IS483 Final Report

KLEE Automated Conversion Tool and Analysis

KLEENOTATION()

Cybersecurity Track

Date: 29th April 2021

Team Members:

- Aldric Chong Kai Yuan (aldricchong.2018@sis.smu.edu.sg) – Project Manager
- Adrian Chang Fu Ren (adrianchang.2017@sis.smu.edu.sg) – Lead Back-End Developer
- Ngo Thanh Van (tvngo.2018@sis.smu.edu.sg) – Quality Assurance
- Qi Xiang (xiang.qi.2017@sis.smu.edu.sg) – Deputy Back-End Developer
- Loh Xiao Binn (xbloh.2018@sis.smu.edu.sg) – Vulnerability Analyst

Faculty Supervisor:

Singapore Management University, School of Computing and Information Systems

- Prof. Wang Yong

Project Sponsor:

Singapore Management University, School of Computing and Information Systems

Prof. Jiang Lingxiao (lxjiang@smu.edu.sg) - Full-time Faculty

Executive Summary

This report will cover the research undertaken in the process of automating the process of bug handling by KLEE, with an additional feature in analysing if the bugs identified could be a potential vulnerability.

KLEE is identified as an all-rounded tool used for testing of possible bugs within a programme, as KLEE is able to formulate its own test cases, it removes the hassle from the user in creating executables to test the programme. However, there are limitations to KLEE as users will need to accurately identify the right lines within the code to make the desired variable symbolic for KLEE to run the analysis. This process is both time consuming and prone to human error, as with more complex codes that stretch across multiple files and variables, users will need to go through one by one and to insert the necessary KLEE commands into the code. Therefore our team aims to smoothen this process through automation for the user, where the user will only require to input the necessary variable he/she desires to make symbolic. Our code will identify the necessary lines and run KLEE on their behalf.

Our programme is broken down into four different modules: parser, annotate, compile, analysis. This modular design allows for scalability of our programme as any additional feature needed can be created as another module. Moreover, we are able to conduct component bug fixing for each module without having to change the entire code for the whole programme.

The parser module is in charge of handling the C input file using the Python library pycparser, by searching through the Abstract Syntax Tree (AST) to identify the line where the variable appeared. This is then passed to the annotate module where it inserts the KLEE library header and the make symbolic command into the file. The compile module will compile the programme into a bitcode (.bc) file and pass it through KLEE for it to conduct the checks. Finally the analysis module will analyse the bug against the bug metric and notify the user if any possible vulnerabilities are present. For the scope of our project we will be focusing on buffer overflow vulnerabilities for the codes we analyse, as aforementioned, our code is scalable for future uses by adding more modules for other vulnerabilities.

Our project was able to significantly reduce the time taken in conducting KLEE analysis from the initial 14 minutes to 9 minutes on average with our graphic user interface. Through our 2 user testings we are able to gather a 98% user satisfaction of our programme. KLEENOTATION() has successfully automated the current process and allowed for the ease of using KLEE symbolic analysis for our users.

Project Overview

Problem Statement

KLEE is a versatile open-source tool that allows for the automated generation of test cases and symbolic execution of a piece of code so that bugs within it can be found. However, in its current implementation, a **limitation** is that it requires many manual pre-processing steps to analyse a piece of code. This problem is further intensified when multiple pieces of code are required to be analysed at a go.

Project Motivation

Our sponsor is in the process of performing bug research using KLEE, but the lack of automation slows down his progress, reducing his efficiency. Thus, this project has been initiated in order to fix this problem.

The **overall benefits** that our project brings is twofold. First, KLEENOTATION() functions as a program transformation tool that enables the users to **speed up** KLEE and their research on bug/vulnerability identification. Second, our tool generates additional analysis based on the outputs of KLEE that enables our users to gain **useful insights** which can further aid in their research.

Stakeholders

The stakeholders for this project are as follows:

Sponsor	This project was initiated by Prof Jiang Lingxiao, the teaching staff for Enterprise Solution Development module who taught half of our members. He is a Full-time faculty member in Singapore Management University, School of Computing and Information Systems.
Users of KLEE	Our primary user is the project sponsor, but KLEENOTATION() can also be used by any other users who wants to perform bug analysis using KLEE and would like to speed up their processes.

Deliverables

At the end of the project, our group is to deliver a command line tool written in Python that automatically converts any C program into bitcode (.bc) files with the required KLEE statements being annotated and passes them on to KLEE for bug checking. This tool will then be released through Github. This is the main requirement of the project form the project sponsor. In addition, as Cybersecurity majors, we also propose that an analysis on the outputs from KLEE will also be delivered, pointing to the user's potential security issues with their code.

Scope

The project scope will encompass developing and testing a command line tool that will prepare files for analysis by KLEE as well as performing an analysis on the output of KLEE. The tool will be written with Python and integrated with KLEE. Options will be given to the user to specify variables that they wish to annotate and be handled as symbolic by KLEE.

To prepare files for analysis, we will convert variables selected by the user to symbolic by inserting the relevant KLEE C code lines into the target C code. This will be done via parsing and replacing text. Third party libraries and tools will be used such as pycparser, a C parser for Python (The full list of the tools used can be found in the Solution Overview section of the report). This is also the first constraint that we have identified as the users of KLEENOTATION() must have the required dependencies installed in their setup to run our tool. However, this is made easy for potential users by utilising Docker and generating the Docker image of KLEENOTATION() with all dependencies already installed which the user can then run a Docker container of at any time. Another important constraint of the project identified is that the team will be treating KLEE as a black box and will only be invoking its functionalities in order to fulfil the needs of the project.

To perform analysis on KLEE’s output, our code will look through the bugs and errors generated by KLEE to analyse the possibility of it being an exploitable vulnerability. Since we are running our analysis on the DARPA CGC (Defense Advanced Research Projects Agency Cyber Grand Challenge) repository, we will compare the known bugs with the bugs identified by KLEE to pick out any possible discrepancies for our analysis. The scope of bugs that is of concern to this project is narrowed down to buffer overflows.

All coding of KLEENOTATION() is done on GNU nano while version control is done using Github. Our development environment will be on Windows 10, with KLEE running on a Docker container. Our system design approach will follow the Rapid application development approach.

What is KLEE

First introduced in 2005 and originally designed at Stanford University, KLEE is a well known dynamic symbolic execution engine that provides the capability to automatically explore the different paths that can be taken within a program (Cadar & Nowack, 2020). This is done with the concept of using a constraint solver to determine path feasibility. Today, KLEE has more than 70 contributors on its Github page and the tool has been used and extended in over 150 publications making it one of the more prominent open-source symbolic execution tools that available.

Symbolic Execution

According to Carnegie Mellon University, the process of symbolic execution is a way of executing a program abstractly, so that one abstract execution covers multiple possible inputs of the program that share a particular execution path through the code (Aldrich & Le Goues, 2018). Through this method of execution, one will be able to determine what inputs cause each part of a program to execute which can aid in finding bugs within a program.

Symbolic Execution allows the user to explore multiple paths of the program to ensure greater coverage. A program with high test coverage has had more of its source code executed during testing. This means it has a lower chance of containing undetected software bugs compared to a program with low test coverage. Thus, it is useful to detect bugs in software with greater accuracy.

DARPA Repository

In 2016 the Cyber Grand Challenge was launched by DARPA with the goal to bring about new cyber defense capabilities through automation[1]. As a part of the challenge, a repository of vulnerable codes were released by them meant to be tested on by teams of researchers and experts in the field with their autonomous cyber defence systems.

[1] https://www.ll.mit.edu/r-d/datasets/cyber-grand-challenge-datasets

Data Analysis

Data Utility

In KLEENOTATION(), data is used to determine a program's susceptibility to attacks and possibility of vulnerabilities based on its bugs and errors generated from KLEE. The type of vulnerability that is focused on in this project is limited to buffer overflow attacks as defined in our project scope.

Data used in Error Analysis and Vulnerability Categorisation

Data used in the project come from mainly 2 sources. The first comes from the DARPA repository[2] where within each DARPA program in the repository, the vulnerability details of the program is clearly identified in their respective `README.md` file. Through this, we are able to extract the generic class of vulnerability as well as the CWE classification of each DARPA program.

The second data source comes from SMU PhD Student Tu Haoxin who was able to run KLEE on each of the DARPA repository programs and generate the respective KLEE errors that are expected into a github post[3].

Combining these 2 data sources, our team is able to generate a risk analysis based on the type of vulnerability as well as the type of KLEE errors that are expected of it. Through this, KLEENOTATION() is able to apply a probability chart to assess and highlight to the user the possible vulnerability of a piece of C code based on future errors codes produced by KLEE when running on a target C program.

[2] https://github.com/CyberGrandChallenge/
[3] https://gist.github.com/Hanseltu/51a1fa263d3d3990ab6fa7c887847827

Solution Overview

Solution Tools Definition and Purpose

The list of tools and libraries that are used in this project are listed in the table below:

Docker	Docker is a virtualization tool that is used to deploy applications within containers via Docker Images. It is used in our project to run KLEE and is also used to deliver our final product with all the required dependencies installed.
Glob	Glob is a Python module that allows for traversing and exploring of file paths used by our program.
Python	Python is the main programming language that is used in our project. Python version 3.8 was used in this project.
Pycparser	Pycparser is a Python library and a complete parser of the C language. Written in pure Python using the PLY parsing library. It parses C codes used in our project into an abstract syntax tree (AST) for line number analysis.
JSON	JSON is a data transmittable in human readable format. Used for stating the necessary input variables from the user to be handled by the programme.
Clang	Clang is a compiler front-end for C language, used by the programme to compile the c files into bitcode (.bc) files which is then passed through KLEE.
OS Path	OS Path is a path module used for Python. It is used to identify the necessary relative and absolute PATH for our input files and libraries.

Subprocess	Subprocess is a Python module used to run Linux executable commands. It is used to automate the process flow passing command from modules to modules.
Tkinter	Tkinter is a GUI toolkit used in our project's UI.
Time	Time is a Python module that is used in our program to space out the execution of lines in our code to ensure that there is sufficient time for a process to end before moving on to the next step.
wllvm	wllvm is a tool that is used by our project to generate llvm bitcode files (.bc) from C codes.
llvm-link	Part of the llvm project, llvm-link is used in our project to combine multiple bitcode files (.bc) into a single bitcode file that is passed into KLEE for processing.
cb-multios	cb-multios is a Github project with a set of tools and libraries that aid in the compilation of DARPA programs.
GNU nano	The main text-editor used to develop our codes in the Linux environment.
termcolor	A Python module that allows for coloured text output in the terminal. Used in printing status and error messages in our code.
kcachegrind	A tool used in our program's Analysis module. Kcachegrind is a data visualization tool that can help KLEE users visualise the execution of their target codes.

Architecture Diagram

The architecture diagram of KLEENOTATION() is as shown in the image below:

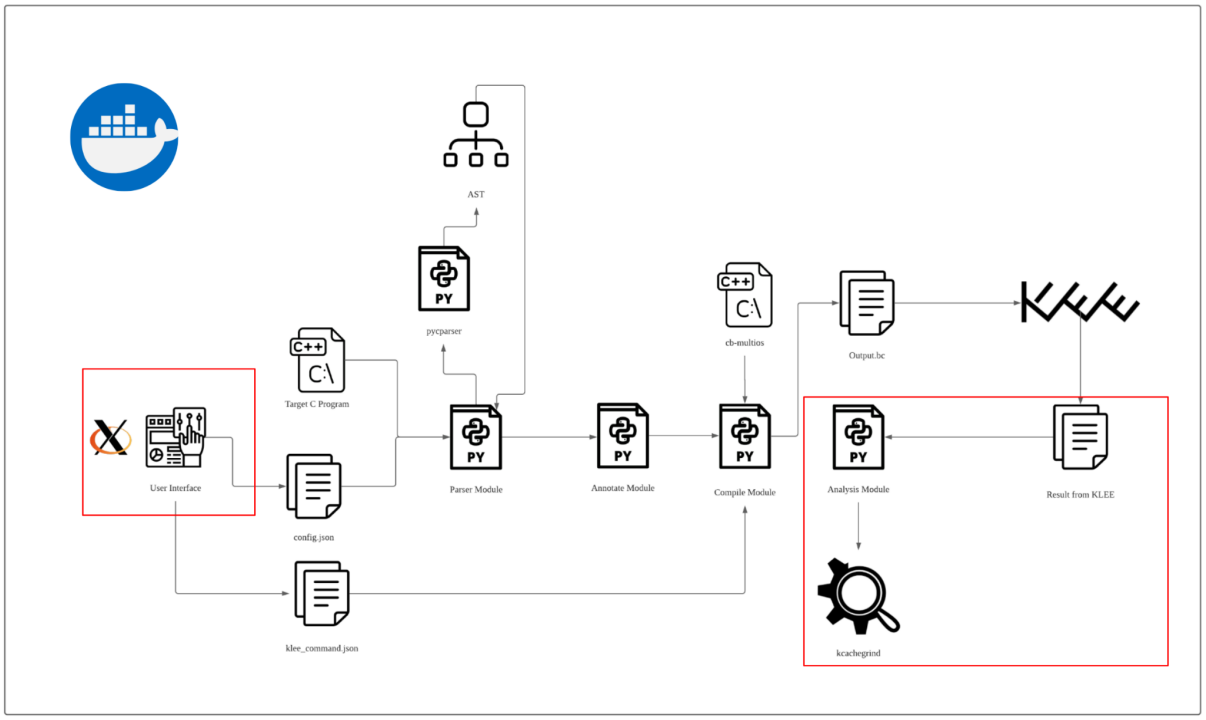
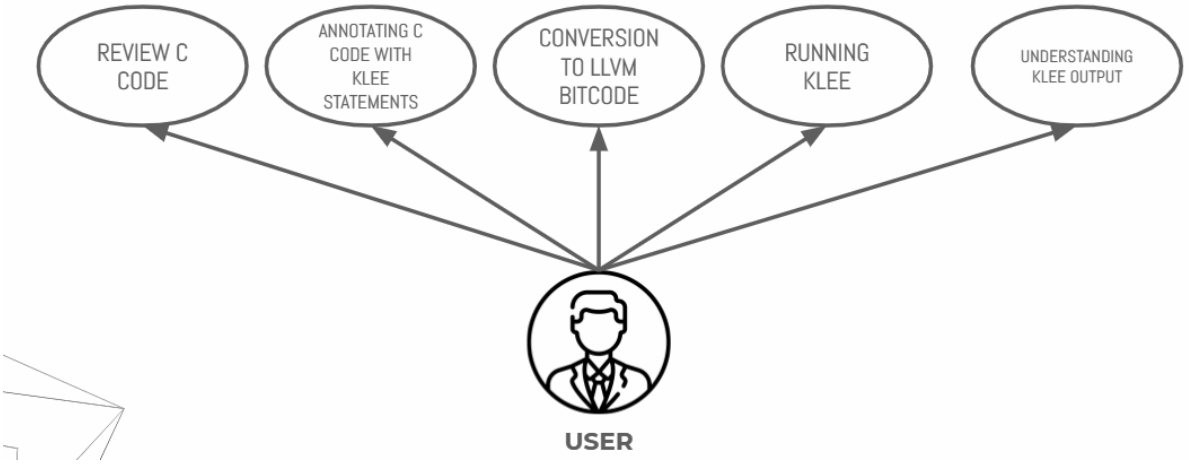


