". In: *ACM SIGPLAN Notices*. Vol. 48. 1. ACM. 2013, pp. 549–560.

[4] David Beazley and William Fulton. *SWIG - Simplified Wrapper and Interface Generator*. 2016. URL: http://www.swig.org/ (visited on 12/02/2016).

[5] Eli Bendersky. *Parsing C++ in Python with Clang*. 2011. URL: http://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang (visited on 12/02/2016).

[6] Dirk Beyer and Stefan Löwe. "Explicit-state software model checking based on CEGAR and interpolation". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2013, pp. 146–162.

[7] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. "RWset: Attacking path explosion in constraint-based test generation". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. 2008, pp. 351–366.

[8] Cristian Cadar and Dawson Engler. "Execution generated test cases: How to make systems code crash itself". In: *International SPIN Workshop on Model Checking of Software*. Springer. 2005, pp. 2–23.

[9] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. "EXE: automatically generating inputs of death". In: *ACM Transactions on Information and System Security (TISSEC)* 12.2 (2008), p. 10.

[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. Vol. 8. 2008, pp. 209–224.

[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: a platform for in-vivo multi-path analysis of software systems". In: *ACM SIGPLAN Notices* 46.3 (2011), pp. 265–278.

[12] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. "Blitz: Compositional bounded model checking for real-world programs". In: *International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 136–146.

[13] Maria Christakis and Patrice Godefroid. "IC-Cut: A compositional search strategy for dynamic test generation". In: *Model Checking Software*. Springer, 2015, pp. 300–318.

[14] Maria Christakis and Patrice Godefroid. "Proving memory safety of the ANI Windows image parser using compositional exhaustive testing". In: *Verification, Model Checking, and Abstract Interpretation*. Springer. 2015, pp. 373–392.

[15] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. "Symbolic crosschecking of floating-point and SIMD code". In: *Proc. 6th Conf. Computer Systems*. ACM. 2011, pp. 315–328.

[16] Continuum Analytics, Inc. *llvmlite*. 2016. URL: `http://llvmlite.pydata.org/` (visited on 12/02/2016).

[17] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. "Efficient deterministic multithreading through schedule relaxation". In: *Proc. 23rd ACM Symp. Operationg Systems Principles*. ACM. 2011, pp. 337–351.

[18] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. "Stable Deterministic Multithreading through Schedule Memoization." In: *OSDI*. Vol. 10. 2010, p. 10.

[19] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. 2008, pp. 337–340.

[20] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact". In: *Empirical Software Engineering* 10.4 (2005), pp. 405–435.

[21] Joe W Duran and Simeon C Ntafos. "An evaluation of random testing". In: *IEEE Transactions on Software Engineering* 4 (1984), pp. 438–444.

[22] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. "The matter of heartbleed". In: *Proceedings of the 2014 Internet Measurement Conference (IMC)*. ACM. 2014, pp. 475–488.

[23] Niklas Eén and Niklas Sörensson. "An extensible SAT-solver". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2003, pp. 502–518.

[24] Ikpeme Erete and Alessandro Orso. "Optimizing constraint solving to better support symbolic execution". In: *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2011, pp. 310–315.

[25] Vijay Ganesh and David L Dill. "A decision procedure for bit-vectors and arrays". In: *International Conference on Computer Aided Verification*. Springer. 2007, pp. 519–531.

[26] Fengjuan Gao, Linzhang Wang, and Xuandong Li. "BovInspector: automatic inspection and repair of buffer overflow vulnerabilities". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM. 2016, pp. 786–791.

[27] Patrice Godefroid. "Compositional dynamic test generation". In: *ACM SIGPLAN Notices*. Vol. 42. 1. ACM. 2007, pp. 47–54.

[28] Patrice Godefroid. "Micro execution". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 539–549.

[29] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *ACM SIGPLAN Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[30] Patrice Godefroid, Michael Y Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing". In: *Queue* 10.1 (2012), p. 20.

[31] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. "Automated Whitebox Fuzz Testing." In: *Network and Distributed System Security Symposium*. Vol. 8. 2008, pp. 151–166.

[32] *International Standard ©ISO/IEC ISO/IEC 9899:201x Programming languages — C*. 2008.

[33] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. "Generalized symbolic execution for model checking and testing". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. 2003, pp. 553–568.

[34] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[35] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient state merging in symbolic execution". In: *ACM SIGPLAN Notices* 47.6 (2012), pp. 193–204.

