Designing and Assessing a Benchmark Set for Fault Localization Using Fault Injection

Bachelor Thesis in Computer Science

03.08.2023

Ludwig-Maximilians-Universität München

Moritz Bierwirth

Supervisor: Prof. Dr. Dirk Beyer
Mentor: Matthias Kettl

Acknowledgements

First of all, I would like to thank Prof. Dr. Dirk Beyer for the opportunity to write this thesis at the chair of Software and Computational Systems Lab. Special thanks go to my Mentor Matthias Kettl, for the outstanding support. Not only did he take the time for weekly meetings, but in addition he was available at any time of the day or night via Zulip to answer my questions. Further thanks are due to Dr. Stefan Winter, who was a perfect point of contact for questions about fault localization and provided great and especially fast support for the Coccinelle tool. I would also like to thank my family, friends and flatmates for their emotional and understanding support.

Abstract

With the further development of communication systems, which are becoming increasingly complex, the number of faults in the software of these systems is also rising. To be able to keep up with this growth, fault localization techniques are becoming increasingly important. Researchers or research groups proposing a new technique for fault localization usually evaluate it on programs with known faults. The main goal of our approach is to create a benchmark set, that can be used to evaluate these techniques. We achieve this goal by creating V-FIT, Verified Fault Injection Tool. It combines the two verifiers CPAchecker and UAutomizer to verify a given subset of safety tasks from the SV-COMP benchmark set and includes the fault injection tool Coccinelle to inject the faults. V-FIT verifies each file after injection again and creates a new fault localization benchmark set, consisting of a sensible folder structure and for each injected fault a metadata file and two files specifying the fault and the exact location. Furthermore, we evaluate the fault localization benchmark set by doing a quantitative analysis to show the performance of V-FIT and a qualitative analysis to examine the weaknesses and strengths of the created benchmark set. In total, V-FIT only processed 3 percent of the subset of tasks from the SV-COMP benchmark set successfully, but nevertheless, a total of 858 fault injections were created and thus a basis for further work was created.

Contents

List of Figures

List of Programs

List of Tables

1 Introduction

In the meantime, we are dependent on software in almost every sector. The rise of software usage and adoption leads to more and more increase in scale and complexity. This boosts the number of faults in programs. Finding the bugs in a program takes a lot of time and costs a lot of money [19]. Fault localization, which involves determining the location of the fault, was previously a manual task and therefore very time-consuming. Moreover, the manual Fault Localization depends on the experience, judgment and intuition of the developer who searches for the fault. Because of the time and cost involved, there is a lot of research on automating fault localization [19]. Furthermore, this fact established the development of fault localization techniques. The evaluation of these techniques is quite difficult because there exist only a few benchmark sets to evaluate them.
Our goal with this approach is to create a Benchmark set for fault localization to evaluate these techniques. To reach this goal we created V-FIT, Verified Fault Injection Tool, to inject the faults into given `C` programs from the SV-COMP benchmark set, an already existing benchmark set. We call these programs in the following *base files*. In the first step, one *base file* is verified by two verifiers, CPAchecker [4] and UAutomizer [9], to prove its validity.

Program 1: Example of a *base file*

```
1 int abs (int x) {
2     int abs = 0;
3         if (x < 0) {
4             abs = -x;
5         } else {
6             abs = x;
7         }
8     return abs;
9 }
```

Program 2: Fault injection into a *mutant*

```
1 int abs (int x) {
2     int abs = 0;
3         if (x < 0) {
4             abs = +x;
5         } else {
6             abs = x;
7         }
8     return abs;
9 }
```

A simple example of a *base file* is provided in Program 1, where one can see

a method to calculate the absolute value of a given number. Afterward, the fault injection takes place. In our work, we accomplish that with the help of COCCINELLE [11], a fault injection tool. Program 2 shows the *base file* after the fault injection. We call this file *mutant*. In this case, all occurrences of the `-` symbol are replaced by a `+` symbol. This example shows, that even a small change alters the complete behavior of the program.
In our approach, we injected more complex faults, which we explain in detail in Chapter 3.
Fault localization techniques usually output lines where they suspect the fault. We created a `.diff` file, an example can be seen in Program 3, to evaluate these techniques by checking if the output lines of the technique match the lines in the `.diff` file. The `.diff` file includes the *base file* name, the *mutant* file name and the changes. If there is more than one fault injected in the *base file*, for each fault injection, a *mutant* and a `.diff` file are created, separately.

Program 3: Point out the fault location in a `.diff` file

```
1  diff --git a/example.c b/example.m
2  index c7b7b30..219290e 100644
3  --- a/example.c
4  +++ b/example.m
5  @@ -1,7 +1,7 @@
6   int abs (int x) {
7     int abs = 0;
8        if (x < 0) {
9  -        abs = -x;
10 +        abs = +x;
11       } else {
12          abs = x;
13       }
```

After the fault injection, each *mutant* is verified by CPACHECKER and UAUTOMIZER again, to prove the *mutant* as invalid. If this is the case, we generate a `.yml` file, to store the metadata, an example can be seen in Program 4. In this file, the format version, the data model, and the programming language is specified. Furthermore, the *mutant* name, the `.diff` file name and the properties are mentioned. For our benchmarks, we only use the property *unreach call*, explained in Chapter 3.

Program 4: Metadata stored in a `.yml` file in the fault localization benchmark set

```
1  format_version: '2.0'
2
```

```
3  input_files: sanfoundry_24-1_1.c
4
5  diff_file: sanfoundry_24-1_1.diff
6
7  properties:
8  - property_file: ../properties/unreach-call.prp
9    expected_verdict: false
10
11 options:
12   language: C
13   data_model: ILP32
```

All the produced files, the *mutant*, the `.diff` and the `.yml` file are stored in our new fault localization benchmark set.
In this thesis, we present the process of creating this new fault localization benchmark set by using V-FIT, its performance, as well as the evaluation of the fault localization benchmark set by a quantitative and qualitative analysis.

2 Related Work

As mentioned in Chapter 1, there is a lot of research on fault localization. Fault localization techniques are mostly evaluated by using artificial faults and not real faults. One interesting evaluation of fault techniques shows, that artificial faults differ from real faults [15]. They evaluated 7 fault localization techniques on both, artificial and real faults. They used 2995 artificial faults in 6 real-world programs and 310 real faults in the same programs. They replicated previous studies on evaluating fault localization techniques with artificial faults and confirmed 70 % of the results, 30 % were falsified. Another interesting fact was, that the results of the previous studies on artificial faults were statistically up to 60 % insignificant on real faults and the other 40 % were falsified.