Figure 1.0: Architecture diagram of KLEENOTATION()

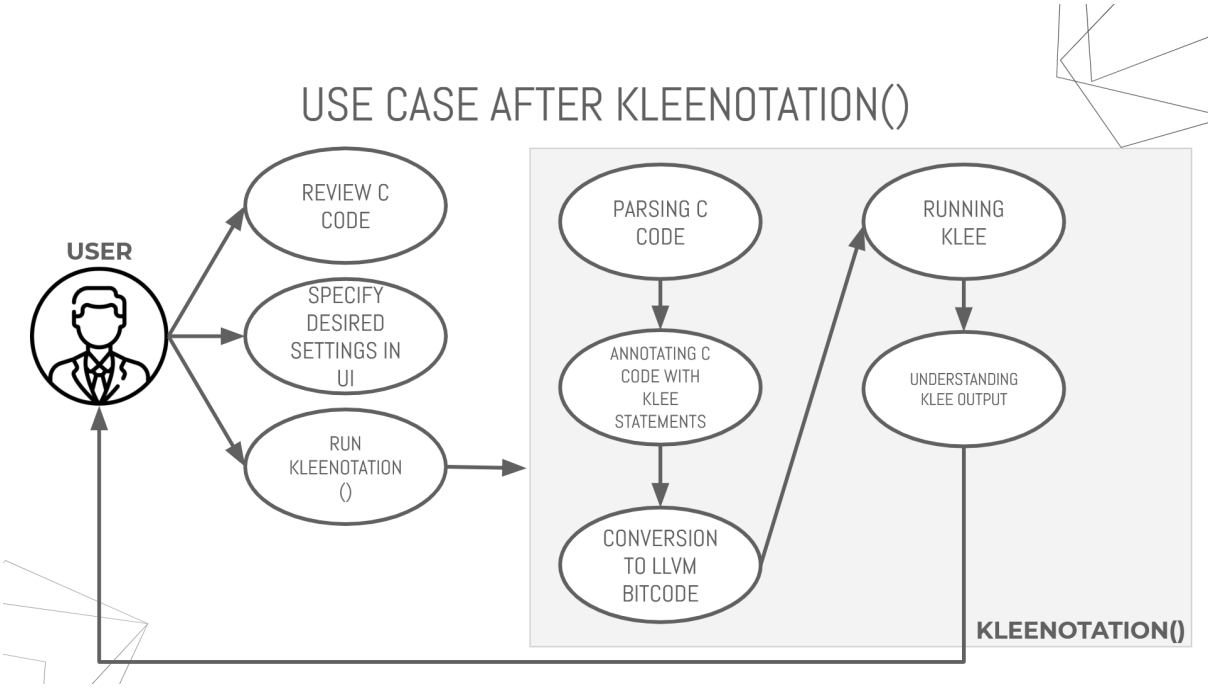
Use Case Diagram

Our team used this diagram before coding to ensure that our program fits the use case required by the sponsor.



Use Case After KLEENOTATION

Our team made use of this diagram to envision what we wanted to code when we wrote the program.



Functional Requirements

Inputs Required by each Module

The tables below explain in detail the required inputs in order for each of our developed modules of KLEENOTATION to function as well as the outputs that they generate.

Parser Module

Input	Output

`config.json` - a json file that specifies the location of the C source code, the functions of interest and the variables within the functions which we .
 	
Sample:
`{`
`"home/klee/POC/test.c":`
`[`
`{"main": ["number1", "number2"]},`
`{"func1": ["var1"]}`
`]`
`,`
`"test1.c":`
`[`
`{"main" : ["var1","var2"]},`
`{"func1": ["var3"]}`
`]`
`}`
	A list of lists where within each list, the filename, function_name, variable_name, and array_status, and line number of the variable is specified.
	
	Sample:
	
	`[ [home/klee/POC/test.c, main, number1 True, 10], [home/klee/POC/test.c, main, number2, True, 16] ]`

Annotate Module

Input	Output
Multiple strings values according to the filename, variable_name, array_status, and line number of interest based on list produced by the Parser Module **Sample:** `annotate(home/klee/POC/test.c, number1, True, 10)`	The annotated version of the target C source file. **Sample:** Before `test.c` After `test_annotated.c`

Compile Module

Input	Output

klee _command.json - a json file that defines the klee flags to be run with each program.
 	
Sample:
`{`
`"home/klee/POC/test.c":`
`[`
`{"klee_flags" :`
`["--external-calls=all"]}`
`]`
`}`
	The bitcode (.bc) version of the target C source files as well as the KLEE report stored in a directory of the target C source files under the folder “klee-last”.
	
	Sample:
	Before
	`test_annotated.c`
	
	After
	`test_annotated.bc`
	`(folder with KLEE report created)`
	`POC/klee-last`

Analysis Module

Input	Output

Requests a “klee-last” directory path from the user.
 	
Sample:
Command line input:
“`/home/klee/POC/klee-last`”
	An analysis of the results generated by KLEE display on the console.
	
	Sample:
	`[!] In testoob.c, memory error: out of bound pointer has occurred on line 13. Please refer to later analysis to determine if the error is dangerous.`
	`[!] In testoob.c, external call with symbolic argument: printf has occurred on line 13. Please refer to later analysis to determine if the error is dangerous.`
	`[!] In testoob.c, memory error: out of bound pointer has occurred on line 16. Please refer to later analysis to determine if the error is dangerous.`
	`======== Analysis ========`
	`[!] In file testoob.c, the`

	following errors have occurred: memory error: out of bound pointer: Appears 1 times. [!] There is a low probability that there is a Buffer Overflow Vulnerability. Impact: Significant [i] Based on DARPA repository code analysis, out of 134 memory errors, there were 43 Buffer Overflows (Occurence rate: 32.09%) [i] Remediation: Strategy: Environment Hardening (CWE-120): When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs. Strategy: Compilation or Build Hardening (CWE-121): Run or compile the software using features or extensions that automatically provide a protection mechanism that mitigates or eliminates buffer overflows. For example, certain compilers and extensions provide automatic buffer overflow detection mechanisms that are built into the compiled code. Examples include the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, and StackGuard. [i] Would you like to run kcachegrind on the latest klee output in klee-last? (Y/n)

Solution Architecture

Technical Setup (Fresh Installation on KLEE 2.1 Docker Image)

- Clone pycparser to `/home/klee`[4]
- Clone cb-multios to `/home/klee`[5]
    - This contains the DARPA repository codes
- Run `sudo apt-get update`
- `sudo apt install`
    - `Python3`
    - `nano`
    - `gcc`
    - `kcachegrind`
    - `python3-tk`
- `pip3 install`
    - `pycparser`
    - `termcolor`
    - `glob2`

Graphic User Interface Setup

Step 1: Install and start Xming
Step 2: Find the IP address of the above program. Example: `192.168.56.1`
Step 3: Ensure that you have Python 3 installed
Step 4: run the following commands

a) This command may be run in any directory.

```
export DISPLAY=192.168.56.1:0.0
```

b) This command is to run the GUI Python file using Python 3. You must run this in the GUI directory or with the appropriate path. In this example, we are already in the GUI directory.

```
python3 gui.py
```

Step 5: If successful, the GUI window should appear like so:

[4] https://github.com/eliben/pycparser
[5] https://github.com/trailofbits/cb-multios

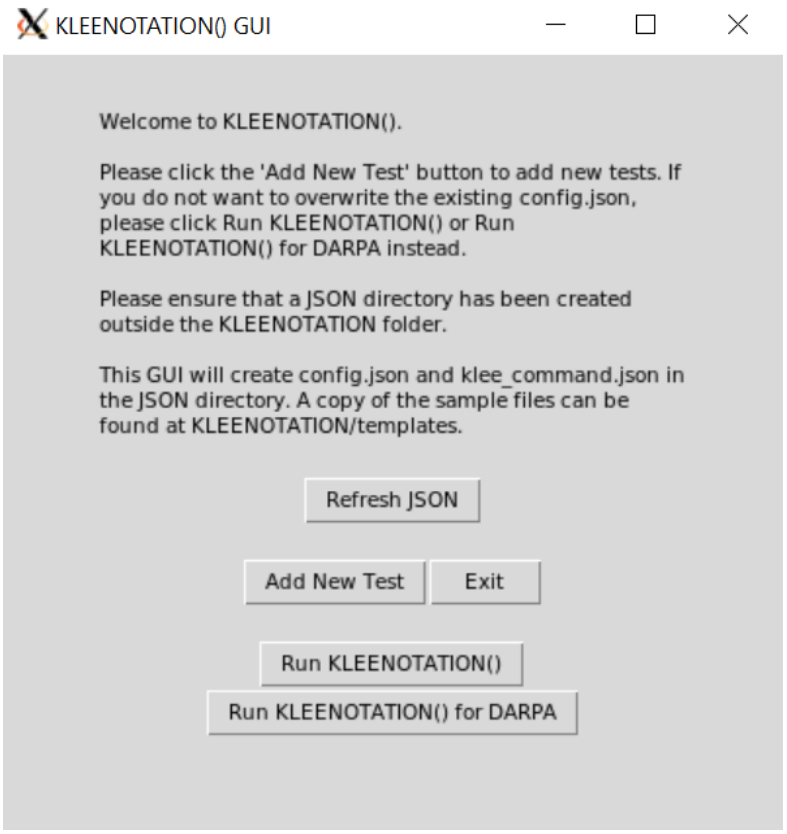


Figure 1.1: Screenshot of main window

Algorithm

Parser Module

Our parser makes use of `pycparser` to parse Python files. `pycparser` is a parser for the C language, written in pure Python. It is a module designed to be easily integrated into applications that need to parse C source code.

We chose `pycparser` because it has no external dependencies (except for a Python interpreter), making it very simple to install and deploy[6].

Given a file, a function name, and a variable name, which are specified in a JSON config file, our parser returns the line where the function is first declared in the given function.

[6] https://github.com/eliben/pycparser

```
{
"CROMU_00016/src/generator.c":
        [
        {"PoissonGenerator":["delta"]}
        ]
}
```

Figure 2.0: Extract of JSON config file

Here we see an example of the json config file. We aim to run KLEE on the file `generator.c`. We aim to make symbolic the variable named delta, and that variable is located in the function `PoissonGenerator`.

We do this by parsing the json file mentioned above.

```
#Import data from JSON file
JSON_PATH = "/home/klee/JSON/config.json"

with open(JSON_PATH) as f:
  data = json.load(f)
```

Figure 2.1: Extract of parser.py

In order to locate the line where the variable is declared, we first create an Abstract Syntax Tree (AST) representation of the file the user specifies using pycparser.

Abstract Syntax Tree (AST)

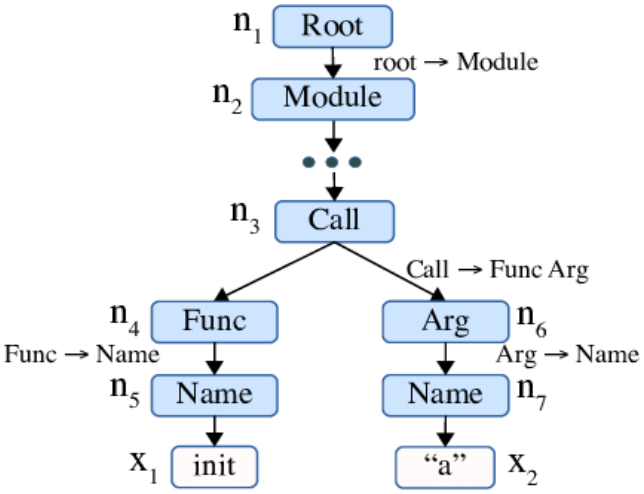


Figure 2.2: Sample diagram of AST

In computer science, an abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code (Kundel, 2020).

Using pycparser's `parse_file()` and `show()` method, we obtained the AST in text format as shown[7]:

```
FuncDef:  (at test.c:3:5)
  Decl: main, [], [], [] (at test.c:3:5)
    FuncDecl:  (at test.c:3:5)
      TypeDecl: main, [] (at test.c:3:5)
        IdentifierType: ['int'] (at test.c:3:1)
  Compound:  (at test.c:3:1)
    Decl: number1, [], [], [] (at test.c:5:9)
      TypeDecl: number1, [] (at test.c:5:9)
        IdentifierType: ['int'] (at test.c:5:5)
    Decl: number2, [], [], [] (at test.c:5:18)
      TypeDecl: number2, [] (at test.c:5:18)
        IdentifierType: ['int'] (at test.c:5:5)
```

Figure 2.3: Output of parser.py

We then performed text search on the textual format of the AST, in order to determine the line number where the variable was declared.

```
TypeDecl: number2, [] (at test.c:5:18)
```

Figure 2.4: Example of variable identified

Once we find the line where the variable is declared, we then extract the line number. In the example above, we are able to determine that the variable named `number2` is declared at line 5.

Fake Headers

Due to the way pycparser works, we also have to handle headers and insert the necessary fake headers if needed. This is to fix dependency issues when attempting to parse C Programs[8].

```c
#include <someheader.h>

int foo() {
    // my code
}
```

Figure 2.5: Example of fake headers to include

[7] https://github.com/eliben/pycparser/blob/master/pycparser/c_ast.py
[8] https://github.com/eliben/pycparser#32what-about-the-standard-c-library-headers

In the example above, `pycparser` needs to know what `someheader.h` contains so it can properly parse the code. As pycparser needs only a very small subset of `someheader.h` to perform its task. The idea of fake headers is simple. Instead of actually parsing `someheader.h` and all the other headers it transitively includes (which probably includes a lot of system and standard library headers), we create a "fake" `someheader.h` that only contains the parts of the original that are necessary for parsing; in other words, the `#define`s and the `typedef`s.

To summarise, `pycparser` does not care about the semantics of types. It only needs to know whether some token encountered in the source is a previously defined type.

Pycparser comes with pre-install fake headers (at `utils/fake_libc_include`), but we have to add to the folder for every new header that is not included. This is why we automate this step before parsing.

This is done by detecting the headers in the program through text search.

```
for line in lines:
    if "#include" in line:
        #print("#include found at line:" + line)
        if "<" in line:
            header = line.split("<")[1].strip()[:-1]
            header_files.append(header)
        elif '"' in line:
            header = line.split('"')[1].strip()
            header_files.append(header)
        elif "'" in line:
            header = line.split("'")[1].strip()
            header_files.append(header)
```

Figure 2.6: Screenshot of headers detected by pycparser

Once we find the headers, we check it against a set list of pre-installed headers included with pycparser.