[36] Guodong Li, Indradeep Ghosh, and Sreeranga P Rajan. "KLOVER: A symbolic execution and automatic test generation tool for C++ programs". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 609–615.

[37] Yude Lin, Tim Miller, and Harald Søndergaard. "Compositional Symbolic Execution using Fine-Grained Summaries". In: *Software Engineering Conference (ASWEC), 2015 24th Australasian*. IEEE. 2015, pp. 213–222.

[38] LLVM Project. *Writing an LLVM Pass*. 2016. URL: `http://llvm.org/docs/WritingAnLLVMPass.html` (visited on 12/02/2016).

[39] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. "Directed symbolic execution". In: *Static Analysis*. Springer, 2011, pp. 95–111.

[40] Mahadevan R and Continuum Analytics, Inc. *LLVMPY*. 2014. URL: `http://www.llvmpy.org/` (visited on 12/02/2016).

[41] Rupak Majumdar and Ru-Gang Xu. "Reducing test inputs using information partitions". In: *International Conference on Computer Aided Verification*. Springer. 2009, pp. 555–569.

[42] Kenneth L McMillan. "Applications of Craig interpolants in model checking". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. 2005, pp. 1–12.

[43] Ana Milanova, Atanas Rountev, and Barbara G Ryder. "Precise call graphs for C programs with function pointers". In: *Automated Software Engineering* 11.1 (2004), pp. 7–26.

[44] Martin Novak and Mate Soos. *klee/scripts/klee-clang*. 2016. URL: `https://github.com/klee/klee/blob/master/scripts/klee-clang` (visited on 12/02/2016).

[45] Saahil Ognawala. *MACKE*. 2016. URL: `https://github.com/tum-i22/macke` (visited on 12/02/2016).

[46] Saahil Ognawala, Martin Ochoa, Alexander Pretschner, and Tobias Limmer. "MACKE: compositional analysis of low-level vulnerabilities with symbolic execution". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM. 2016, pp. 780–785.

[47] Hristina Palikareva and Cristian Cadar. "Multi-solver support in symbolic execution". In: *Computer Aided Verification*. Springer. 2013, pp. 53–68.

[48] Alexander Pretschner. "Classical search strategies for test case generation with constraint logic programming". In: *Formal Approaches to Testing of Software*. 2001, pp. 47–60.

[49] LLVM Project. *The LLVM gold plugin*. 2016. URL: `http://llvm.org/docs/GoldPlugin.html` (visited on 12/02/2016).

[50] Python Software Foundation. *ctypes - A foreign function library for Python*. 2016. URL: `https://docs.python.org/3.4/library/ctypes.html` (visited on 12/02/2016).

[51] Python Software Foundation. *Extending Python with C or C++*. 2016. URL: `https://docs.python.org/3.4/extending/extending.html` (visited on 12/02/2016).

[52] David A Ramos and Dawson Engler. "Under-constrained symbolic execution: correctness checking for real code". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 49–64.

[53] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise interprocedural dataflow analysis via graph reachability". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. 1995, pp. 49–61.

[54] Ronald Rivest. "The MD5 message-digest algorithm". In: *RFC 1321* (1992).

[55] Eric F Rizzi, Sebastian Elbaum, and Matthew B Dwyer. "On the techniques we create, the tools we build, and their misalignments: a study of KLEE". In: *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 132–143.

[56] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 5. ACM. 2005, pp. 263–272.

[57] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. "Multise: Multi-path symbolic execution using value summaries". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 842–853.

[58] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. "Interpolation-based function summaries in bounded model checking". In: *Haifa Verification Conference*. Springer. 2011, pp. 160–175.

[59] Nishant Sinha, Nimit Singhania, Satish Chandra, and Manu Sridharan. "Alternate and learn: Finding witnesses without looking all over". In: *International Conference on Computer Aided Verification*. Springer. 2012, pp. 599–615.

[60] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[61] O. Tange. "GNU Parallel - The Command-Line Power Tool". In: *;login: The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47. URL: `http://www.gnu.org/s/parallel`.

[62] Gregory Tassey. "The economic impacts of inadequate infrastructure for software testing". In: *National Institute of Standards and Technology, RTI Project* 7007.011 (2002).

[63] Antti Valmari. "The state explosion problem". In: *Lectures on Petri nets I: Basic models*. Springer, 1998, pp. 429–528.