The *Common Vulnerabilities and Exposures*, *CVEs*, is a database of publicly known security vulnerabilities on the Internet. This repository can be used, for example, for intrusion detection, security information management, or vulnerability assessment. One tool for vulnerability localization in *CVEs* is called VulnLoc [1]. It automatically reveals vulnerabilities in one given exploit with high accuracy [17]. The approach examined 43 *CVEs* arising in large real-world applications. VulnLoc identified vulnerability locations in about 88 % of the given *CVEs*. The tool includes on the one hand the fuzzer ConcFuzz, which takes a vulnerable program and an exploit as input. It produces a test case for each vulnerability or runs into a timeout. On the other hand, it includes a ranker, which provides necessity and sufficiency scores for each location. This combining of directed test-generation techniques with statistical localization allows vulnerability localization in large real-world programs.

We also want to mention a database and extensible framework to enable controlled testing studies for Java programs, Defects4J[2] [10]. Defects4J

[1] `https://github.com/VulnLoc/VulnLoc`

[2] `https://github.com/rjust/defects4j`

contains 357 real bugs from 5 real-world open-source programs. The framework contains artifacts and bug metadata for each bug. These files include revisions from the programs version control system, a patch of isolated bugs, which is the difference between the bug and the fix for it and a list of individual tests that expose the bug. For each test, they store the name, root cause and stack trace. Furthermore, they emphasize extensibility as the main feature, because of the ability to add additional bugs easily to the programs. This is possible because DEFECTS4J builds on top of the program's version control and build systems. With DEFECTS4J it is possible to enable reproducible studies in software testing research. The framework contains artifacts and bug metadata for each bug. These files include revisions from the program's version control system, a patch of isolated bug, which is the difference between the bug and the fix for it and a list of individual tests that expose the bug. For each test, they store the name, root cause and stack trace.

3 Background

In the following section, we describe all tools and techniques that are necessary to understand our approach. First of all, we address the two used verifiers CPAchecker [4] and UAutomizer [9]. Afterward we deal with the fault injection tool Coccinelle [11] and the SV-COMP benchmark set [1]. As a last, we provide an overview of different fault types.

3.1 CPAchecker

For our approach, we could use any software verifier that participates in SV-COMP 2023. One of the used verifiers so far is CPAchecker, a formal verification framework [4]. CPA stands for *configurable program analysis*. It is a concept to combine data flow analysis with model checking [3]. We choose this verifier because overall it was the third-best verifier of SV-COMP 2023, the 12th Competition on Software Verification [1], so we can trust the developers of this tool and use it off-the-shelf to verify our benchmarks. The second best verifier at SV-COMP 2023 was PeSCo [16], a machine learning approach that uses CPAchecker as a base verifier in six different configurations. This algorithm selection is unnecessary for our approach, therefore we decided to not use this verifier.

3.2 UAutomizer

The second and last Verifier we use is UAutomizer [9]. It verifies safety properties based on an automata-theoretic approach to software verification. It participated also at the SV-COMP 2023 and was the best verifier overall [1]. Thats the reason why we choose this verifier to proof our benchmarks.

3.3 Fault Types

We use specific types of Faults for the fault injection. We choose them from the most frequent fault types proposed by a field data study with 668 software errors found in 12 widely used software systems [7].

Type	Description
MFC	Missing Function Call
MVIV	Missing Variable Initialization using a Value
MVAV	Missing Variable Assignment using a Value
MVAE	Missing Variable Assignment using an Expression
MIA	Missing IF construct Around Statements
MIFS	Missing IF construct plus Statements
MIEB	Missing IF construct plus Statements plus ELSE Before Statements
MLC	Missing AND/OR clause in branch condition
MLPA	Missing small and localized part of the algorithm
WVAV	Wrong Value Assigned to Variable
WPFV	Wrong Variable used in Parameter of Function call
WAEP	Wrong Arithmetic Expression in Parameter of Function Call

Figure 1: Fault types overview for possible fault injection [13]

For our approach, we used four of the twelve proposed faults presented in Figure 1. The first is **MFC**, Missing Function Call, which affects function calls that do not return any value or do not make use of the return value. Second, there is **MVIV**, Missing Variable Initialization using a Value, which is dedicated to variables that are not assigned by a value. **MVAV**, Missing Variable Assignment using a Value, refers to variables whose assignment by a value is missing. The next is **MVAE**, Missing Variable Assignment using an Expression, the same as the one before, only an expression is used for the assignment of the variable. Fault Type **MIA**, Missing IF construct Around statements, describes that only a statement exists but there is a need for an if construct. When there is no if construct and no within statements, its **MIFS**, Missing IF construct plus Statements. **MLC**, Missing AND/OR clause in branch condition, which refers to loops and if constructs that lack an

AND/OR clause. In addition, there is **MLPA**, Missing small and localized part of the algorithm. It explains the missing of a brief, location-based part of the program. The **WVAV** Fault Type, Wrong Value Assigned to Variable, describes that the value of a variable is not assigned correctly. When the parameter of a function contains a wrong variable its the **WPFV**, Wrong Variable used in Parameter of Function call Fault Type. As last **WAEP**, Wrong Arithmetic Expression in Parameter of Function Call, displays also a parameter of a function but refers to a within wrong mathematical statement. For our approach we started a sample run with all different fault types and on the base of this we decided on four of them. We inject **MIA**, **MVAE**, **MVAV** and **WVAV** in the tasks given to our program, because they differ from each other and we thus achieve a great diversity in our new benchmark set. In Chapter 5 is explained how we inject these fault types by using Coccinelle in detail.

3.4 Coccinelle

To inject a fault in the given *base file*, we use Coccinelle [11]. It is an open-source software for automated rewriting of C code. Coccinelle provides the Semantic Patch Language to create transformations or desired matches in C Code, named semantic patches [14]. A semantic patch consists of a sequence of rules, each of which begins with context information denoted by a pair of @@s, one can see in Program 5. The context information declares a set of metavariables, which can be any term of the kind specified in its declaration (identifier, expression, etc.). In our example, there are only five statement metavariables declared. The transformation rule is represented by a term with modifiers - and + at the beginning of the line to indicate the code to be removed or added, respectively. Furthermore, the question marks before the second till the fifth statement display that these statements are optional, the first is required. This shows that for the **MIA** fault type we only inject this fault into if constructs including one to five statements.

Program 5: Semantic patch given by a Coccinlle template

```
1 @@
2 statement s1,s2,s3,s4,s5;
3 @@
4 (
5 - if(...){
6 s1
7 ? s2
8 ? s3
9 ? s4
```

```
10  ? s5
11  -}
12  )
```

Because of the comprehensible syntax and clear structure of these semantic patches, Coccinelle fits perfectly to inject our faults into the *base files* of the SV-COMP benchmark set.

3.5 SV-COMP Benchmark Set

To create a new fault localization benchmark set we use a selected part of tasks of the SV-COMP benchmark set. This is a collection of verification tasks for evaluating the effectiveness and efficiency of state-of-the-art verification technology. We chose this set because everybody can contribute and the category Reachsafety, which we are using consists of many sub-categories. After several research and development groups have submitted tasks, they were checked and probably removed, because of no property encoded or unknown architecture. Some tasks may contain compiler warnings or memory model fails, so they were technically improved. Therefore the set is qualitatively high and fits perfectly for our approach.
Furthermore, it was the set used for the International Competition on Software Verification SV-COMP 2023 and was also part of the 5th International Competition on Software Testing, Test-Comp 2023 [2], a comparative evaluation of automated test creation tools, which take place annually.