If the header is not in the pre-installed headers, we create a file and write the preset fake header code in it.

```
for header in header_files:
    if header not in existing_h:
    # create the fake headers
        #print(os.getcwd())
        with open(os.path.join(filepath, header), "w") as f:
            #print(os.getcwd())
            to_write = '#include "_fake_defines.h"\n#include "_fake_typedefs.h"\n'
            f.write(to_write)
```

Figure 2.7: Error thrown when header is not found

Commenting out Unnecessary Code

Additionally, the program also comments out occurrences of `scanf()` and `gets()`. This is because the version of KLEE included from Docker does not support it, and throws errors. Upon consultation with our sponsor, we determined that it is more beneficial to user to comment it out. This is because the functionality of KLEE will not be majorly affected by not having `scanf()` included. As such, they are false errors and the user experience will be improved by removing the unnecessary code.

```
## Comment out SCANF & gets ##
content = []
with open(filename, "r") as f:
    lines = f.readlines()
    for line in lines:
        if "scanf(" in line and "/*" not in line:
            content.append("/*" + line + "*/")
        if "gets(" in line and "/*" not in line:
            content.append("/*" + line + "*/")
        else:
            content.append(line)
```

Figure 2.8: Screenshot of parser.py commenting out external function call

Upon finding the line number of the given variables, control is passed to another program, called Annotate, via importing and calling it with the required parameters. We will outline the Annotate module in the next section.

Annotate Module

The main purpose of the Annotate module is to add the required KLEE code to the given C file, before it is compiled for analysis by KLEE.

To elaborate, let us use an example from the KLEE website:

```
int get_sign(int x) {
  if (x == 0)
    return 0;

  if (x < 0)
    return -1;
  else
    return 1;
}
```

Figure 3.0: Extract of example from KLEE tutorial

In this example, if we were to convert the program to symbolic, we would need to insert code as such:

```
int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

Figure 3.1: Extract of code to insert from KLEE tutorial

As observed, the user would need to manually find the line where the variable a is declared, and insert a `klee_make_symbolic()` line before that variable is used. This is automated by our program, to make things faster and simpler for the user.

Additionally, due to the way the `&` (address-of) operator works in C, the user would need to check if the variable in question is stored as a memory location. For example, if the variable is an array, the user would need to omit the `&` operator. This is because `klee_make_symbolic()` accepts an address, and since arrays point to the address location in C, using the address-of operator on address will cause issues. For ease of use, our program also automates this step for the user, since we have already detected whether the variable is an array or not earlier.

We would also need to insert an `include` statement at the beginning of the file, in order to allow the usage of `klee_make_symbolic()`. This is to ensure that the program loads the necessary KLEE C Header files required.

Include statement to be inserted:
`#include "/home/klee/klee_src/include/klee/klee.h"`

The Annotation module automates all of these steps. This saves the user time, both in including the include statement, as well as locating the variable, finding out where to include the klee statement, and finding out whether to use the address-of operator. The annotate module writes to a file named `filename_annotated.c`.

Compile Module

The compile module is called once our annotate module has completely finished annotating the selected C files. 2 different methods are then deployed to compile the C program into their binary files. These 2 methods are split based on whether the codes are from the DARPA repository or stand-alone non-DARPA codes.

For non-DARPA codes:

Upon startup, the module would traverse to where the C program is stored and run clang on each of the C files in the target directory.

Full clang command:

```
clang -I/home/klee/klee_src/include -emit-llvm -c -g -O0 -Xclang
-disable-O0-optnone -I./ (filename)
```

This would create the bitcode (`.bc` file) of the C program that is ready for KLEE processing.

For DARPA codes:

Upon startup, the module first runs the "`make`" command for the relevant DARPA program selected by the user through `build.sh` (from `cb-multios` project[9]). The binary files of the C programs will then be generated. Next, we turn these binary codes into bitcode (`.bc` file) for KLEE processing. This is done through the extract-bc tool invoked by `extract-all-bc-files.sh` which is a shell script (credits to SMU PhD student Tu Haoxin[10]).

After deploying either of the 2 methods above, the compile module will then run KLEE on the generated bitcodes with the necessary command flags chosen by the user. (The user would have been able to specify the required KLEE command flags to run the bitcode from the UI which then generates a `klee_command.json` file. The compile module reads this file and appends the required flags to the KLEE command)

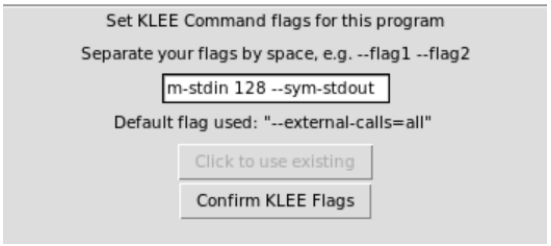


Figure 4.0: Screenshot of UI when inputting custom KLEE flags

With all these steps done, we have successfully automated the steps required for the user. KLEE has been run, and the results are ready for analysis. This brings us to the next module, the Analysis module.

[9] https://github.com/trailofbits/cb-multios
[10] https://gist.github.com/Hanseltu/69ba7343e8640ef20a37ad8c2288041a#file-klee_runs_cgc-md

Analysis Module

This module aims to analyse the output of KLEE. It will extract the errors from KLEE, and come up with an approximation to the probability of possible vulnerabilities as well as their associated impact. We derive a risk score from this analysis. Additionally, we use KCachegrind to visualize the output.

To elaborate, output KLEE is automatically saved to a folder named klee-last, in the same directory as the C file. This file contains important information regarding the test results.

The files we are interested in are:

1. `.err` files
    a. Contains the error codes and other important information when running KLEE
2. `run.istats`
    a. KCachegrind file used for visualization

First, the program prompts the user for the location of their klee-last files.

```
[i] Please enter directory where your klee-last folder is in
```

Figure 5.0: Screenshot of input needed from user

Error Analysis

We first obtain the errors from all the .err files in all the klee-last folders and all the klee-out-n ($\forall\, n \in Z^{\geq 0}$) folders. Example of such a file: `test000001.exec.err.` This indicates the error produced when executing test case 1.

Let us look at an example of this error file:

```
Error: memory error: out of bound pointer
File: testoob.c
Line: 13
assembly.ll line: 31
Stack:
        #031 in main () at testoob.c:13
```

Figure 5.1: Screenshot of error thrown from KLEE

The analysis module extracts the most important information, the error, the filename, and the line the error occurred at. This is done via text parsing.

Then, the module cross-checks with a predefined dictionary as to whether the error in question is serious or not.

```
REFERENCE_DICT = {
  "memory error: out of bound pointer" : ["Buffer Overflow Vulnerability", "Significant",
}
```

Figure 5.2: Screenshot of error when dictionary is not found

Above is a snippet of the code.

Based on the error, the impact, as well as the probability of its seriousness, we will print an output to the user to convey this information in an easy to understand format.

We determined that buffer overflow vulnerabilities are Significant in impact after researching CWE, Common Weakness Enumeration (MITRE, 2021), run by MITRE, a company that specialises in cyber threat sharing, and cyber resilience (MITRE, 2021).

Scope	Impact
Availability	**Technical Impact:** *Modify Memory; DoS: Crash, Exit, or Restart; DoS: Resource Consumption (CPU); DoS: Resource Consumption (Memory)* Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.
Integrity Confidentiality Availability Access Control	**Technical Impact:** *Modify Memory; Execute Unauthorized Code or Commands; Bypass Protection Mechanism* Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy.
Integrity Confidentiality Availability Access Control Other	**Technical Impact:** *Modify Memory; Execute Unauthorized Code or Commands; Bypass Protection Mechanism; Other* When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

Figure 5.3: Screenshot of the CWE used

As we can see, the above impacts are dangerous and thus we decided on a Significant impact grading.

We determined the probability of seriousness by reviewing the results of running KLEE on DARPA codes[11]. Based on that analysis, we observed that most errors only appear once, and rarely, multiple errors may occur up to about 4 times. Therefore we decided, based on the occurrence rate of errors, to use this grading scheme:

[11] https://gist.github.com/Hanseltu/51a1fa263d3d3990ab6fa7c887847827

```
if count <= 1:
  risk = "low"
elif count <= 3:
  risk = "medium"
else:
  risk = "high"
```

Figure 5.4: Screenshot of analysis code

Any time an error occurs once, it is low, else if it occurs more than once but less than 4 times, it is medium, and finally if it occurs 4 times or more, it is high risk.

We also performed data analysis on the DARPA repository and determined the percentage of the time a memory out of bound pointer error is indicative of a buffer overflow error. This is because such an error occuring may not always mean that a buffer overflow vulnerability exists (it could be a false positive). Therefore, we inform the user of the probability (roughly 32.09%), to help him/her decide whether it is suitable for further analysis.

We also decided on the remediation steps from the user, referring to the robust CWE website, to advise the user on how to fix such a buffer overflow vulnerability.

Combining this information, we inform the user of the:

1. Error,
2. Which file it occurs,
3. How many times it occurs in,
4. The probability of the error being indicative of a vulnerability,
5. Impact of the vulnerability,
6. Percentage chance that the error is indicative of the vulnerability,
7. Remediation steps to fix the vulnerability

First, we tell the user which error occurs where:

```
In testoob.c, memory error: out of bound pointer has occured on line 13. Please refer to later analysis to determine if the error is dangerous.
```

Figure 5.5: Screenshot of error captured

Then, we perform further analysis to gather more information and inform the user of the detailed analysis of the possible vulnerabilities.

```
[!] In file testoob.c, the following errors have occured:
memory error: out of bound pointer: Appears 1 times.
[!] There is a low probability that there is a Buffer Overflow Vulnerability. Impact: Significant
[i] Based on DARPA repository code analysis, out of 134 memory errors, there were 43 Buffer Overflows (Occurence rate: 32.09
%)
[i] Remediation:

Strategy: Environment Hardening (CWE-120):

 When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed in
put values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.

 Strategy: Compilation or Build Hardening (CWE-121):

 Run or compile the software using features or extensions that automatically provide a protection mechanism that mitigates o
r eliminates buffer overflows. For example, certain compilers and extensions provide automatic buffer overflow detection mec
hanisms that are built into the compiled code. Examples include the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY
_SOURCE GCC flag, and StackGuard.
```

Figure 5.6: Example of output from our analysis module

KCachegrind Visualization

KCachegrind is an excellent profiling visualization tool, originally written for use with the callgrind plugin for valgrind[12]. We chose to integrate KCachegrind due to its useful features[13]:

- Direct support for profiles generated by Cachegrind/Callgrind
- Support for arbitrary event types and derived event types
- Sorted function list, with grouping according to ELF object/source
- File/symbol namespace (such as C++ classes)
- Correct handling of recursive cycles (similar to GProf)
- Various visualization views for a selected function, such as
    - Treemap in caller/callee direction
    - Call graph around function
    - Source & assembly annotation

We run KCachegrind if the user chooses to do so at the end of each analysis, on the latest klee-last folder:

```
[i] Would you like to run kcachegrind on the latest klee output in klee-last? (Y/n)
```

If the user chooses Y for yes, he/she is brought to KCachegrind for deeper analysis.

[12] http://kcachegrind.sourceforge.net/html/Home.html
[13] https://github.com/KDE/kcachegrind

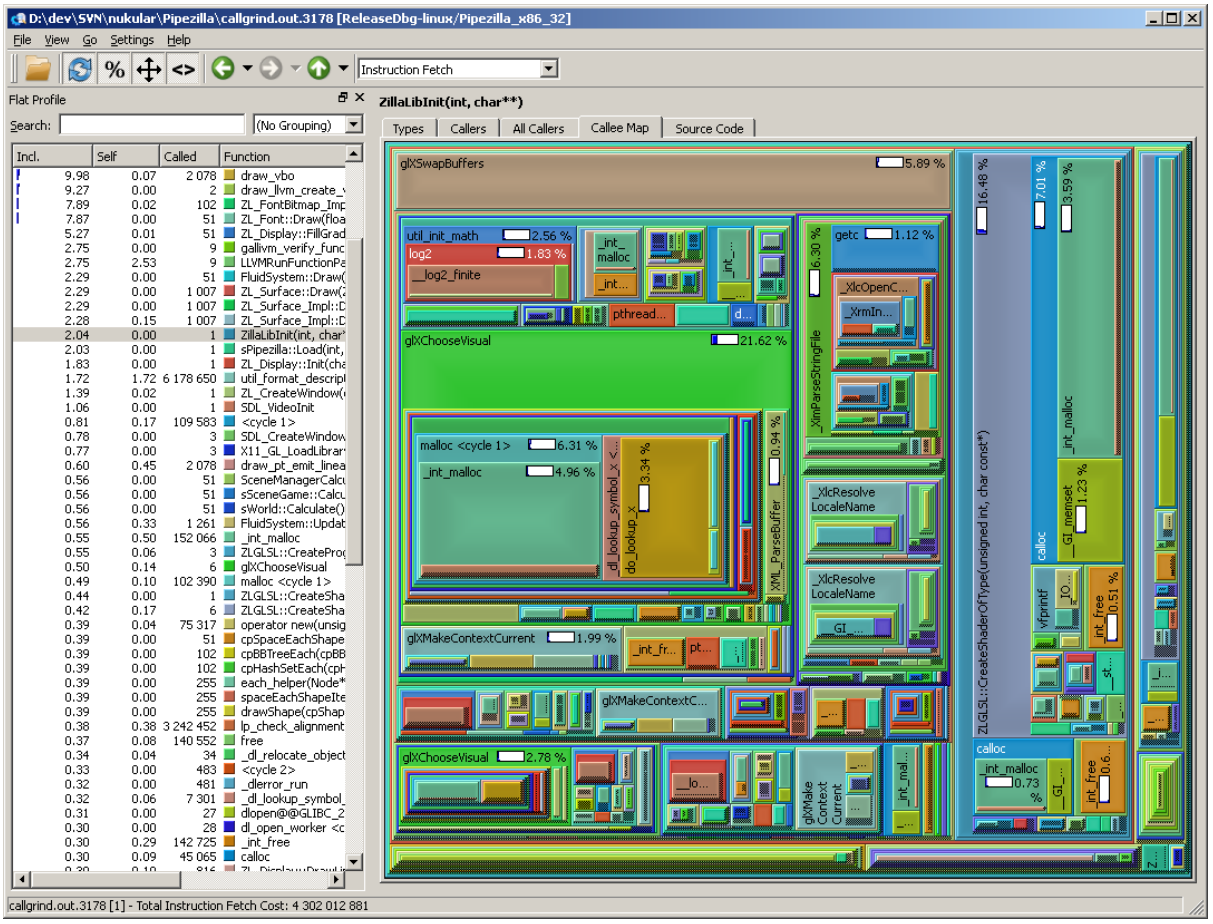


Figure 6.0: Example of KCachegrind graphical information

Above is an example of the feature-rich KCachegrind, offering the user graphical information about the code he/she is analysing. Some useful features of KCachegrind include information about Types, Callers, All Callers, Callee Map and Source Code.

Thus, we come to an end for our KLEENOTATION() analysis module. In the next section we will document our graphical user interface.

Graphical User Interface

Despite initially not requiring a graphical user interface (GUI) from our sponsor, after our first user testing we received the feedback that users find command line interface hard to read and understand. Hence we decided to adopt a GUI for easy access by the users. Our first User Acceptance test conducted on 19 February 2021 with the GUI was conducted in the format of A/B testing. The test shows that users who are less familiar with the command line interface found using KLEE and KLEENOTATION() more intuitive. The input of the required fields were also easy to understand without having to read through the entire KLEE Manual. You may refer to our test conclusions in the chapter of “Testing” in this report.

The main function of this GUI is for users who are not familiar with JSON files to create the file and run klee without having to input from the command line interface. The GUI does not facilitate the removal or editing of any tests. Users familiar with JSON files and the structure of the tool may upload a `config.json` file in the sample as outlined in Appendix C. The user can then open the GUI to click on the button “Run KLEENOTATION()” to simply run the tool without clicking “Start”. If the user clicks on “Start” accidentally, close the window to prevent emptying the existing `config.json`. The details are populated when clicking on the button “Confirm Add Test'' in the subwindow.

The user may edit the json files using `vim` or `nano` after its creation should he need to. The user may also use the `cat` command to display the contents to check before firing up the GUI to run the KLEE commands with the buttons on the main window.

The GUI requires the use of Xming as we are calling it from the Docker container to handle the X Window System[14]. We chose Xming because it is fully featured, lean, fast, simple to install and because it runs standalone on native Windows. It is also easily made portable (does not need a machine-specific installation or access to the Windows registry).

The user also needs to define and set the `DISPLAY` to the user’s local IP, with `<port :0.0>`, to his own Docker container accordingly. You may refer to our GUI Setup Guide above for detailed instructions. Alternatively, a GUI guide textfile is provided in the Docker image of KLEENOTATION().

[14] http://www.straightrunning.com/XmingNotes/

UI Explanation

The GUI main window shows 5 buttons.

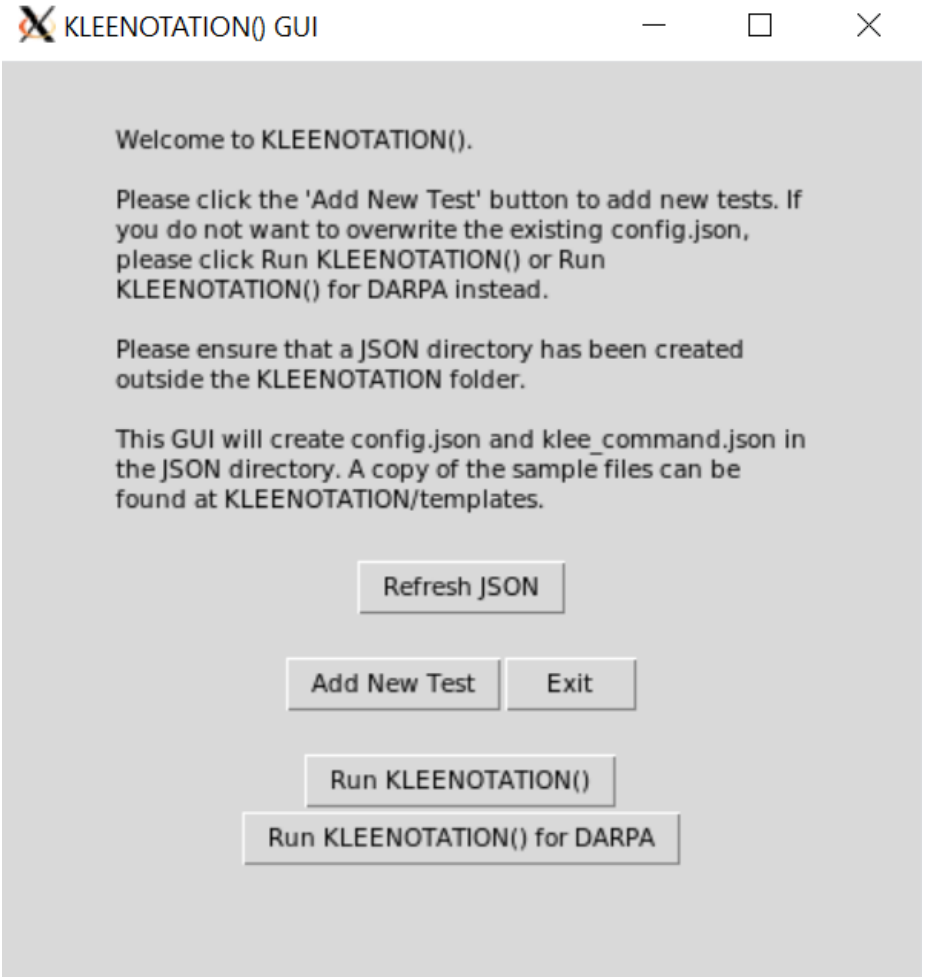


Figure 6.1: Screenshot of main window of GUI

1) **Refresh JSON**: Empties the existing json files to ensure there is no repeat runs of old test cases
2) **Add New Test**: To invoke a subwindow to add a new test case.
3) **Exit**: To close the main window and stop the program from running.
4) **Run KLEENOTATION()**: Invokes the tool to execute the parser module on the newly created `config.json` file and starts the automation process.
5) **Run KLEENOTATION() for DARPA**: invokes the tool to execute the parser module on the newly created `config.json` file. Invokes methods suitable for DARPA codes as they require further configuration.

The subwindow is invoked by the "**Add New Test**" button. It records every entry for a new test which is input to the `config.json` and `klee_command.json` files. The variables are as outlined in the following screenshot of the Subwindow.

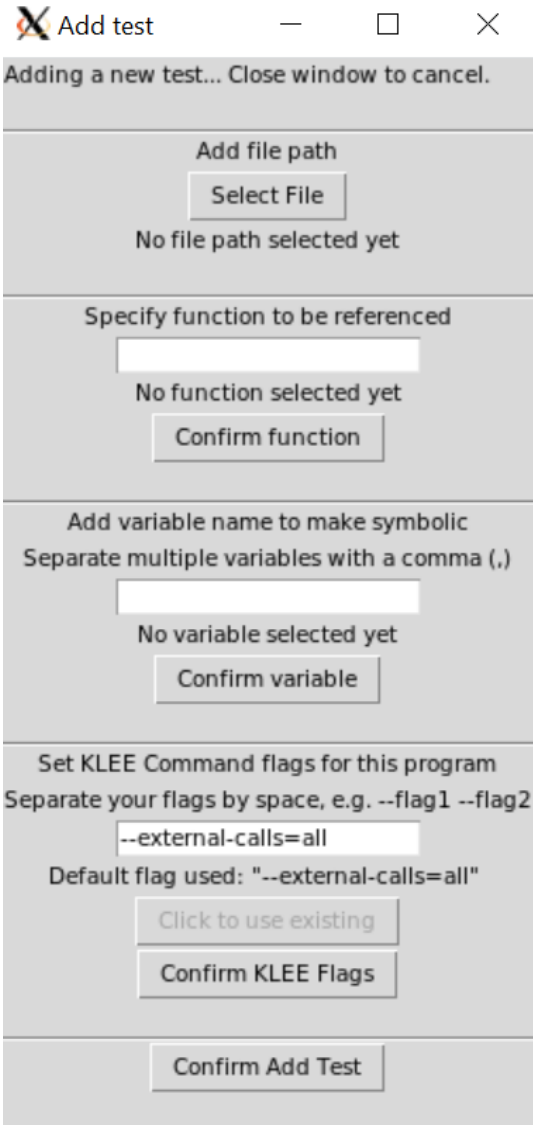


Figure 6.2: Screenshot of Subwindow to add a test

The `config.json` and `klee_command.json` files are rewritten every time the user starts up the Window and confirms adding a new test. This is to ensure that no run is repeated again, which would result in many klee-out files or even errors as the files have already been processed by KLEENOTATION().

Given the scenario when a user has already created a test case, if he wants to run another test case under the same directory as the file specified in the set test case, he may choose to use the previously-supplied flags. When a user selects the file and the commands are found in the `klee_command.json` file, the button "**Click to use existing**" will be enabled. Upon clicking the button, these flags will be populated in the entry field.

The user will still need to confirm the input by clicking on the button "**Confirm KLEE Flags**".

The GUI was developed with the tkinter library in Python version 3. The prerequisite step is to create a folder named "JSON" at the root directory (`/home/klee`). Afterwards, to invoke the GUI, run `python3 gui.py` in the GUI directory. The GUI will create `config.json` and `klee_command.json` files. This "JSON" directory would have already been created if the user installs the complete container from our group's deployed Docker image.

Smooth User Experience

During the programming of KLEENOTATION(), efforts have been made to increase the visibility and robustness of the code so that the user would have a much more seamless experience in using our tool. In addition, we have also applied some of the recommended practices such as "Modularity" and "Reusability" from industry standard ISO25010[15] to ensure that our code is of a minimal quality. The full details of the ISO25010 framework can be found in the appendix.

Error and Exception Handling within Codes

Throughout KLEENOTATION() modules, errors and exceptions when handling C programs have been accounted for.

This has been done through the use of `try, except` statements as well as setting Boolean values for when a particular search fails. Often, with complex codes that require the use of many custom libraries, `pycparser` might fail to parse them. In this case, we handle such cases by allowing the user to manually specify the line number instead through the console as shown in the code below.

```
#extract relevant lines
try:
    ast = pycparser.parse_file(filename[:-2] + "_PP.c")
    ast_text = ast.show(showcoord=True, buf=open(filename[:-2] + "_testout.txt", "w+"))
    ast_lines = []
    with open(filename[:-2]+ "_testout.txt", "r") as f:
        ast_lines = f.readlines()
    pycparser_passed = True

except:
    pycparser_passed = False
    function_found = False
    print("pycparser failed to parse " + process[-1])
```

Figure 7.0: Screenshot of error handling

[15] https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

In doing so, we implement error protection measures to ensure our code remains robust and can give the user alternative methods to carry on with the process should the automated one fail.

Status Messages

Whenever our code proceeds on to the next step of its automated process, a status message is output to the console so that the user is aware of the current progress of their task.

Examples of some of these outputs are shown below:

```
[*] Checking existing fake headers [*]
[*] Handling file headers of /home/klee/cb-multios/challenges/Palindrome/src/service.c [*]
 [!] missing headers located. Now generating headers for pycparser [!]
 [!] missing headers located. Now generating headers for pycparser [!]
```