[64] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. "Fitness-guided path exploration in dynamic symbolic execution". In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE. 2009, pp. 359–368.

A. Source code repositories

This appendix gives a list of all used source code. It names places, where it can be obtained (i.e. a Download-URL or a version control repository) and the version used for the experiments in the thesis. All links were checked in December 2016.

A.1. Components for MACKE

MACKE ..... `https://github.com/hutoTUM/macke`
`b9efa5290aaa5ec9686727fd8466450f906b83eb`
forked from "old MACKE" (`https://github.com/tum-i22/macke`)

MACKE opt llvm ..... `https://github.com/hutoTUM/macke-opt-llvm`
`33eb4998131cd585a8e8d3f0875196155db374db`

KLEE with targeted search ..... `https://github.com/hutoTUM/klee`
`21c0e2497e37e7fbe09f6394f4ece7ad60672e8e`
forked from KLEE 1.2.0 (`https://github.com/klee/klee`)

make+llvm ..... `https://github.com/hutoTUM/MakeAdditions`
`45ddd12d5a9a91b3f56b5909a6a8af01bed66daf`

Finally a small gratitude to GNU parallel: This tool was quite helpful for scheduling and executing multiple KLEE runs in parallel. As promised after the installation, here is the citation [61] for the authors.

A.2. Analyzed programs

bc 1.06 ..... https://ftp.gnu.org/gnu/bc/bc-1.06.tar.gz

bison 3.0.4 ..... https://ftp.gnu.org/gnu/bison/bison-3.0.4.tar.xz

bzip2 1.0.6 ..... http://bzip.org/1.0.6/bzip2-1.0.6.tar.gz

coreutils 6.10 ..... https://ftp.gnu.org/gnu/coreutils/coreutils-6.10.tar.gz

coreutils 8.25 ..... https://ftp.gnu.org/gnu/coreutils/coreutils-8.25.tar.xz

diff 3.4 ..... https://ftp.gnu.org/gnu/diffutils/diffutils-3.4.tar.xz

flex 2.6.0 http://downloads.sourceforge.net/project/flex/flex-2.6.0.tar.xz

flex SIR orig.v0 [20] ..... http://sir.unl.edu/portal/bios/flex.php

goahead 3.6.3 ..... https://embedthis.com/software/goahead-3.6.3-src.tgz

grep 2.25 ..... https://ftp.gnu.org/gnu/grep/grep-2.25.tar.xz

grep SIR orig.v0 [20] ..... http://sir.unl.edu/portal/bios/grep.php

jq 1.5 ..... https://github.com/stedolan/jq/archive/jq-1.5.tar.gz

less 481 ..... https://ftp.gnu.org/gnu/less/less-481.tar.gz

libuv 1.9.1 ..... https://github.com/libuv/libuv/archive/v1.9.1.tar.gz

lz4 r131 ..... https://github.com/Cyan4973/lz4/archive/r131.tar.gz

ngircd 23 ..... http://ngircd.barton.de/pub/ngircd/ngircd-23.tar.xz

sed 4.2.2 ..... https://ftp.gnu.org/gnu/sed/sed-4.2.2.tar.bz2

tar 1.29 ..... https://ftp.gnu.org/gnu/tar/tar-1.29.tar.xz

zopfli 1.0.1 .. https://github.com/google/zopfli/archive/zopfli-1.0.1.tar.gz

B. Used and discarded flags for KLEE

This appendix gives a list of all flags I used for the KLEE runs triggered by MACKE. Since the flags differ from the ones used by old MACKE, it also names all old flags, that are no longer used. Additionally, it gives reasons, why I decide to select each flag or discard respectively.

Used flags

`--allow-external-sym-calls` allows calls with symbolic arguments to external functions. The symbolic arguments are concretized to one representative of their range before the external call is processed. This leads to a loss of almost all states but is still better than directly stop the execution of the path.

`--libc=uclibc` replaces a lot of external calls to C lib functions with the uclibc bitcode variants of the called functions. This widely extends the range of analyzable programs.

`--max-memory=1000` limits the maximum amount of memory allocatable by KLEE in MBs. With multiple KLEE runs in parallel, this limit asserts, that one analysis assigning huge amounts of memory, does not interfere with the other concurrent KLEE runs.

`--only-output-states-covering-new` reduces the number of generated test cases, so only tests covering new code are stored. This generates fewer test cases with a higher overall quality.