3.5.1 Structure

In this section, we describe the structure of the SV-COMP benchmark set. Due to its large scope, we only present the parts of the set that are relevant for our approach.

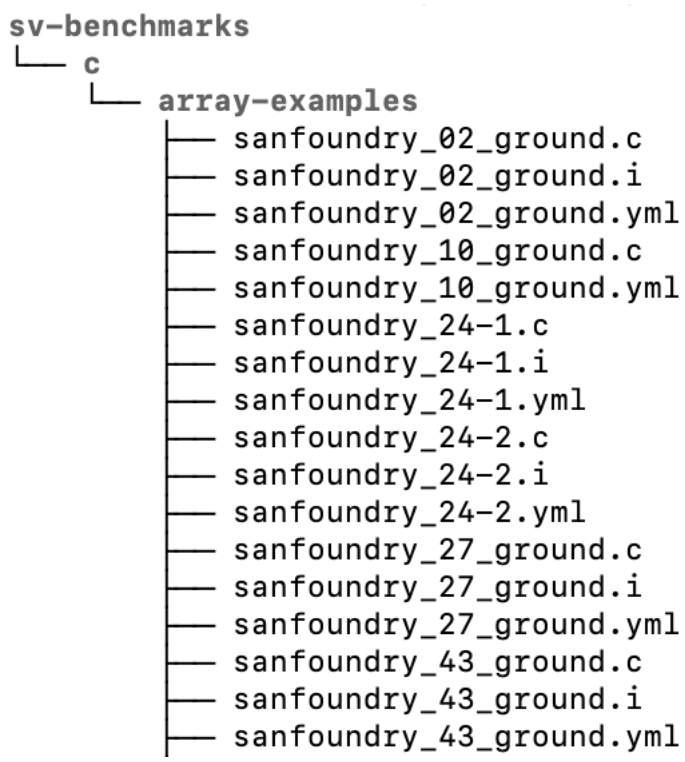

Figure 2: Excerpt of the SV-COMP benchmark set folder structure

The folder structure starts with the folder `sv-benchmarks` followed by `/c`, which specifies the programming language type of the files included, which can be seen in Figure 2. Next exists a sub-category, in our example `/array-examples`, which describes the included files in more detail.
Lastly, we got the `.yml` file and the related `.c` or `.i` file, or both. The `.c` extension is for not preprocessed files and the `.i` extension for preprocessed files. In our approach we use the given input file, specified in the `.yml` file, regardless of the indicated file extension, this is why we call the files *base files*.

Program 6: Metadata stored in a SV-COMP benchmark `.yml` file

```
1  format_version: '2.0'
2
3  input_files: 'sanfoundry_24-1.i'
4
5  properties:
6    - property_file: ../properties/no-overflow.prp
7      expected_verdict: true
8    - property_file: ../properties/termination.prp
9      expected_verdict: true
```

```
10    - property_file: ../properties/unreach-call.prp
11      expected_verdict: true
12
13  options:
14    language: C
15    data_model: ILP32
```

The metadata is represented by the `.yml` file, an example can be seen in Program 6. First, it includes the string format version. Second, the program files to be executed and third the specified properties. For our approach, we focus on the *unreach-call* property. It means that, if the expected verdict flag is set to true, a certain function call must not be reachable in the *base file*. As last it displays options including language type, in our case the C programming language and data model, either 32 or 64-bit architecture.

4 Creating a Benchmark set using VFIT

In this section we describe the basic workflow of V-FIT, Verified Fault Injection Tool. Afterward we give a detailed explanation of the fault localization benchmark set we generated.