```
[*] Parser Module DONE parsing the target code(s), now calling Annotate Module[*]
[['/home/klee/cb-multios/challenges/Palindrome/src/service.c', 'cgc_check', 'string', True, '47']]
[*] Now annotating string in /home/klee/cb-multios/challenges/Palindrome/src/service.c [*]
The Target DARPA programs to compile are:
['/home/klee/cb-multios/challenges/Palindrome/']
[*] Inserting LINKER statements into CMakeLists.txt [*]
[*] Commencing intermediary file clean up before compiling DARPA code(s) [*]
[*] Annotate Module DONE, calling Compile Module [*]
[*] Compiling DARPA codes... [*]
```

Figure 7.1: Screenshot of example output

Testing

Unit Testing

Continuous unit testing was carried out along the process of tool development as the first phase of testing. Attached in the Appendix is the link to all of the unit testing templates for each module in the tool. In total, there are five (5) modules in development and thus, 5 unit testing files The total number of test cases run is 205. A breakdown of test cases implemented for each module can be found below.

Module	Total number of test cases
Parser	50
Annotate	50
Compiler	50
Analysis	20
GUI	30

The unit testing would be scheduled on the day after progress or any change has been made to the module or the module has been completed. In conducting the unit test cases, it is noteworthy that our group followed the same procedure as we learnt in the module Software Project Management - one of the prerequisites before embarking on this project - hence, we did not utilise external resources such as Python's `unittest` or `pytest` module.

Unit testing procedure:

1. Create the template to fill in the results of the test cases
2. Create test cases according to the purpose of the module or according to the new change/progress, whichever applies
3. Run the code module with the specific test cases, one at a time
4. Note down the result of the test in the template. If there is any bug, log in the bug metric
5. Repeat the process with the other modules

Benefits of regular unit testing:

- Any bugs are found easily and more quickly. This means we can log them in the bug metrics earlier and plan the solution accordingly

- Ensuring that every module works to achieve a quality product and meet the standard ISO requirements
- Allowing for code refactoring and design improvements. By conducting regular unit testing, we were able to refine our functions in the code to reduce the complexity and speed up the time of the automation process
- Aligning with the agile methodology of the project management
- Providing resources for documentation

Integration Testing

In the first phase of development, which was before UAT 1, we took the Incremental Approach to our integration testing. In the later phase, which was before UAT 2, we implemented both Incremental Approach and Big Bang Approach.

Incremental Approach

1. Parser and Annotate were integrated
2. Compiler was integrated
3. KLEE run was integrated
4. Virtual environment Debian was integrated
5. GUI was integrated
6. Compiler_DARPA was integrated
7. Analysis was integrated

Big Bang Approach

All modules were integrated together on 16 April and tested for a smooth flow.
Benefits of integration testing in this manner include:

- Ensuring that integrated modules work properly
- Detecting errors related to the GUI between modules
- Covering more test cases and aiming to simulate real-life experience

System Testing

System testing was carried out twice during our project, each right before user acceptance testing respectively. In system testing, we tested the whole complete program within the correctly built environment. The details of the environment are as follows:

1. System Testing 1:
    - Docker image: `klee/klee:2.1`
    - Dependencies as specified in the dependencies/installation document

2. System Testing 2:
    - Docker image: `klee/klee:2.1`
    - Virtual environment: Debian
    - Graphical user interface: `tkinter`
    - Dependencies as specified in the dependencies/installation document

A system testing session was also conducted on 22 March with our project sponsor Professor Jiang Lingxiao. This session was initially scheduled for user acceptance testing; however, after discussing with our sponsor, we have decided to convert it to a client system testing session instead. The reasons for this are that the specifics of the testing are more specific to a user who understands the inner workings of KLEE. Therefore, if we were to do UAT, most users would not be able to interpret the usefulness of the output. With these reasons in mind, our sponsor had advised us to do client system testing instead.

Benefits of system testing:

- Ensuring that the tool was working correctly from the point of view of a user
- Ensuring proper running of the tool before user acceptance testing
- Simulating a complete flow of all modules in the build environment

User Acceptance Testing

Strategy and Plan

The last phase of testing carried out was User Acceptance Testing (UAT). Our group hosted two sessions of user acceptance testing on 19 February 2021 and 21 April 2021. The method used in both sessions was Alpha - Beta (A/B) Testing. The primary purposes of such a method were:

- Making sure that the tool developed work entirely and completely
- Gauge the differences in perspectives between version A and version B of the tool
- Gather feedback from the users about what to improve and what to further develop
- Measure the true statistics (quantitative feedback) when running the tool

Below is a short summary of the versions of the tools that our group implemented for UAT 1 and UAT 2 respectively

UAT 1:

Version A	Version B
Run the tool separately for each of the 5 test cases	Type the requirements in one (1) JSON file and run the tool once using the JSON file

UAT 2:

Version A	Version B
Type the requirements in one (1) JSON file and run the tool once using the JSON file	Use the graphical user interface to input the requirements and run the tool

Benefits of user acceptance testing

- Validating that business requirements were met
- Reducing risks of finding defects post-production
- Identifying the behaviour of the fully developed tool and perspectives of the end-users

Execution

For both UAT sessions, our group followed the same procedure as we learnt in the module Interactions Design & Prototyping, which was taught alongside Software Project Management. The procedure for UAT is as follows:

1. Create the testing persona and scenario.
2. Create the consent and testing templates.
3. Create the test cases for UAT and place them in a clean directory, isolated from the production environment.
4. Invited participants to sign the consent and NDA.
5. Participant to run the KLEENOTATION() tool on the facilitator's personal computer device. A facilitator is any member of the project team.
6. Participant to note down the results of the test cases and specify if the test passed or failed.
7. At the end of the testing, participants would be asked for feedback on the experience. Note that this feedback would not be taken into consideration for analysing the testing results but rather for the team's own improvement.

Feedbacks recorded in the UAT:

- Time taken to carry out the tasks
- Number of mistakes/slips (typo, accidental click, etc)
- Number of tasks passed
- Think-aloud data (questions or feedbacks during the process of running the tool)

- Critical incidents (program stopped running after click, etc)

Results

UAT 1:

Quantitative data:

Version A

- Average time taken: 839 seconds - 14 minutes
- Average difficulty rating: 3.2/4
- Passed tasks rate: 100%

Version B

- Average time taken: 485 seconds - 8 minutes
- Average difficulty rating: 1.9/4
- Passed tasks rate: 100%

Qualitative data:

- 8/10 of the participants completed the tasks without or with little help from test facilitator
- Participants generally:
    - Find the tool easy to use
    - Find the tool helpful for automating pre-KLEE tasks
    - Find inputting the file name and variable name tedious and prone to mistakes
    - Find the command-line tool fine to use, but would prefer a GUI
    - Move from module to module seamlessly

Verbal usability feedback:

- There is no GUI, which can be not friendly for non-IT people
- Typing in JSON format in command-line tool is cumbersome
- Would be good to have a GUI
- Tasks were sped up significantly with the tool compared to the manual process

Actions taken after UAT 1:

- Graphical User Interface (GUI) module was developed
- All other modules were further enhanced to handle more complex C programs
- Error Analysis module was developed to show intuitively the results of the tool as well as map the errors to a vulnerability analysis matrix

UAT 2:

Quantitative data:

Version A

- Average time taken: 621 seconds - 10.35 minutes
- Average difficulty rating: 3.3/4
- Passed tasks rate: 100%

Version B

- Average time taken: 569 seconds - 9.48 minutes
- Average difficulty rating: 2.2/4
- Passed tasks rate: 100%

Qualitative data:

- For both versions, 7/10 of the participants completed the tasks without or with little help from test facilitator
- Participants generally:
    - Find the graphical user interface more intuitive and easier to use than the command line interface
    - Find the tool helpful for automating pre-KLEE tasks
    - Find inputting the file path, file name, and variable name tedious and prone to mistakes when typing in the JSON file in version A
    - Like to see the complete JSON file generated by the GUI

Verbal feedback:

- Command-line interface maybe better for IT people, but GUI is great for both IT and non-IT
- Some parts are still being displayed in the command line, which could be confusing to switch
- Tasks were sped up significantly with the tool compared to the manual process

Challenges and Solutions during Testing

In carrying out the various testing techniques, our group encountered a number of technical difficulties as well as other challenges. The main reasons for this being the fact that prior to the project, we had limited knowledge of the tools and technologies that would be used. However, as a team we were able to overcome the hardships and successfully implemented the above testing scheme for our project. Below is a table of the problems faced during testing sessions and the solutions we took to overcome them.

Challenge	Solution
Limited knowledge of technology (C language, KLEE, Python libraries, etc)	• Consult project sponsor • Research and self-study • Trials and errors
Technology complexity (module integration, environment integration, environment set up, etc)	• Implement pair-coding • Continuous unit testing • Trials and errors
Limited pool of users for testings	• Talk with more people to identify prospects • Ask project sponsor for connections

Results and Outcome of project

Value to Sponsor

KLEENOTATION() has enabled our sponsor to increase his efficiency in finding bugs through KLEE and symbolic execution. Through automated parsing, annotation, and compiling, the number of manual steps that was once required has been significantly reduced. KLEENOTATION() also provides an additional risk analysis to our sponsor through the analysis module where he can then make supported judgement on probability, impact, risk and remediation of the bugs found in the analysed code. This is something that KLEE has yet to implement and can be seen as extending the functionalities of an already industry proven tool. The value of KLEENOTATION() can only continue to be further increased as emerging technologies provide high potential for improvements to the tool.

As observed in our UAT findings, the addition of first automation, and then UI, has led to substantial time savings. The first UAT showed an improvement in the speed of annotating a C file by 42%, as well as a decrease in the difficulty level by 40%. The second UAT showed a further improvement in the speed by 10%, with a decrease in the difficulty level by 33%. (Please refer to Appendix 5 for more details)

Final Release

The final product will be exported as a docker image and procured to our sponsor. Due to privacy agreements, we will not release the source code yet, as of time of writing. However, if the sponsor explicitly allows us to, we may consider making the code open source, for example via Docker Hub.

Innovative Outputs

Our team came up with 2 innovative outputs for this project:

Analysis & Remediation

Apart from the promised automation of the process for KLEE analysis, the team went beyond the requirements and built an analysis module. The module aims to make sense of KLEE output files and analyse in-depth if the bugs identified by KLEE is an actual vulnerability. Moreover, we will trigger a warning message if the bug identified is an actual vulnerability.

Analysis on DARPA repository

From our analysis, we were able to determine the percentage chance that a memory out of bounds error is actually a buffer overflow vulnerability. This was done via data analysis on the DARPA repository. Using this novel technique, we were able to tell the user what the percentage chance is, so that he/she is aware of the true risk value assigned to that error, instead of being overwhelmed by errors. In doing so, we are enabling the user to make better value judgments on what code to fix.

Implementation Quality

Security

Our tool is primarily aimed at an offline environment, since we do not require Internet connectivity for it to function. Therefore, the risk of cyber attack via the Internet is negligible. However, we still have guidelines to ensure the security of the user of our program and his/her data.

Security Guidelines

Do not use Xming on untrusted connections. Using Xming remotely via untrusted connections could be dangerous due to Man-in-the-Middle attacks (Swinhoe, D., 2019). If the connection is untrusted, use Plink for Xming SSH client. Plink is a command line connection tool, similar to Linux/Unix SSH, that is used for automating secure remote

operations from XLaunch and Xmingrc[16]. SSH is a cryptographic network protocol for operating network services securely over an unsecured network. By using a secure implementation of SSH, we reduce the risk of a successful network attack (ssh.com, 2021).