`--optimize` is very similar to the effect of compiling with `-O3`. It reduces the amount of instructions needed in the bitcode and thereby simplifies the analysis. Furthermore, production code is very likely to be also optimized.

`--output-source=false` prevents KLEE from storing a human readable bitcode version of the analyzed program. MACKE stores all relevant bitcode file in its output directory. Storing an additional version is a waste of storage, especially with the much bigger size of the human readable form.

`--posix-runtime` allows the usage of the `--sym-args`, `--sym-files` and `--sym-stdin` flags. The first two flags are a must-have for the symbolic execution of the main function, but quite irrelevant to the analysis of the components. Nevertheless, the last flag enables symbolic standard input, which might be read in all parts of the program. Therefore, this is used for all KLEE runs triggered by MACKE.

`--stats-write-interval=3600` and `--istats-write-interval=3600` control the frequency, how often KLEE updates the files with coverage information during a run. This is helpful for live run-time analysis, but this feature is not used by MACKE. Therefore, the high number of 3600 seconds basically disables the updates. Nevertheless, KLEE writes the final summary. With these files being one of the biggest files in the output directory, this removes unnecessary write operations of hundred of gigabytes.

`--sym-stdin` makes the content read from standard input symbolic. This allows a deeper analysis of components, that actually reads from stdin. For all components, that do not read anything, the symbolic content is left unconstrained and is thereby basically ignored.

`--watchdog` enforces the run-time limits for the overall KLEE analysis and the time spent on one instruction. MACKE requires both limits to control its run-time. Both limits are set with other flags.

Discarded flags

`--disable-inlining` prevents the optimizer from inlining any functions. All MACKE function (i.e. entry points) have attributes set to prevent inlining anyway and for the rest of the code inlining is absolutely fine and reduces the effort of the analysis.

`--max-memory-inhibit=false` modifies KLEE's forking behavior when it is close to its memory limit. I do not see any reason, why MACKE should not keep the KLEE default (`true`) at this point.

`--max-static-{cpfork,fork,solve}-pct=1` control the percentage of time KLEE can spend on one task. Restricting all of them to 100% is meaningless. Additionally, these are the default variables.

`--max-sym-array-size=4096` limits the maximum size of an array, that is marked as symbolic. I do not see any reason to limit this explicitly and therefore, I kept the KLEE default and do not limit the symbolic array size.

`--randomize-fork` randomize the branch, that is taken first after forking. Per default, KLEE always chooses the true-branch. This adds more randomness to the analysis and I do not see any benefit from it. So I decide use KLEE's default.

`--simplify-sym-indices` simplifies symbolic indices using equations from other constraints. This feature seems to be experimental and is deactivated by default. Therefore, I also deactivate it.

`--switch-type=internal` modifies KLEE's switching strategy. Internal is the default value, so there is no reason for adding it explicitly.

`--use-batching-search`, `--batch-instructions=10000`, `--search=nurs:covnew` and `--search=random-path` determines KLEE's search strategy. This combination of variables is exactly the default, so there is no need to add them explicitly.

`--use-cex-cache` enables counter example caching. It is enabled by default and should not be added explicitly.

`--use-forked-solver` runs the SMT solver in a forked process. This is activated by default and thereby not added explicitly.

`--write-cov`, `--write-cvcs` and `--write-smt2s` add additional information about the generated SMT queries and coverage of each test case to the output directory. None of this information is used by MACKE and therefore, adding them purely wastes space.

Flags redirected to the user

All the flags in this section can directly be given to MACKE. It passes the variables to all underlying KLEE runs. None of them is required so this section also denotes the corresponding default variable.

`--max-instruction-time` Limits the maximum time, KLEE can spend analyzing a single instruction. MACKE uses a default of 12 seconds (10% of the maximum execution time) for this flag. Anyway, the user may want a more intense analysis, so this parameter can also be adjusted.

`--max-time` Limits the maximum time, MACKE spends analyzing one component. MACKE uses a default of two minutes per component, but the user may want to cut or extend the overall runtime.

`--sym-args` makes the arguments for the main function (`char** argv`) symbolic. The number of parameters depends on the analyzed program and therefore, must be set by the user. Per default, MACKE does skip the analysis of the main function, if this parameter is missing.

`--sym-files` the input read from files is also considered as a symbolic variable. This variable depends on the analyzed program. Not all programs read from external files and some may need bigger files. Per default, MACKE does not use any symbolic files.