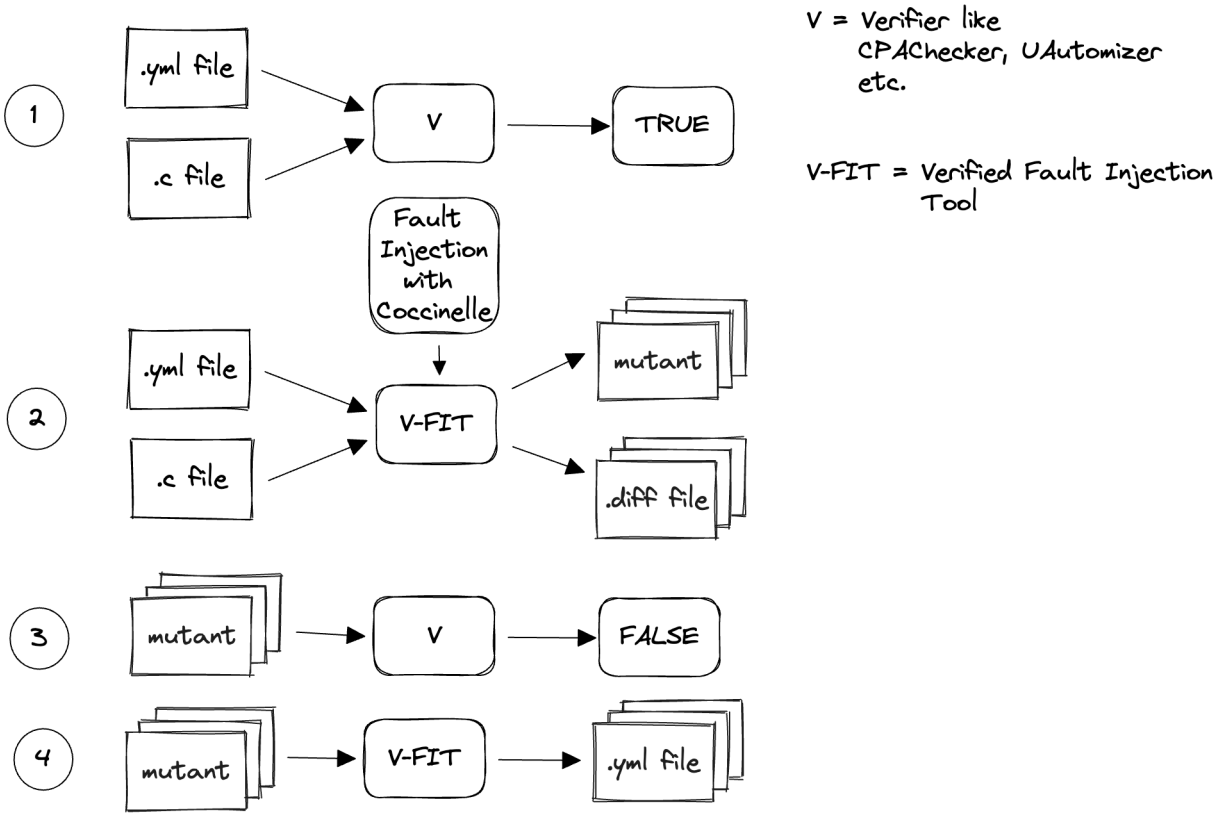

Figure 3: V-FIT step by step

4.1 V-FIT Basic Workflow

For our approach we design a program and call it V-FIT, Verified Fault Injection Tool. First of all, we filter tasks of the SV-COMP benchmark set that fit our requirements. The tasks must have the specification *unreach-call* as described in Chapter 3. After the generation of this subset of tasks, V-FIT allows us to verify the *base file* with, so far, CPAchecker and UAutomizer, as one can see in the first step of Figure 3. If both verifiers prove the *base file* to be valid, we continue with the second step in Figure 3, to inject the fault given by a Coccinelle file. Coccinelle then generates as many *mutants* as faults are injected in the *base file*. When this process is done, for each *mutant file* a `.diff` file is created to store the changes separately. Then in the third step in Figure 3, each mutant is checked by CPAchecker and UAutomizer again, and if both verifiers assessed it as invalid, for each *mutant*, a new `.yml file` is created as one can see in the fourth step in Figure 3. This shows that we designed a wrapper to combine the two verifiers and Coccinelle to generate our new fault localization benchmark set.

4.2 Fault Localization Benchmark Set Structure

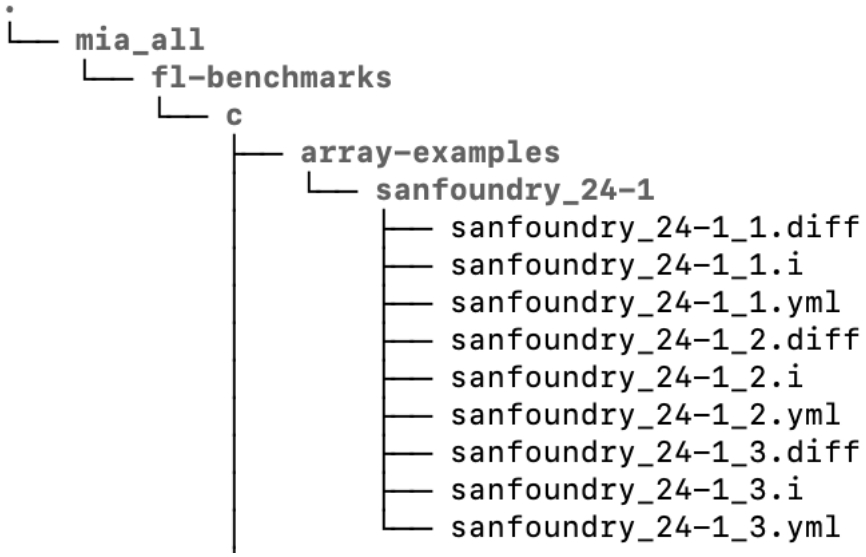

Figure 4: Fault localization benchmark set folder structure

We sort our results in a folder structure inspired by the SV-COMP benchmarks set. However, it differs in that for our purpose we start the structure with a folder named like the fault type we injected. In our example, which can be seen in Figure 4 it is the **MIA** fault type. Continuing there is `fl-benchmarks/c/`, `array-examples/sanfoundry_24-1` as in the SV-COMP benchmark set, only the prefix differs from sv to fl. We choose this structure because for future work there could be added more programming languages. Afterward, the sub-categories describe the purpose of the included files in more detail. In our example, it is `array-examples/sanfoundry_24-1`. Lastly, we got the `.yml` file and the related `.diff` file, and the `.c` or `.i` *mutant*. The `.c` extension is for not preprocessed files and the `.i` extension for preprocessed files, the same as in the SV-COMP benchmark set. The filenames consist of the parent sub-directory name and the *mutant* index number as a suffix before the extension.

4.3 Included Files in the Fault Localization Benchmark Set

4.3.1 YAML File

The Meta Data is represented by a `.yml` file. In comparison to the `.yml` file in the SV-COMP benchmark set, it has the produced `.diff` file after the input files entry and all unnecessary property specifications are deleted, as one can see in Program 7. Deletions are highlighted in red and bold, and insertions in green and cursive. Important here is that the input file changes, because of the various *mutants* created. As one can see, we added a number before the extension to exactly label the *mutant*.

Program 7: Comparison of the SV-COMP benchmark and fault localization benchmark metadata

```
1  diff --git a/sv-example.yml b/fl-example.yml
2  index 8ce06b6..f469f48 100644
3  --- a/sv-example.yml
4  +++ b/fl-example.yml
5  @@ -1,14 +1,12 @@
6   format_version: '2.0'
7
8  -input_files: 'sanfoundry_24-1.i'
9  +input_files: sanfoundry_24-1_1.c
10 +diff_file: sanfoundry_24-1_1_1.diff
11
12  properties:
13 -  property_file: ../properties/no-overflow.prp
14 -  expected_verdict: true
15 -  property_file: ../properties/termination.prp
16 -  expected_verdict: true
17 -  property_file: ../properties/unreach-call.prp
18 -  expected_verdict: true
19    property_file: ../properties/unreach-call.prp
20 +  expected_verdict: false
21
22  options:
23    language: C
24    data_model: ILP32
```

4.3.2 DIFF File

Because there is a seperate `.diff` file for each *mutant*, the file shows us the compared files and afterward the index hash, which can be seen in Program 3.

Then it shows for the one file a + and for the other a - to signal in which file there were deletions and extensions. The first digit after the @-sign in the next line displays how many lines in the first file are changed and the digit after the comma indicates how many characters are changed. Finally, the modified code appears with - for a deletion and + for an extension.

4.3.3 Mutant File

The *mutant* difference to the *base file* is that it contains the fault, of course. Important is that for each fault injection, there exists a separate textitmutant. For example, there are nine if statements in the *base file*, and the fault type is **MIA**, then every if statement, except the whithin statement, is deleted. V-FIT thus produces nine *mutants*, each for every deleted if statement.

4.4 Approach Advantages

For our Evaluation, V-FIT is also integrated into BenchExec, a framework for reliable benchmarking [5].
One advantage of our approach is that it is arbitrarily expandable. We can easily add other verifiers to prove the tasks.
Another great advantage is that, assuming more and more research and development groups contribute to the SV-COMP benchmark set, our fault localization benchmark set grows simultaneously. Of course, it has to be executed.
Furthermore, through Coccinelle our newly created set is extended by a semantic component.

5 Implementation

The Code structure and implementation process, as well as the correct execution of V-FIT, this chapter will focus on. Furthermore we will describe how we implemented the cocinelle template files to inject the faults. The last topic, we will explain, are the major challenges during this process.