```
var wshell = new ActiveXObject('WScript.Shell');
wshell.Run('"C:\\Program Files\\Xming\\plink.exe" -ssh -2 -X
colin@chamonix xterm -ls -rightbar',0);
```

Above is an example using Plink in a `.js` file to display a remote xterm[17].

Of course, another effective method is to ensure that the computer is air gapped and not connected to the internet, since our tool works with full functionality offline.

Reliability

As explained previously, our code comes with robust error handling features, such that even if one module doesn't work as intended, the user is able to step in to intervene. For example, if `pycparser` were to fail, the Exception will be handled and the user will be able to input his/her own choice of program line number, to annotate the variable. Therefore, regardless of any error, our code will be able to continue running smoothly due to error exceptions.

In our own tests as well as UAT tests, there have not been any random errors that occur. Our code performs the same tasks and gives the same results each time it is run. Our thorough code reviews and UAT / Unit tests, as mentioned previously, also ensure that our code is robust and reliable.

Technical Challenges Faced

KLEE and the Linux Environment

All of our team members have no prior background knowledge in KLEE and KLEE as a tool for symbolic analysis. We spent two months in advance to study and understand the documentation and the use of KLEE as a tool for symbolic analysis. Nonetheless with the help of our sponsors and proper knowledge sharing of the team we were able to use KLEE's primary functions to perform the basic analysis needed for our project.

[16] http://www.straightrunning.com/puttymanual/Chapter7.html#plink
[17] http://www.straightrunning.com/XmingNotes/xterm.js

All of our team members are also Windows users, with little prior knowledge of the linux systems. This increased the difficulty of getting the tool up and running since users of KLEE are required to be familiar with either Linux x86-64, FreeBSD or macOS environments since it is only tested and compatible on them.

DARPA Codes & C Language

We faced many initial challenges with DARPA codes as DARPA codes were specifically designed for its own kernel, DARPA Experimental Cyber Research Evaluation Environment (DECREE), without which we are unable to properly compile the raw C files into executables. Furthermore, DECREE is not accessible anymore. This also created further complications as without compiling the code, many of the code dependencies are unable to perform and will return errors when a custom function is being called upon and that function is not found in the generic C library. The DARPA codes themselves are also of relatively high complexity, making the task of automating through them harder.

Lastly, as we are not proficient in the C language, we faced a challenge in trying to accurately identify the variable which needed to be made symbolic for KLEE analysis, that will trigger the necessary error within the code. Hence, more time was spent for us to analyse the code and understand the code.

We engaged in consistent knowledge sharing sessions, so as to ensure that every member is on the same page. By splitting out the learning processes amongst multiple people, we ensured that we sped up our learning process and increased our efficiency.

Project Management

Methodology

After much deliberation and consideration, our team decided to use the Crystal Method, developed by Alistair Cockburn. In 1991, he was commissioned by IBM to develop a method for a specific project. After extensive research, he concluded that all successful teams shared the same pattern and techniques without using any specific Project methodology. He named this method the Crystal Method.

Our team adopted this approach as after reading Alistair's work, we felt that his ideas strongly resonated with us, and as such we chose to adopt the Crystal Method for our project management. We will explain in detail in the next section.

Crystal Method

The 7 Properties

Alistair Cockburn outlined 7 common properties in all successful projects, namely:

1. Frequent delivery
2. Reflective improvement
3. Osmotic communication
4. Personal safety
5. Focus
6. Easy access to expert users
7. Technical environment

Frequent Delivery

Frequent Delivery is the regular releasing of iterations of the software program. This idea comes from agile methodologies. Designers and developers decide what features to include in each release and they design and test for each release.

Our team does this via our constant meetings with our sponsor. (Please refer to the Communication section for more information). Whenever we have a major change in our project, we will release the iteration and demonstrate the changes to our sponsor. We will then gather feedback from the sponsor in terms of what he liked about the iteration, what he thinks can be improved, etc.

By releasing iterations, our sponsor will be able to spot problems earlier in the project which will save a lot of hassle later on. Another point on this is that if the sponsors decide that the project does not do things the way they'd like it to be done, then steps can be taken to resolve this before it is too late.

Reflective Improvement

"Reflective improvement involves developers taking a break from regular development and trying to find ways to better their processes. Iterations help with this by providing feedback on whether or not the current process is working. Meaning to say that we learn from our feedback and make improvements for the next iterations." (Cockburn, 2004)

By meeting with our sponsor frequently and gathering feedback, we are able to gather important information for reflective improvement. At the start of our meetings, we will reflect on what the sponsor has said, and create a plan of action as to how to progress with our development as outlined in our meeting agenda. This not only covers technical

progress but also teamwork and team operations. An example would be pairing for pair programming. The template of the meeting agenda is in the Communication chapter.

Osmotic communication

In Crystal Clear and the smaller Crystal methodologies, Osmotic Communication is used. Since our team is made of 5 people, we implement the Crystal Clear method.

"Osmotic communication involves the team being together in a room and getting information to flow around it. This is because if the developer has to break concentration to move somewhere else to ask a question then their thought process will probably be lost." (Cockburn, 2004)

To implement this, our team has in-person meetings often, to ensure that information flows quickly throughout the team. Questions can be rapidly answered and all the team members know what is going on as well as having the ability to correct any misconceptions that may arise. This way, we mitigate the problem of knowledge being kept only with one person, which may cause a bottleneck. Knowledge sharing is current and up to date and team communications are dynamic. New ideas can occur on the spot and discussion can be carried out immediately. Our team was thus also able to benefit from this practice as communication overhead is greatly reduced.

Personal Safety

"This involves the issue of free speaking within a group of people. If a person is ridiculed whenever they ask a question, suggest an idea, etc then they will be less likely to speak up the next time. The people in the team must be able to trust each other and feel free to speak up about issues or whatever arises." (Cockburn, 2004)

To ensure a conducive environment, we always encourage each other to speak up when they have any questions. Nobody is ridiculed and we treat each other with respect. We honor all questions and ensure there are no "dumb" questions, because all questions are helpful to the project.

Focus

Following what we have learnt in the module Software Project Management, before a new sprint, we reviewed what we have done during the last sprint in terms of both individual and team work, how effectively and efficiently was the progress and how complete was the tool. We would also map out to-do lists to ensure that all tasks would

be completed within the specified timeframe and the certain metrics to measure the progress

As we work conducively both in team and individually, we were able to minimise the distractions by not dragging the meetings and entrusting responsibilities to the members. Our meetings were generally kept to a length of at most three hours. This is to ensure that the focus of each team member is not broken, and that we program at our peak capacity during all meetings.

Easy Access to Expert Users

The sponsor of our project, Prof. Jiang Lingxiao, is a person of expertise on the topic of our project and is located in Singapore Management University. This means that when working on the project, we have ease of access to an expert user that can clarify questions that we have as well as give suggestions to the problems that we encountered.

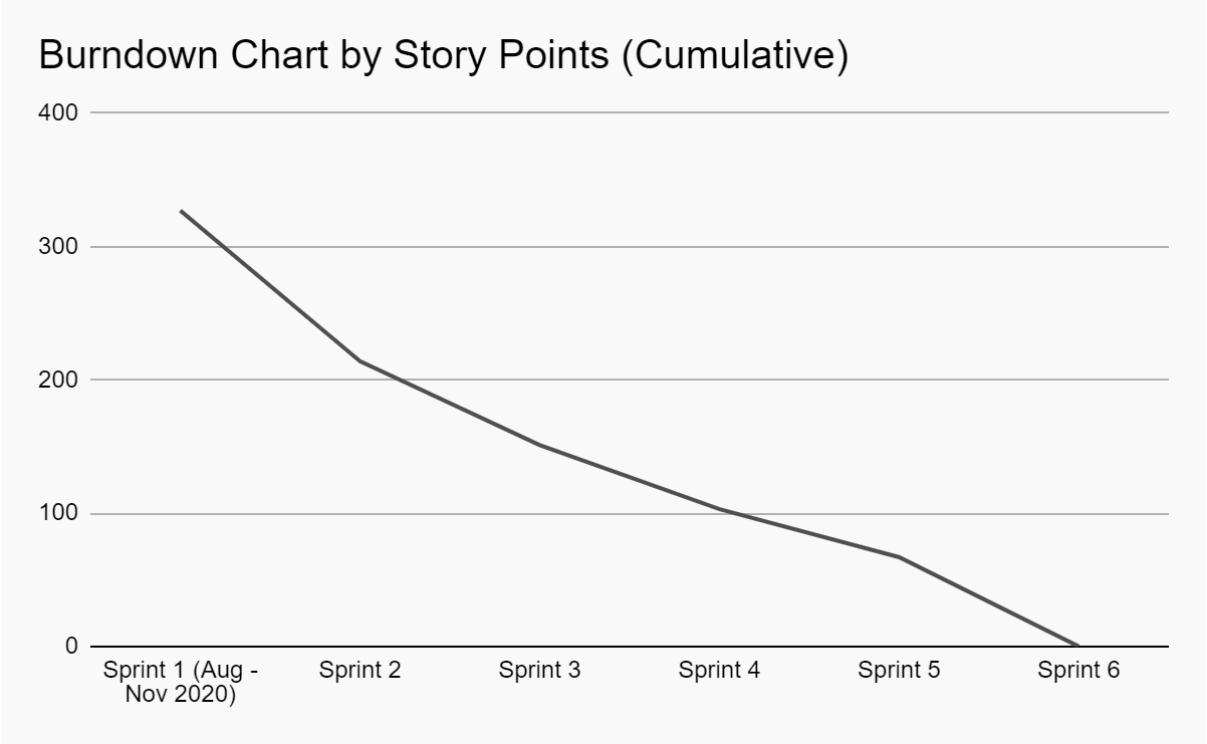
It also helps that Prof. Jiang is also highly approachable and willing to meet us either through physical meetings or email exchanges. As such, we had a total of 9 in-person meetings with him over the course of the project. He is also a real life user and is the intended end user, which means his feedback is more valuable to the project.

Technical Environment

The team meets up regularly between weeks for scheduled meetings to integrate the different modules and conduct regression testing on the programme. With the regular meetings amongst the team we are able to correct any errors when it first occurs before the error spun out of control. The problematic codes will be removed from the repository at the earliest possible stage. Moreover, with regular unit testing it allows the team to be able to quickly and effectively resolve the problems before they arise.

Metrics

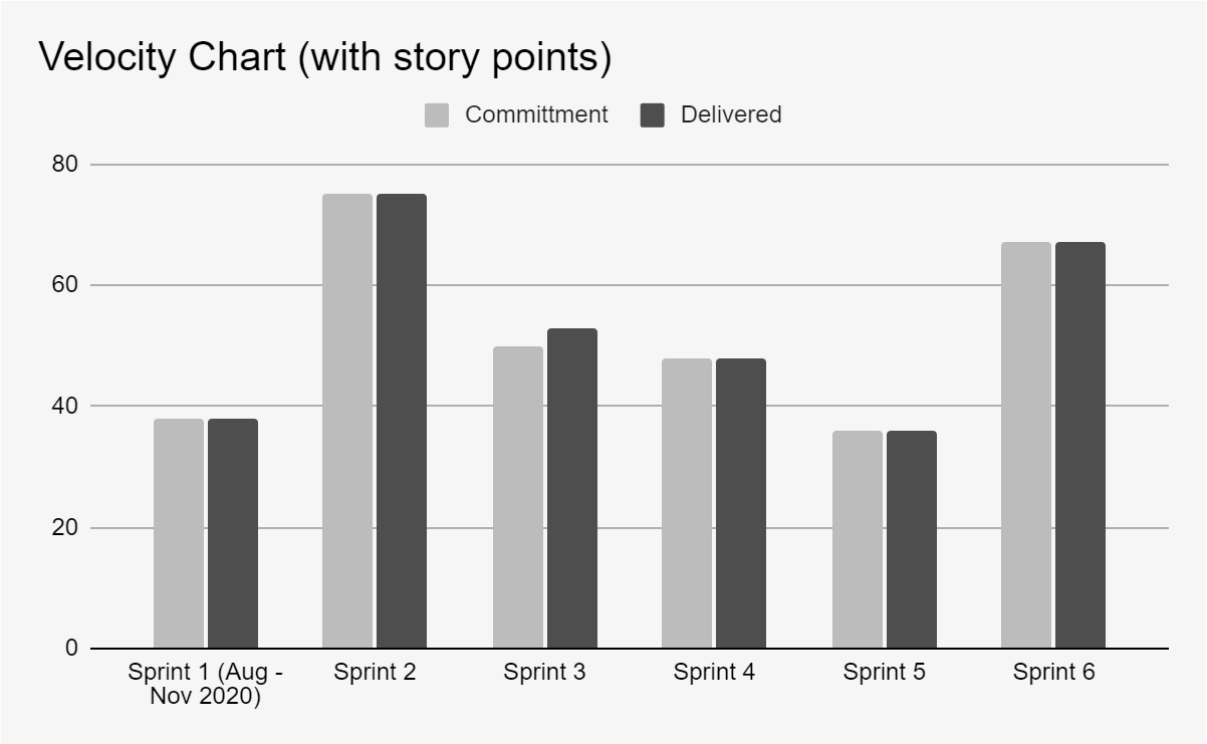
Burndown Chart



Burndown Chart by story points against sprints

A burndown chart is a graphic representation of how quickly the team is working through a customer's user stories, an agile tool that is used to capture a description of a feature from an end-user perspective. The burndown chart shows the total effort against the amount of work for each sprint. From the burndown chart above, we can see that our team has evenly spaced out the work to be done, such that it is decreasing in a stable manner. This is to ensure that all members are able to complete their tasks successfully without feeling exhausted.

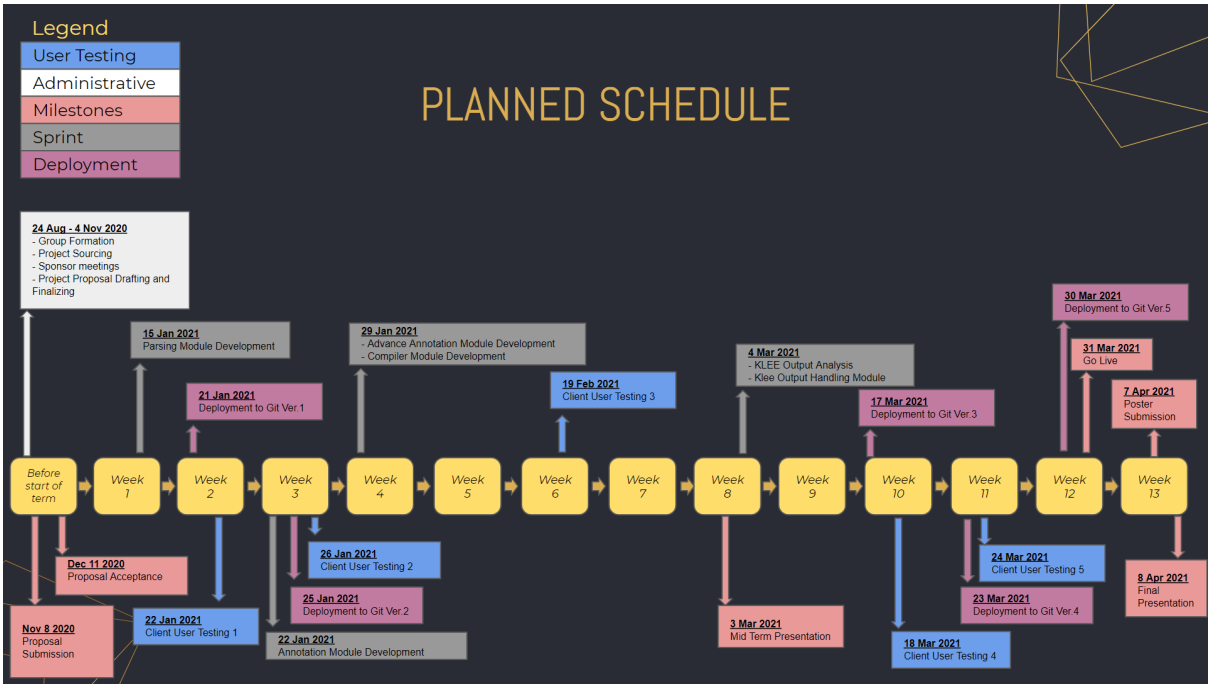
Velocity Chart



Velocity chart of story points against sprints

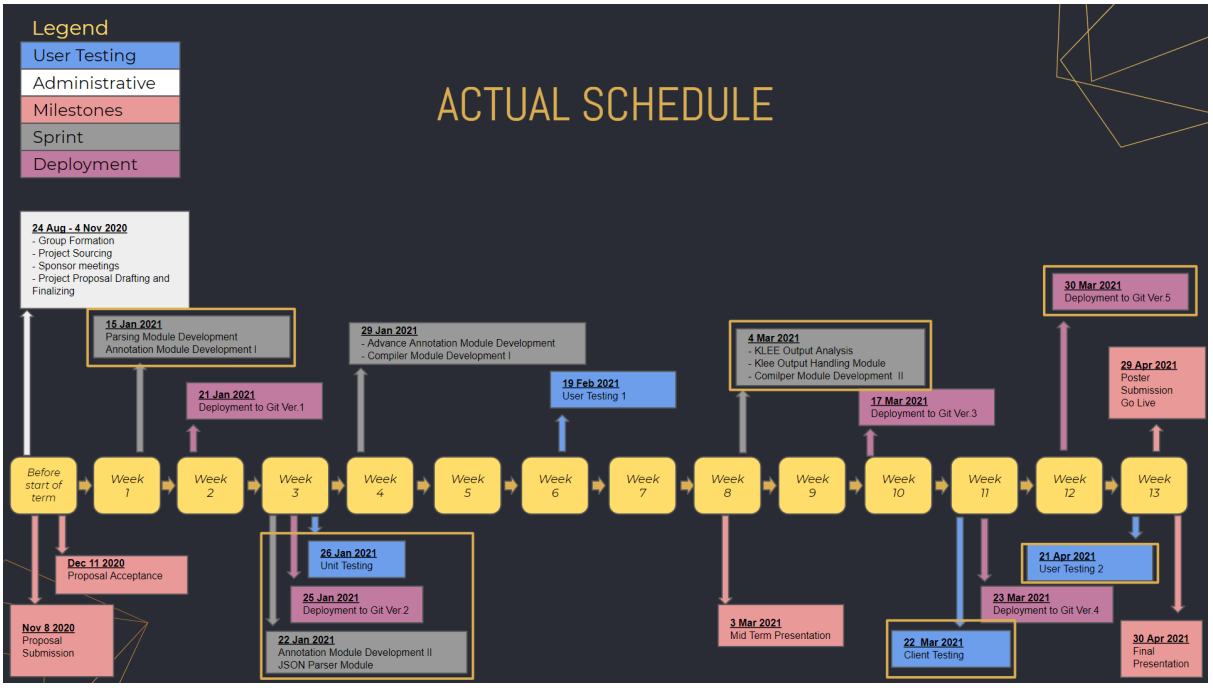
The Velocity Chart shows the amount of value delivered in each sprint, enabling us to predict the amount of work the team can get done in future sprints. From the velocity chart above, we can see that we have delivered all the work that we have committed in a timely manner.

Schedule



Overview of initial project schedule at project proposal phase

It can be observed that we followed iterative feature boxing in our schedule. However along the way, additions and modifications were made to the schedule as we made more changes to the project requirements.



Overview of final project schedule at project conclusion phase

Communication

Within the Team

The team fixed regular mandatory meetings every Friday throughout the duration of the project. The agenda is as outlined as below. It highlights the necessary tasks and events for each meeting. The time allocated takes into account the possibility of absentees for valid reasons and ensures members are all updated for this meeting.

Agenda Template		
Time Allocated	**Task/ Events**	**Member-in-charge**
5 minutes	Check-in with each task team on progress (SCRUM)	PM
15 minutes	Go through project requirements for any changes and discuss change management	All Task holders
30 minutes	Presentation of Bugs and Debugging	Necessary Task holders
15 minutes	Update Bug Logs and Documentations	Necessary Task holders
1 hour	Continue working on to-do	All Task holders
10 minutes	Sum-up of progress at meeting and acknowledge individual week to-do list	All members
Total Time/ Duration		2 hours 15 mins

The common difficulty of scheduling to pair-program was dealt with in a 2-pronged approach. Firstly, weekends were specially committed to make up for task completion without having to tap into weekdays unless needed. Secondly, we further split the work within the task pair to work at their own time before the stipulated internal deadline.

The primary medium of discussions were Telegram and Microsoft Teams. Microsoft Teams was used specifically for calls. This came in extremely useful when our members were unwell in compliance with responsible social distancing during the pandemic.

Minutes are recorded at each meeting for reference and shared on Telegram at the end of every meeting with emphasis on the updated week's to-do.

With the Stakeholders

Our team committed to the communication with Professor Jiang Lingxiao (project client/ project sponsor) and Professor Wang Yong (project supervisor) to keep up to date with the project requirements on both sides. The meetings were conducted biweekly and minutes were taken for all occurrences. Meetings were conducted in person and online.

The difficulty of this project was indeed the topic. As such, there was extra effort made to link up with relevant experts in the field for knowledge sharing which also resulted in project changes. These experts include Tu Haoxin, a PhD from Singapore Management University who we communicated with for knowledge sharing and participation in our testing for feedback and diversity in the users tested to cover expert users as well.

Documentation

Minutes

Minutes were taken for all meetings. Internal meeting minutes focused on the to-do and records the actions needed to implement the change in project requirements. Internal deadlines were set and communicated across the team in the Telegram chat to ensure acknowledgement and compliance. Meetings with the sponsor and the supervisor recorded the new findings and change in project requirements and the rationale to justify the change. The role of taking minutes was shared between only two members to stick with a similar format and style for standardization to maximise the readability of the minutes for quick follow ups.

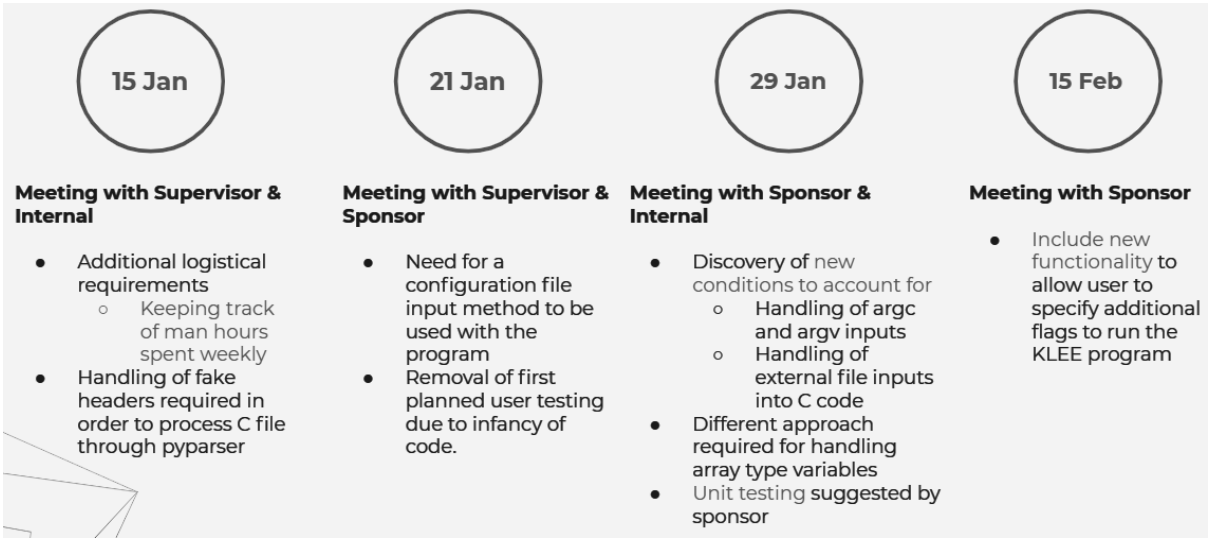


Figure 8.0: Snapshot of key meeting minutes from early iterations

Issues

Issues are brought up during the mandatory meetings but its resolution process begins upon its finding. That way we are able to get a headstart to the debugging before the meeting and leverage everyone's experience and maximise the usefulness of the bug tracking system. Non technical issues such as the scheduling and change in project requirements are communicated within the team and if necessary, discussed with the project sponsor. Issues with the team members, should there be any, were communicated in a timely and respectful manner in the Telegram chat to hold each other accountable to the team's time management.

Bugs

Bugs found were to be kept in record in our bug tracking system. Small bugs were also required to be logged so that the error description and its solution could be referred to for future testing and debugging purposes. The team found this very useful especially in the beginning when dealing with errors from KLEE were still difficult to comprehend.

Bug Metrics										
S/N	Iteration	Function	Description	Severity	Points	Mitgation	Discovered On	Resolved On	Final Bug Metric (Points Total)	Final Solved Bug Metric (Points Total)
1	1	Parser	Parser cannot handle fake headers (required process to generate AST)	High	5	Add functionality to handle fake headers	22-1-2021	27-1-2021	5	5
2	2	Parser	Parser cannot handle JSON formatted inputs	Low	1	Add JSON parsing using python library	27-1-2021	27-1-2021	6	6
3	2	Parser	Array type variables could not be located	High	5	Use search to find array declarations	3-2-2021	3-2-2021	11	11
4	2	Annotate	"&" value did not change when array was being made symbolic	High	5	Use conditional code to assign & only when needed	5-2-2021	5-2-2021	16	16
5	2	Annotate	Error thrown by KLEE when handling scanf functions within C code	High	5	Comment out lines in C program which requires user input	12-2-2021	15-2-2021	21	21
6	3	Klee Utilisation	Issue with os.system() running commands. Does not run command line as per string assigned	High	5	Use subprocess.run()	18-2-2021	18-2-2021	26	26

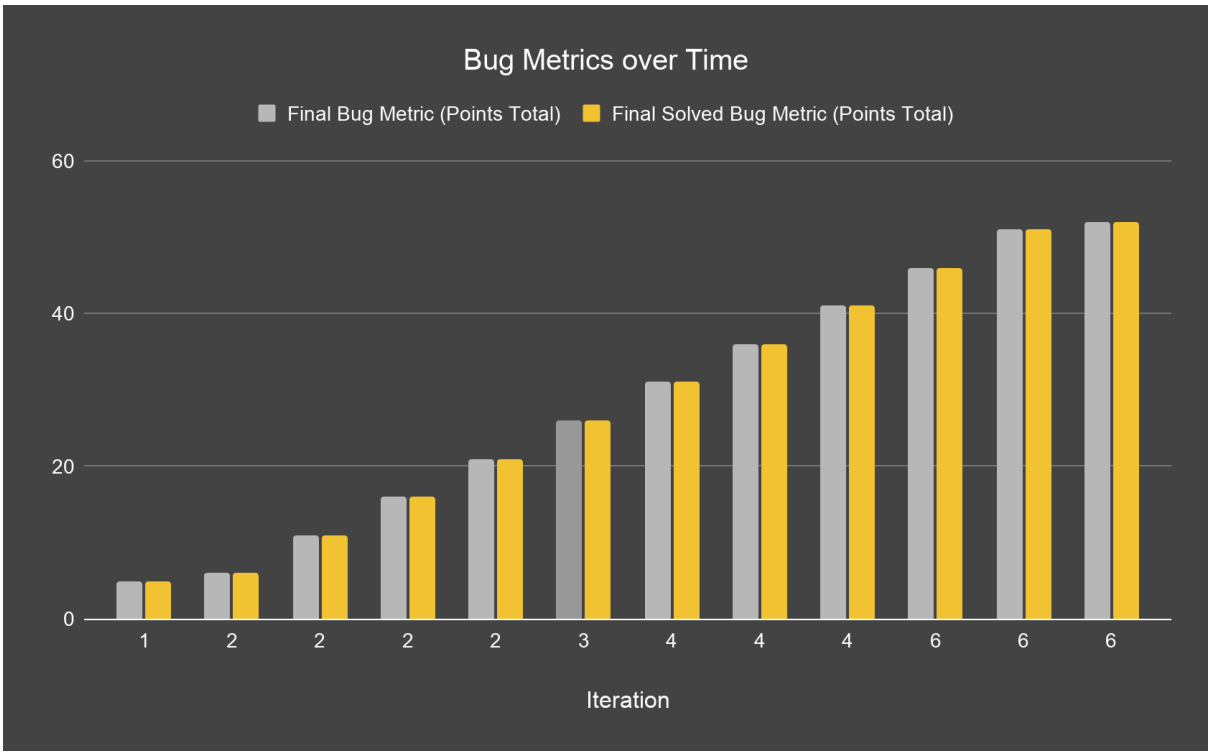
Figure 8.1: Snapshot of bug tracking system showing severity points

The severity points were useful in helping us manage our time efficiently where the goal was to resolve all bugs but prioritising the most critical bugs. The severity impact matrix and actions to take are outlined in the tables below.

Severity	**Description**
Low Impact (1 points)	Unimportant. Typo error or small user interface alignment issues.
High Impact (5 points)	The system runs. However, some non-critical functionalities are not working.
Critical Impact(10 points)	The system is down or is unusable after a short period of time. We have to fix the bugs to continue.

Points in Iteration	**Action**
Points < 10	Use the planned debugging time in the iteration.
Points >= 10	Stop current development and resolve the bug immediately. Project Manager reschedules the project.

Bug Metrics over Time



As seen from our bug metrics over time, we have successfully solved all our bugs as they have cropped up, ensuring reliability and stability of our program.

Resource and Task allocation

The staffing of the task took into consideration the number of people required, matching the people's skills to the task and to minimise conflict. We considered individual working styles, and experience which will contribute to the project's quality. All members were expected to source for relevant information and knowledge to fulfill their task regardless of experience.