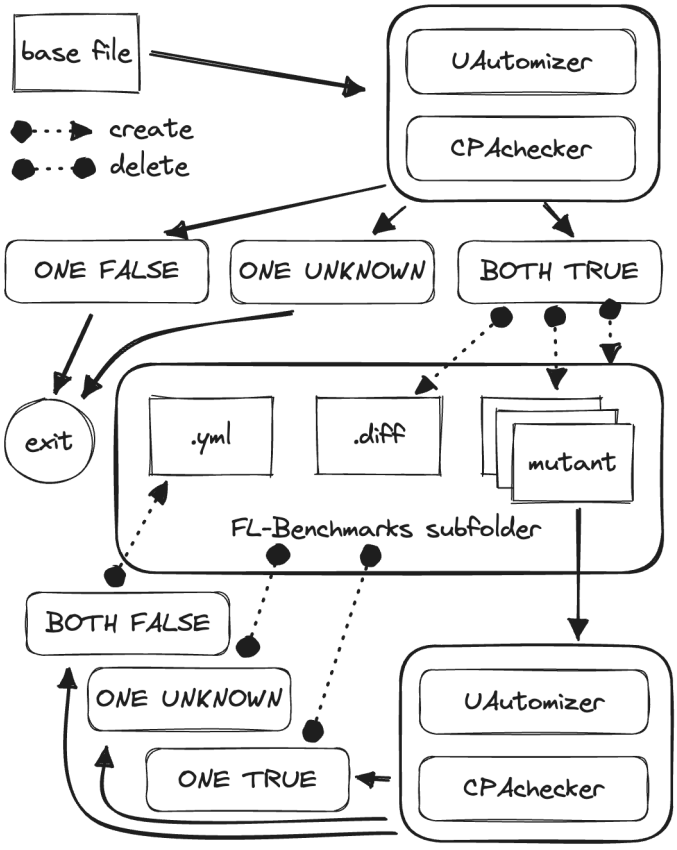

Figure 5: V-FIT main process in detail

5.1 V-FIT Detail Structure

V-FIT is written in Python and has three Python files included. The `vfit.py` file is the main file and is responsible for command line parsing, verifier creation, file transfer and controller tasks. Both next files are included in a `src` folder. First there is `verifier.py`, which contains the template for a Python class called `Verifier`. It receives the execution command for the specified verifier and executes it. The last file is `fl_inject.py`. Its main purpose is the creation of the fault localization benchmark set folder structure, the *mutants*, the `.diff` and the `.yml` file. Furthermore, it deletes the created files and folders, if the verification process failed.
The process starts with command line parsing. The various options we will explain in the next section in detail, because in this section we will focus on the main option available, which one can see in Figure 5. This figure is designed to better understand the following process. We start with the verification of the *base file*, extracted from the SV-COMP benchmark set. CPAchecker is the first verifier the *base file* has to pass. We check the output text of CPAchecker for the verification result, and continue only if the result is *TRUE*, otherwise we exit the program and log the failure. After that, we let UAutomizer verify the *base file*. Again, we only proceed, if the verification result turns out to be true. If this is not the case, we log the failure and exit the program. Now that the *base file* has passed both verifiers, we start creating the fault localization benchmark set folder structure, an excerpt of this structure can be seen on Figure 4. Next we generate the *mutants* by handing over the `.cocci` template and the *base file* to Coccinelle. We store the generated *mutants* in our newly created fault localization benchmark set folder structure and also create a `.diff` file, can be seen at Program 3, in the same folder. The just now stored *mutants* are verified again with the aim of proving the existence of a fault. We iterate over each *mutant* and check it with CPAchecker. If the result is *TRUE*, we delete all generated files and new folders, exit the program and log the failure. If it is *FALSE*, we continue with handing the *mutant* over to UAutomizer. Here is the same case, when UAutomizer verifies a *mutant* to *TRUE*, all generated files and folders are deleted, the program is exited, and the failure logged. The other case, if UAutomizer also verifies the program to be *FALSE*, we generate a `.yml` file, which can be seen in Program 7. This file is also stored in the created folder structure of the fault localization benchmark set, and the process is done.

5.2 Command Line Arguments

Table 1: Survey of the different command line options

argument	**input**	**description**
–**collect**	folder path, output file name	generates a file containing a list of `.yml` files
–**template**	`.cocci` file template	generates mutants
–**data_model**	data mode type	specifies data model type
–**processAll**	file including list of `.yml` files	executes verifiers to check the base files for each `.yml` file in the list. Creates fault localization benchmark set structure and stores generated files in it.
–**yml**	`.yml` file to process	similar to –processAll, but only processing one file
–**c**	`.c` file to process	similar to –yml but getting the *base file* directly as input

In Table 1, we give an overview of the different command line options in V-FIT. The `--collect` option enables us to iterate over the SV-COMP benchmark set and search for `.yml` files that meet our requirements. These are the property *unreach call* exists and the expected verdict flag is set to *true*. As one can see in Table 1 `--collect` takes two arguments, folder path and output file name. The folder path is the parent folder of the given Benchmark set, in our case the `sv-benchmarks` folder displayed in Figure 2 and the output file name, which is variable. V-FIT then iterates recursively through the files and searches for the specified `.yml` files. Afterward it creates a list and writes it to an output file named by the given file name. After each `.yml` file in the output file, a line break is created.
In order to use Coccinelle, the `--template` option offers the opportunity to provide the `.cocci` template for the fault injection.
To specify the data model, we can use the `--data_model` option, either 32 or 64 bit architecture.

The `--processAll`, in combination with the `--template`, flag iterates over the created `.yml` file list and executes the steps mentioned in Section 5.1 for each `.yml` file.
V-FIT provides the `--yml` option to specify a specific `.yml` file to process.
The `--c` argument takes a specific `.c` file to run our tool on.
A combination of `--c`, `--template` and `--data_model` is used to run the tasks for our evaluation.

The following code snippet shows an example run of V-FIT:

```
python3 vfit.py --c example.c --template example.cocci --data_model
<data model type>
```

First we use Python version 3.10 to run our `vfit.py` file. The flag `--c` is used to provide the *base file*, the `--template` flag for the COCCINELLE template and the `--data_model` flag to insert the desired data model type.

5.3 Coccinelle Mutant Templates

For our approach, we created four `.cocci` templates, each for another fault type. The challenge was to create not a single *mutant*, including all faults in the *base file*, as COCCINELLE usually does, but generate *mutants* for every injected fault in the *base file*. Due to this, we created a new template, which creates multiple *mutants*. It is a combination of the `.cocci` template of a given fault type and a `mutate.cocci` template to do the changes one at a time and safe them to different *mutants*. This template was written in Ocaml and in the following, we will explain how we merge these two templates to one. For the sake of simplicity, we call the `.cocci` template for the given fault *file 1*, which one can see in Program 5 and the `mutate.cocci` template *file 2*.
file 2 has three rounds included, marked by surrounded @@'s. In the fist and the second round, we implemented the same, so we just explain on of them. The third round is a bit different because here the actual fault injection takes place. In rounds one and two, we add the metavariables, as one can see in the following code snippet:

```
  28 @r1 depends on !after_start@
  29 position p;
+ 30 statement s1,s2,s3,s4,s5;
  31 @@
```

In our case, the metavariable consists of the five statements. Notice, that the metavariable *position p*, which is already given in *file 2*, is important for the following.
The next step is including our transformation rule from *file 1* to *file 2*. In the following code snippet, we show the added rule:

```
+ 34 (
+ 35 if(...){
+ 36 s1
+ 37 ? s2
+ 38 ? s3
+ 39 ? s4
+ 40 ? s5
+ 41 }@p
+ 42 )
```

Notice, in the lines 35 and 41 we deleted the - from *file 1* and furthermore added *@p* to mark the position of the sematic patch, explained in Chapter 3 after the closing brackets from the if statement.
As already mentioned, we skip the second round, because it it similar to the first.
What we added in the third round can be seen in the next code snippet:

```
+ 34 (
+ 35 -if(...){
+ 36 s1
+ 37 ? s2
+ 38 ? s3
+ 39 ? s4
+ 40 ? s5
+ 41 -}@p
+ 42 )
```

Only in this round did we include the code snippet from *file 1* to *file 2* without changes.

Now the new template file is created and ready to run in V-FIT.

5.4 Challenges

The implementation of V-FIT confronted us with a number of challenges. Our goal was to implement as much as necessary and as little as possible. To combine CPAchecker, UAutomizer, Coccinelle, and the SV-COMP benchmark set in three Python files was quite difficult. The reading and writing from and into files was one of the most tim-consuming tasks. First to get the `.yml` files, fulfilling the requirements of the SV-COMP benchmark set, second to get the correct information out of the `.yml` file, third to use the Coccinelle template properly, and last to generate the fault localization benchmark set structure with the correct name convention.

6 Evaluation

The combination of CPAchecker[1] and UAutomizer[2] with Coccinelle[3] is one major goal of V-FIT[4]. To create the fault localization benchmark set with tasks of the SV-COMP benchmark set[5] is another great success. In this chapter, we look at our results in detail by doing a quantitative and qualitative analysis to examine the performance of V-FIT and the quality of our new benchmark set. To have a look at the results in detail, we provide the raw data in an archive in Zenodo [6], an online repository that enables the exchange of publications and associated supporting data [18].

6.1 Setup

For our approach, we use a subset of tasks from the SV-COMP benchmark set, which we already described in Chapter 3. The subset consists of only tasks with the *unreach call* property.

Setup: The machine is an Intel Xeon E3-1230 v5 @ 3.40 GHz with 8 cores. For our approach we use only 4 cores.

Included are the following task sub-categories from the SV-COMP benchmark set:

- ReachSafety-Arrays
- ReachSafety-BitVectors
- ReachSafety-ControlFlow

[1] `https://github.com/sosy-lab/cpachecker`
[2] `https://github.com/ultimate-pa/ultimate`
[3] `https://github.com/coccinelle/coccinelle`
[4] `https://gitlab.com/sosy-lab/software/fault-injection`
[5] `https://github.com/sosy-lab/sv-benchmarks`

- ReachSafety-ECA
- ReachSafety-Floats
- ReachSafety-Heap
- ReachSafety-Loops
- ReachSafety-ProductLines
- ReachSafety-Recursive
- ReachSafety-Sequentialized
- ReachSafety-XCSP
- ReachSafety-Combinations
- ReachSafety-Hardware
- ConcurrencySafety-Main
- SoftwareSystems-AWS-C-Common-ReachSafety
- SoftwareSystems-BusyBox-ReachSafety
- SoftwareSystems-coreutils-ReachSafety
- SoftwareSystems-DeviceDriversLinux64-ReachSafety
- SoftwareSystems-DeviceDriversLinux64Large-ReachSafety
- SoftwareSystems-uthash-ReachSafety

To produce our fault localization benchmark set, we benefit from BenchExec[6] [5], a framework for reliable benchmarking. This gives us the advantage of defining resource usage, automatically executing commands on large sets of input files and the generation of interactive tables and plots for the results afterward. For the execution of one set we define the limit specifications in a XML file provided by BenchExec.

Time Limit: 1800 seconds (30 minutes)
Memory Limit: 15 GigaByte
CPU Cores: 4

[6] `https://github.com/sosy-lab/benchexec`

We inject four different fault types with Coccinelle, as already described in Chapter 3. For each fault type, we execute our program with a set of 6790 files in total from the tasks mentioned above. We use the V-FIT configuration, explained in detail in Chapter 5.

6.2 Results Overview

In Table 2, we summarize the main results of the processing. Unfortunately, only 36% of the *base file* verifications succeeded. This, of course, shrinks our final benchmark set.
It is quite interesting that overall, 61079 mutants were produced. The great difference from **Mutants produced** and **Mutants verified valid**, is attributable to the tasks that run into timeout, out of memory or a verifcation process failure. Of course, the larger the *base file*, the more *mutants* are produced, but also the time and memory consumption increases. In **Mutants verified valid**, we only count the *mutants* that have a related `.yml` file. Important to notice is that in the **Mutants verified valid** are tasks included, which run into a timeout due to the fact that the timeout could be after creating a bunch of `.yml` files. Because all *mutants* for one task are generated before the related `.yml` file, and then a timeout occurs, neither the related `.yml` files are generated, nor the `mutants` are deleted.
The great number of results in **WVAV**, **Mutants produced**, and **Mutants verified valid** is due to the fact that the fault given by **WVAV**. A wrong value assignment is much more frequent than the other fault types.

Table 2: Results overview

	MIA	**MVAE**	**MVAV**	**WVAV**	**Overall**
Timeout	426	1820	1792	584	17%
Out of memory	195	195	199	201	3%
Done	6169	4775	4799	6005	80%
Base file verified valid	2367	2379	2371	2548	36%
Completely succeeded tasks	340	119	49	180	3%
Mutants produced	2284	3327	3259	52209	61079
Mutants verified valid	321	131	50	356	858

6.3 Quantitative Analysis

In the following section, we do some quantitative analysis based on the results of our processed data. In Figure 6 one can see how much *CPU time* each task required. The *x-axis* represents the time spent in seconds in the range 0 - 1800, until a timeout occurs. The *y-axis* shows the number of tasks processed. We used only **Mutants verified valid** for this graph. Because we run V-FIT for every fault type on the task selection of the SV-COMP benchmark set tasks seperately, we split up the data and sort it into the underlying fault types, **MIA**, **MVAE**, **MVAV**, and **WVAV**.