The steps taken to reduce logical and task dependencies were timely updates and changes to schedule based on experience in previous iterations and changes to project requirements.

The project was scheduled to be in operation for only the semester. Much time was taken before the start of the semester to resolve initial prerequisites needed before embarking on the technical processes and development. This greatly boosted the beginning iterations as it would have been a bottleneck in knowledge sharing otherwise and prevented the start of the technical development. Emphasis was on the backend functionality which was a requirement from the project sponsor. Analytics and the GUI were of a lower priority on the list of project requirements as they form the bulk of our additional suggestions which will eventually be the base of our additional features and can be considered our X-factor.

Managing our manpower and time was crucial in enabling us to have the capacity to think beyond the project requirements and continually improve the quality of our features successfully.

Scope Changes with Risk management

Since the beginning of the project lifecycle in its initiation stage, there was a high risk which stems from the fact that the topic and coverage of this project has many components that have not been covered in the curriculum at this point in time. The team worked with a preventative approach to risk management and its implementation is reflected in change management. Allocating sufficient buffer time for research and knowledge sharing was crucial from the very beginning, before the start of its implementation and execution phases. This enabled the team to embark on technical development on time and freed up time to research further on improving the performance of the tool.

CHANGE MANAGEMENT

S/N	Iter.	Date	Change	Proposed by	Description	Action	Status
1	2	15/1/21	Parser module must handle fake headers within C file	KLEENOTATION ()	More time is required to implement fake header functionality	Work on annotation started early	Closed
2	2	21/1/21	User testing 1	Sponsor and Supervisor	Due to infancy and constrained functionalities of code, user testing 1 is cancelled	User testing 1 removed	Closed
3	2	21/121	Alternate method of input requested by Sponsor in the form of a configuration file	Sponsor	Individually specifying program inputs is not ideal. A configuration file can save more time	JSON file used as a configuration file and input for our program. JSON task added to schedule	Closed

Figure 9.0: Snapshot of change management document showing preventative measures in risk management

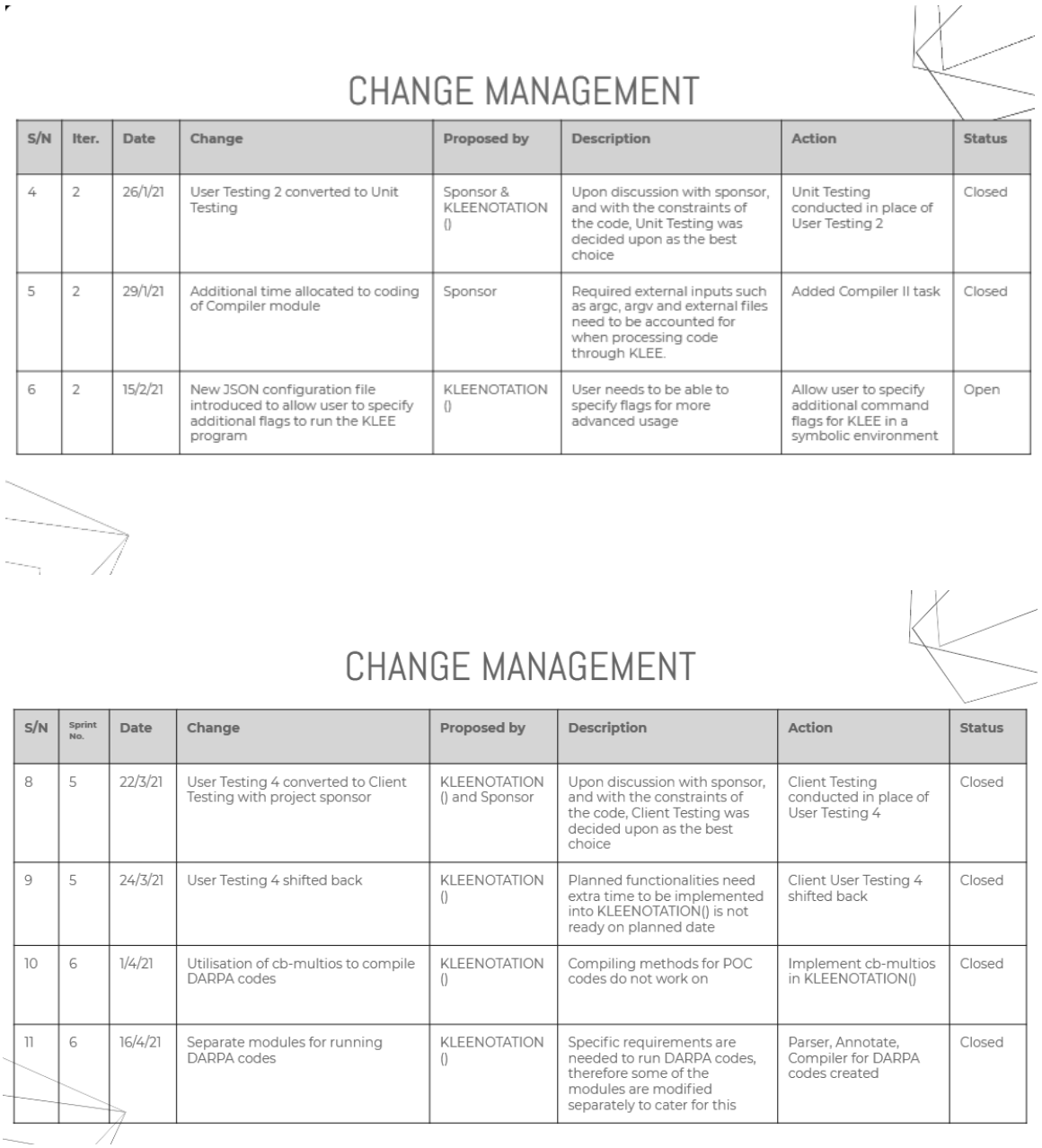
RISK MANAGEMENT

Risk Type	Risk Event	Likelihood	Impact	Mitigation
Project Management Risk	Time taken to complete a task may exceed the time planned	Low	Medium	• Use proper scheduling to ensure that we have enough buffer time • If not enough time, engage in change management to reschedule
Technical Risk	Little/no knowledge of the inner workings of KLEE, which may lead to more time taken to develop app	Low	Low	• Assign more time to develop modules if necessary • Conduct research on KLEE before embarking on the project • Knowledge sharing within group
Resource Risk	Unexpected case of laptop failure or crash	Low	High	• Code and other documents are backed up in Git and Google Drive, respectively
Client Management	Client may wish to add certain extra functionalities at a later point	Low	Medium	• Discuss with the sponsor to ensure he understands the time taken may increase • Engage in change management if more time is needed to reschedule

Figure 9.1: Snapshot of risk management documentation showing preventative measures

Iterative feature boxing was a key principle in our scheduling to ensure the quality of our project. Within each iteration, the tasks and resources were allocated efficiently to fulfill the requirements of the feature and allowed for sufficient buffer time. In circumstances where changes due to newly found information on the technical tools used, there were necessary changes made to the schedule to allow for more time to fulfil the requirements and then move on to advancement and quality assurance of the feature measured by reliability testing. For example, we tested the Parser module with many different types of codes. With iterative feature boxing, regression testing was

enforced to ensure compatibility with other modules and the further automation of our tool to improve its usability.



CHANGE MANAGEMENT

S/N	Iter.	Date	Change	Proposed by	Description	Action	Status
4	2	26/1/21	User Testing 2 converted to Unit Testing	Sponsor & KLEENOTATION ()	Upon discussion with sponsor, and with the constraints of the code, Unit Testing was decided upon as the best choice	Unit Testing conducted in place of User Testing 2	Closed
5	2	29/1/21	Additional time allocated to coding of Compiler module	Sponsor	Required external inputs such as argc, argv and external files need to be accounted for when processing code through KLEE.	Added Compiler II task	Closed
6	2	15/2/21	New JSON configuration file introduced to allow user to specify additional flags to run the KLEE program	KLEENOTATION ()	User needs to be able to specify flags for more advanced usage	Allow user to specify additional command flags for KLEE in a symbolic environment	Open

CHANGE MANAGEMENT

S/N	Sprint No.	Date	Change	Proposed by	Description	Action	Status
8	5	22/3/21	User Testing 4 converted to Client Testing with project sponsor	KLEENOTATION () and Sponsor	Upon discussion with sponsor, and with the constraints of the code, Client Testing was decided upon as the best choice	Client Testing conducted in place of User Testing 4	Closed
9	5	24/3/21	User Testing 4 shifted back	KLEENOTATION ()	Planned functionalities need extra time to be implemented into KLEENOTATION() is not ready on planned date	Client User Testing 4 shifted back	Closed
10	6	1/4/21	Utilisation of cb-multios to compile DARPA codes	KLEENOTATION ()	Compiling methods for POC codes do not work on	Implement cb-multios in KLEENOTATION()	Closed
11	6	16/4/21	Separate modules for running DARPA codes	KLEENOTATION ()	Specific requirements are needed to run DARPA codes, therefore some of the modules are modified separately to cater for this	Parser, Annotate, Compiler for DARPA codes created	Closed

Figure 9.2: Snapshot of an example of our change management documentation showing change to improve KLEENOTATION()'s reliability and usability

The scope of the project continuously changed with every meeting and new discovery of the behaviour of our main tool, KLEE and the initial goal of working with codes from DARPA CGC Corpus Challenge. After necessary discussions within the team and with our stakeholders, the changes were implemented with risk management. There was still a focus on functionality and reliability as well as the other requirements such as usability, performance and supportability taking into account the difficulty of the project. It is

important to note that eventually, one of the changes were to implement a quality tier of our project to measure its performance and functionality on simple, intermediate and advanced complex levels of code where codes from the DARPA CGC Corpus Challenge was categorized under advanced as suggested and approved by our advisor Tu Haoxin and Professor Jiang respectively.

With the continuous new findings of the tools and our technical dependencies, it was a risk to stay focused on the project's main requirements. As such, with every change, there was emphasis on the main focus which was the main purpose of the product tool we were working on and this was shared at every meeting, serving as a compass.

Learning Points

In hindsight, the project scope was continuously growing rather than changing. The team's experience outlines the principles of managing resources through conscientious planning and effort to think ahead with preventing bottlenecks in mind. Where the team wants to improve the tool as a whole, motivation was kept going through effective communication to serve as team encouragement at times of change. This highlights the importance of managing at the psychological dimension to build rapport and stamina for team resilience.

Project Handover

From the beginning, we planned for the quality and managed the risks well throughout the project lifecycle. The team managed to explore and conclude findings which will be useful in future relevant projects universally, including the project sponsor who will be using this for research purposes. Scope changes were not a hindrance but an enabler in contributing to the body of knowledge in this field which we have documented in our project. The further development of this project is explored in the conclusion of this report.

Conclusion

Future Work

On 16th April 2021, near the end of our project, it was published that there has been work done on the topic as well, with certain research projects being carried out in SMU along the same vein. Further works hence include collaborations on the flexibility and usability of the tool. Flexibility would include examples such as the use of KLEE options though users are already able to decide so in the current setup. Usability would include the ease of installations and knowing the different options available through helpful tooltips.

An interesting aspect to explore would be to enhance the analytics of the code to find patterns in certain source codes found in DARPA Challenge Corpus to potentially develop a tool that warns developers when saving files with these potentially dangerous codes. We hope that this encourages developers to keep security in mind when working on codes and also share the use of KLEE to educate and enhance developers.

Another aspect to consider is the extensibility of our code. The analysis module, in particular, can be extended with a greater reference dictionary, to cover more errors and vulnerabilities, beyond buffer overflows. However, greater research would first be needed to ensure that the dictionary data is accurate and robust, similar to the one carried out in our report.

Additionally, the automation aspects can be further modified with other text search techniques such as regex.

Closing Note

This project has brought different people from different walks of life to learn something new, something that has not been taught within the curriculum. Though it was not easy at first, the experience was fulfilling and enriching. Our project forms a robust platform for deeper analysis using KLEE, and could be passed down to future cohorts to improve on.

We would like to extend our gratitude to our sponsor, Professor Jiang Lingxiao, and our supervisor, Professor Wang Yong, for their immense support and guidance throughout

this project. We are grateful for this learning experience and we look forward to more new experiences.

Reflections

Aldric Chong Kai Yuan – Project Manager

Working as the Project Manager has taught me many important lessons about leadership. For one, I have learned that leadership is by example. By walking alongside our teammates and working together, we are able to motivate them more. My experience with Leadership and Teambuilding, a course taught in SMU, has helped me immensely in being a good Project Manager. From that module, I have learned how to bring out the best in my teammates and how to identify their working styles. Based on their working styles, I have found effective ways to challenge them and bring out the best in them. I have also learned how to manage the project effectively using Scrum/Agile techniques. I also learned the importance of good time management and communication, as well as respecting each team member. Another value I learned is leading by example. If anything cropped up, I would work alongside my teammates in fixing the issue. I learned that a true leader does not give orders; instead, a true leader walks beside his teammates, encouraging them. All in all, this project has helped me grow as a person in both the realm of coding experience as well as management skills.

Adrian Chang Fu Ren – Lead Back-End Developer

Working on this project has been a fruitful experience and is also one that I believe I can continue to reflect on when tackling new challenges working in the Information Systems field. When tackling the project, the concepts and knowledge learned throughout the 4-year IS program were further reinforced and the value of knowing and applying them were also understood. Aside from this, many new and important skills had to also be picked up along the way in order to meet the sponsor's project requirements. An example would be the usage of Linux OS as well as Docker platform handling that has become an integral part of our project. The combination of self-learning and guidance from the school has allowed us to overcome hurdles and deliver the project.

Taking on the developer role in this project, I was also able to appreciate some of the struggles of creating a good tool that can meet the demands of the user while still remaining user-friendly. This balancing between the usability and functionality of KLEENOTATION() was considered in every step of the development process and I believe the team at the end has managed to deliver an easy-to-use product to our sponsor.

Ngo Thanh Van – Quality Assurance

Quality Assurance was a role that I have never taken up until this project, but the journey has been challenging yet fulfilling thus far. Along with conducting unit testing regularly for our modules and ensuring that our code worked and met the predefined standards before moving on to the next development phase, I also utilised past experience from previous course projects in school such as Interaction Design & Prototyping and Software Project Management to implement the user acceptance testing. Furthermore, in order to detail good test plans and test cases and monitor the whole developing process, I needed to understand the tools and technology used well. This prompted me to learn and practiced my technical skills more. Overall, the experience I gained during this project was very meaningful and has helped me grow greatly.

Qi Xiang – Deputy Back-End Developer

As the deputy back-end developer my role was to assist the lead developer in coding as well as handling any error messages or any bugs that arise.During the journey, I have learnt that it is necessary for us to plan ahead with the schedule and follow closely to it. Moreover, it is important for us to keep our progress in check and through constant communication with one another to ensure that everyone is aware of the progress.

Documentation is also important for us to ensure every change is properly documented, that way we can make sure that the code can be worked on by another member. That also meant we need to leave proper commit messages so that we are able to understand the work and changes made to the codes.

Loh Xiao Binn – Vulnerability Analyst

As a team player, I learnt when to take the initiative to get things moving when things get difficult. Motivation and team morale is crucial to maintaining or improving the team's productivity. I am grateful to be a part of a team where we took turns to push each other on. The team took on a very adventurous project to explore things we had never learnt of before and we worked with a 'can do' attitude in exploring the topic and ways to incorporate analytics. We took pride in being ambitious and avoiding groupthink to improve the resilience of our product.

Additionally, change is a constant in any project journey not only for project requirements but also its priority levels. As such, I am proud of the group's ability to adapt quickly to solve the problem at hand by helping each other out, and move on with the mentality of improving the quality of the code. It is because of this that I highly encourage future groups to work with people whom they have not worked with, to

facilitate a high-speed learning adventure, one that is not only enriching but also memorable. The project topic was foreign to us like the deep sea but the voyage was blessed with many people who helped us along the way and are excited to see what the future holds for this project!

Special Thanks

Our team would like to express our heartfelt gratitude to current SMU PhD student Mr Tu Haoxin for his support on this project.

Appendices

Appendix 1 - KLEENOTATION() User Guide

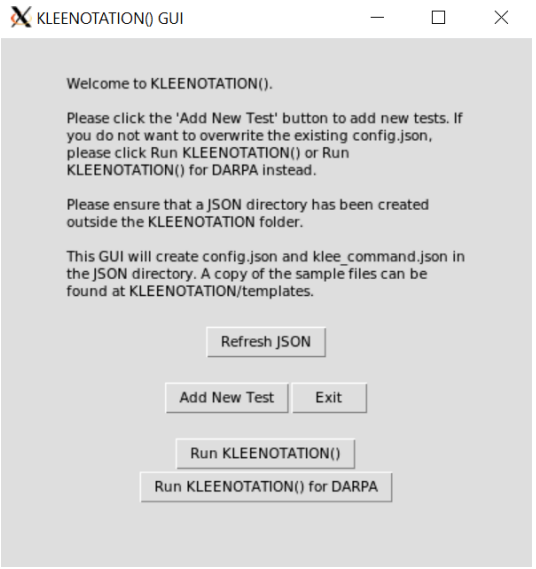
After completing the setup either through a fresh installation or from running the Docker image of KLEENOTATION(), using it will be straightforward and can be done in the following steps:

Step 0: Start KLEENOTATION() and ensure that a directory name “JSON” exists in the home directory of the container. (Note: This will already have been done for you if you are using the Docker Image version of KLEENOTATION())

Step 1: Navigate to the GUI folder (e.g. cd home/klee/git/GUI folder)

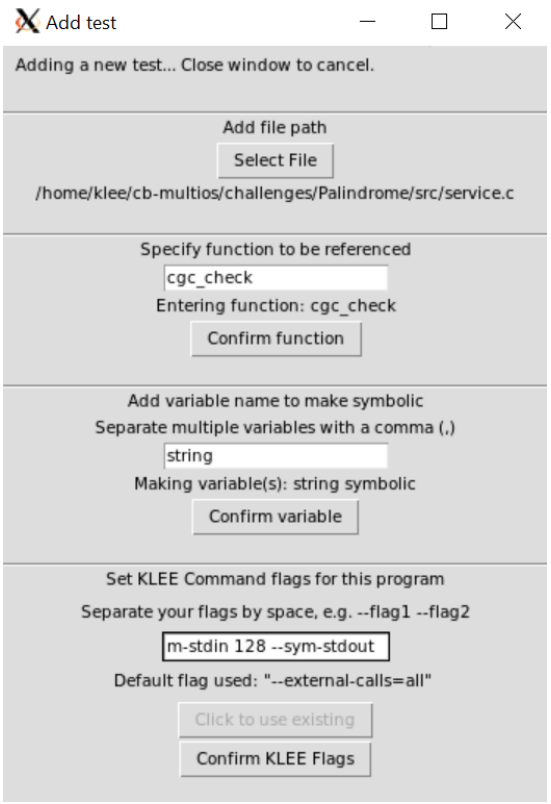
Step 2: Ensure that Xming Server is running on your local environment

Step 3: Run gui.py with python3 (`python3 gui.py`). A UI homepage window will appear as shown below:



Step 4: Click on the “Add New Test” button to begin.

Step 5: A new window will appear as shown below. Click on “Select File” and choose a C program file that you wish to test with KLEE.



Step 6: In the first text field, input the function name where your target variables are located. (Note: you can leave this field empty if you just want to run the file normally with KLEE)

Step7: In the next text field below that, input the names of the variables that you want to annotate with "`klee_make_symbolic()`" statements. For inputting multiple variables, separate them with a comma. (Note: you can leave this field empty if you just want to run the file normally with KLEE)

Step 8: In the third text field, enter the KLEE commands flags that you wish to use.

Step 9: Finally, click on "Confirm Add Test" and the JSON file that will be used by our parser module and compile module will be generated. The output JSON will also be displayed on the console for the user to verify its contents.