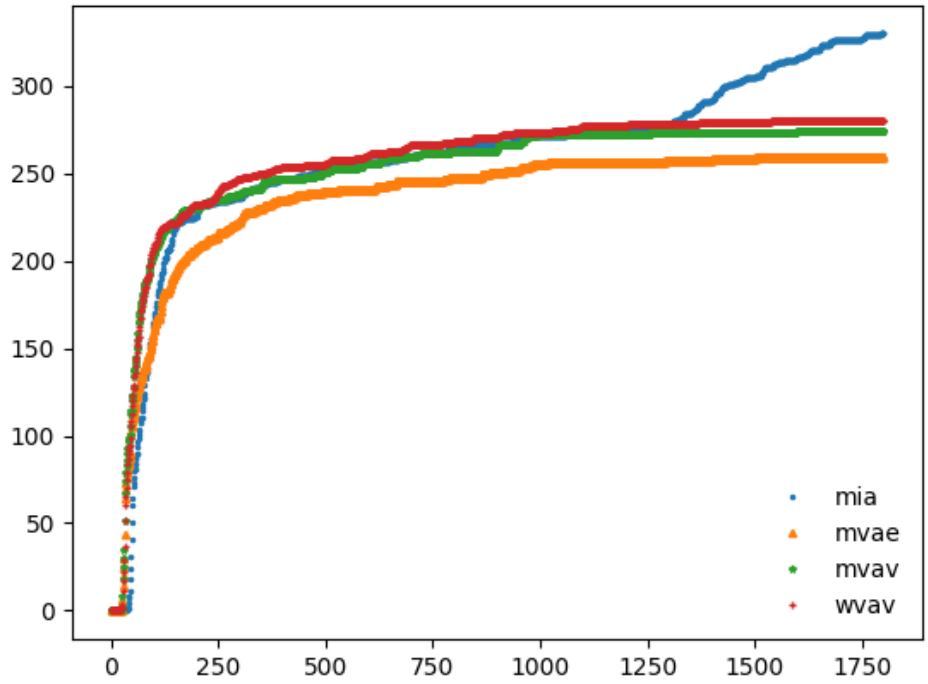

Figure 6: CPU time consumption of successfully completed mutants

Interesting is the fact that most of the tasks only needed under approximately 250 seconds to proceed completely. After that time, about 250 tasks proceeded. This is due to the fact, that most of the tasks produced only one or a few *mutants*. Approximately 100 tasks, so the minor part, take about 250 to 1800 seconds to proceed, because of the vast amount of *mutants* a larger *base files* produces.
Another outcome is that a larger quantity of the **MIA** tasks took 1250 to 1800 seconds to proceed, compared to the other injected fault types. This is because the **MIA** fault is the most complex fault type of the four. Therefore the fault injection consumes more time.

6.4 Qualitative Analysis

In the following section, we want to show the weaknesses and strengths of the new fault localization benchmark set. In order to do this, we pick four

mutants, to show the potential of the new set. For the sake of clarity, we choose the shortest files possible, because there are also files with over 8000 lines of code.

6.4.1 Weaknesses

Program 8: **MVAV** fault type injected to show the weaknesses

```
1  int main()
2  {
3    float x, y, z, r;
4  - x = 1e7f;
5    y = x + 1.f;
6    z = x - 1.f;
7    r = y - z;
8    __VERIFIER_assert(r ==
         2.f);
9    return 0;
10 }
```

Program 9: **MVAE** fault type injected to show the weaknesses

```
1  int main()
2  {
3    double x, y, z, r;
4    x = 1e8;
5    y = x + 1;
6    z = x - 1;
7  - r = y - z;
8    __VERIFIER_assert(r ==
         2.);
9    return 0;
10 }
```

In this subsection, we describe the weaknesses of our approach.
Our goal is to create a Benchmark set for fault localization. We want to simplify the evaluation of fault localization techniques. Therefore, we have the `.diff` files to determine the exact location of the fault and the related fix. But what if there is another solution to the fault possible, such as just reverse the Coccinelle injection. Of course, there is always another possible solution for example, deleting all lines of code would lead to a successful verification of the program. It is important to notice that we want only fault fixes that retain the semantic of the given *base file*. In Program 8, we see one line deleted by Coccinelle. The fault type injected is **MVAV**, because `x` and its value are deleted. Another solution is to just insert the value of `x` for all instances of `x`. So `x` is never used, and the program works again. Another example next to it is Program 9. It shows the deletion of the variable `r` assigned by the expression `y - x`. To not reverse the Coccinelle injection to fix the fault, one could initialize `r = 2.` in line 7, and the fault is also fixed.
This problem shows that our approach performs poorly for some `base files` in providing all fixes possible and therefore the evaluation of fault localization techniques could be difficult, because maybe they provide one correct solution, but not the reverse Coccinelle injection. Because this problem streches almost over the entire set, we provide a solution at Section 6.5.

6.4.2 Strengths

This subsection describes the strengths of our approach.
We provide two examples of the fault localization benchmark set, where the fault injection was successful.
First, there is a example from the robustness analysis of finite precision implementations, by Goubault and Putot [8]. We provide only the necessary lines, so this is not the entire code. As you can see in Program 10, we have two *Doubles*, `S` and `I`, given. They are then calculated in different ways. If `I` is greater than or equal to 2, a different calculation is made, than if `I` is lower than 2.

Program 10: **MIA** fault type injected *mutant* to show the strength

```
1  double sqrt2 = 1.41421353816986083984375 0;
2  int main()
3  {
4    double S, I;
5    I = __VERIFIER_nondet_double();
6    assume_abort_if_not(I >= 1. && I <= 3.);
7  -   if (I >= 2.) S = sqrt2 * (1.+(I/2.- 1.)
       *(.5-0.125*(I/2.-1.)));
8  -   else S = 1.+(I-1.) * (.5+(I-1.) * (-.125+(I-1.)
       *.0625));
9  +   if (I >= 2.) {}
10 +   else {}
11   __VERIFIER_assert(S >= 1. && S <= 2.);
12   return 0;
13 }
```

This example shows perfectly the fault injection with the fault type **MIA**, because the statements in the `if` and the `else` blocks are deleted. This shows, that the fault injection worked and the benchmark is correct.

Program 11: **MVAE** fault type injected*mutant* to show the strength

```
1  int main()
2  {
3    unsigned int i = 0;
4    unsigned int j = 0;
5    unsigned int k = 0;
6    while (k < 0x0fffffff) {
7      i = i + 1;
8      j = j + 2;
9  -   k = k + 3;
10     __VERIFIER_assert(k == (i + j));
```

```
11      }
12  }
```

The next example is a benchmark used to verify Chimdyalwar, Bharti, et al. "VeriAbs: Verification by abstraction (competition contribution)." from the International Conference on Tools and Algorithms for the Construction and Analysis of Systems [12], one can see in Program 11. In this example, one can see the fault injection of the fault type **MVAE**. In line nine, one can see the deletion of the variable `k` together with its initialization. The fault was injected perfectly because `k` is used in the `__VERIFIER_assert()` method, but after fault injection, `k` is neither declared, nor initialized.
These examples show that the fault injection works perfectly, and with the given `.diff` file, there is a way to determine the exact position of the fault. Furthermore, there is a solution for the fault given by the reverse Coccinelle injection.

6.5 Future Work

With our approach, we created only the base for additional research, and there are many possibilities for proceeding further. In this chapter, we want to present ideas for further research.

6.5.1 Increase Verifiers

At SV-COMP 2023 alone, 52 verifiers participated. For V-FIT, we only use CPAchecker and UAutomizer so far for the verification task before and after the fault injection. This could be changed by adding more verifiers to V-FIT. The implementation is very simple if the verifiers have a command line interface. Then only the code structure has to be added to `vfit.py` and also the correct parameters have to be provided. This extension would not only increase the quality of verification, but even the quality of the entire fault localization benchmark set. As a side effect, the additional verifiers would be tested for correctness.

6.5.2 Raise templates

Moreover, additional Coccinelle templates could be created. So far, we use the fault types **MIA**, **MVAE**, **MVAV**, and **WVAV** as inspiration for the Coccinelle templates. As described in Chapter 3, there exist eight

other fault types, which could be implemented as a Coccinelle template. Of course, one can use their own fault types and implement them.
The `mutant.cocci` file, as mentioned in Chapter 5, could be automated by a Python file, so one can insert the preferred template into this file and get the generated `mutant.cocci` template. This approach would increase the amount of results and the diversity in the fault localization benchmark set.

6.5.3 Extend Benchmark Set

In our approach, we use the SV-COMP benchmark set and generate the fault localization benchmark set from it. The fault localization benchmark set could be extended by taking another Benchmark set, assuming that this set also contains files written in the C programming language. And then running V-FIT over it to generate more fault localization benchmarks. This would increase the quantity and, moreover, the diversity of the FL-Benchmark set.

6.5.4 Insert Programming Languages

Until now, all *base files* we hand over to V-FIT are written in the C programming language. At the beginning of this work, we did not know about Coccinelle and thought about injecting the fault on our own. Fortunately Coccinelle also supports fault injection for other languages, so it could be used for fault injection in the programming language *Java*, for example. To implement this tool in V-FIT would greatly extend the fault localization benchmark set. The folder structure in the fault localization benchmark set is already given, as one can see in Figure 4.

6.5.5 Synchronize Benchmark sets

Another opportunity is, to update our fault localization benchmark set immediately, if there is a new contribution to the SV-COMP benchmark set, of course, V-FIT has to run again to update the fault localization benchmarks as well. This could be automated by synchronizing the two Benchmark sets. The advantage is, that the fault localization benchmark set grows together with the SV-COMP benchmark set.

6.5.6 Provide Open Source

If V-FIT and the FL-Benchmark set would be putted open-source at the free disposal, other research groups and developers may benefit from this proposal. Furthermore the SV-COMP benchmark set is also open-source. This enhancement would also improve the reputation of V-FIT as well as from the fault localization benchmark set.

6.5.7 Enable Contribution

In addition to the open-source provision approach, it could be enabled to contribute to the FL-Benchmark set in the form of adding other fault fixing solutions. This would increase the number of possible solutions to a fault and therefore contribute to solving the problem mentioned in Section 6.4.

6.5.8 Improve Fault Localization

Previously, our approach only provides the `.diff` and the *mutant* files for fault localization. V-FIT could be combined with fault localization techniques to improve the quality.

7 Conclusion

Fault localization techniques are difficult to evaluate because of the lack of benchmark sets, which provide benchmarks including faulty programs with the fault location given. In this approach, we presented V-FIT, or Verified Fault Injection Tool. Using this tool, we verified tasks of the given SV-COMP benchmark set using CPAchecker and UAutomizer, we injected four different types of faults using Coccinelle and verified the output again. Thereby, we created the FL-Benchmark set, including the folder structure, inspired by the SV-Benchmark set, a *mutant* file for each fault injected, as well as a `.diff` file for the exact fault location and a `.yml` file for metadata. To produce our results, we ran V-FIT on BenchExec with 6790 sub tasks of the SV-COMP benchmark set for each of the four fault types and set a time limit of 1800 seconds per task. Our evaluation shows the benefits and drawbacks of our tool, we can only successfully inject faults in 3% of the tasks yet and 858 mutants were produced. Although this might seem little, it shows the potential. Further improvements and adjusted time limits will increase the numbers. We also investigated the strengths and weaknesses. We suffer from an old problem that it is not easy to decide what a real bug fix is. In the future, we allow crowd sourcing on our benchmarks to make them more robust and less prone to biases. The benchmarks set should grow dynamically and with the help of the community. We could also find strengths in our approach. The advantage of our work is that the fault injection works perfectly with four different fault types, so we produced a diverse new benchmark set. Furthermore, we could determine the exact location of the fault and provide a solution to fix it.
Our approach created a solid basis for further work, because now people can build on the benchmark set. This is the first large benchmark set for C language that is community based.

Bibliography

[1] D. Beyer. Competition on software verification and witness validation: Sv-comp 2023. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 495–522. Springer, 2023.

[2] D. Beyer. Software testing: 5th comparative evaluation: Test-comp 2023. *Fundamental Approaches to Software Engineering LNCS 13991*, page 309, 2023.

[3] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings 19*, pages 504–518. Springer, 2007.

[4] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 184–190. Springer, 2011.

[5] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21:1–29, 2019.

[6] M. Bierwirth. Designing and Assessing a Benchmark Set for Fault Localization Using Fault Injection, Aug. 2023. Available at `https://doi.org/10.5281/zenodo.8211917`.

[7] J. A. Duraes and H. S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

[8] E. Goubault and S. Putot. Robustness analysis of finite precision implementations. In *Asian Symposium on Programming Languages and Systems*, pages 50–57. Springer, 2013.

[9] M. Heizmann, Y.-F. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, et al. Ultimate automizer and the search for perfect interpolants: (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II 24*, pages 447–451. Springer, 2018.

[10] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[11] J. Lawall and G. Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 601–614, Boston, MA, July 2018. USENIX Association.

[12] A. Legay and T. Margaria. Tools and algorithms for the construction and analysis of systems: a special issue for tacas 2017. *International Journal on Software Tools for Technology Transfer*, 24(4):611–612, 2022.

[13] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, 2012.

[14] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*, pages 10–es, 2006.

[15] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.

[16] C. Richter and H. Wehrheim. Pesco: Predicting sequential combinations of verifiers: (competition contribution). In *Tools and Algorithms*

for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III 25, pages 229–233. Springer, 2019.

[17] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury. Localizing vulnerabilities statistically from one exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 537–549, 2021.

[18] M.-A. Sicilia, E. García-Barriocanal, and S. Sánchez-Alonso. Community curation in open dataset repositories: Insights from zenodo. *Procedia Computer Science*, 106:54–60, 2017.

[19] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

03.08.2023	
Datum	Moritz Bierwirth