```
Printing from config.json...
{'/home/klee/POC/program4/testoob.c': [{'main': ['number1']}]}
Printing from klee_command.json...
{'/home/klee/POC/program4': [{'klee_flags': ['--external-calls=all']}]}
```

Step 10: Back at the homepage window, click on “Run KLEENOTATION()” to start the tool’s automated process on standard C programs. If you are carrying out testing on DARPA programs from cb-multios click “Run KLEENOTATION() for DARPA” instead.

Step 11: The console will prompt you for additional information required to complete the test. An example is shown below. Fill them out as necessary. (Note: if you have left the fields for the function and variables empty, you just input a random value when prompted for the line number and array)

```
[*] Checking existing fake headers [*]
[*] Handling file headers of /home/klee/cb-multios/challenges/Palindrome/src/service.c [*]
 [!] missing headers located. Now generating headers for pycparser [!]
 [!] missing headers located. Now generating headers for pycparser [!]
Are there any scanf occurances in the code to comment out? (Y/N): N
Running gcc command for /home/klee/cb-multios/challenges/Palindrome/src/service.c
Attempting to run pycparser on Palindrome
Unable to locate cgc_check due to unconventional declaration
Please specify the line number in the code for string in cgc_check: 47
Is string an array? (True/False): True
[*] Parser Module DONE parsing the target code(s), now calling Annotate Module[*]
[['/home/klee/cb-multios/challenges/Palindrome/src/service.c', 'cgc_check', 'string', True, '47']]
[*] Now annotating string in /home/klee/cb-multios/challenges/Palindrome/src/service.c [*]
```

Step 12: Once the console prints out the text “KLEE done” the automated testing process has completed.

```
KLEE: WARNING: undefined reference to function: __isoc99_sscanf
KLEE: WARNING: undefined reference to function: getenv
KLEE: WARNING: undefined reference to function: strlen
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignment of 8.
KLEE: WARNING ONCE: calling external: getenv(94625382869240) at /home/klee/cb-multios/include/libcgc.c:202 17
KLEE: WARNING ONCE: calling external: syscall(4, 94625382869128, 94625382406048) at /tmp/klee_src/runtime/POSIX/fd.c:528 5
KLEE: ERROR: /home/klee/cb-multios/challenges/Palindrome/src/service.c:67: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: HaltTimer invoked
KLEE: halting execution, dumping remaining states

KLEE: done: total instructions = 163176104
KLEE: done: completed paths = 180200
KLEE: done: generated tests = 1
```

Appendix 2 - Using Analysis Module

Step 0: Navigate to the KLEENOTATION() folder

Step 1: Run `./analyse.py`

Step 2: Input the folder where your klee-last files are located, and press <Enter>

Step 3: Console will print the full analysis, and give the option of opening KCachegrind. Input “Y” and <Enter> to open it, else input “n” and <Enter> to exit program.

Appendix 3 - ISO25010

Product Quality - ISO/IEC 25010

Characteristics	Sub-Characteristics	Definition
Functional Suitability	Functional Completeness	degree to which the set of functions covers all the specified tasks and user objectives.
	Functional Correctness	degree to which the functions provides the correct results with the needed degree of precision.
	Functional Appropriateness	degree to which the functions facilitate the accomplishment of specified tasks and objectives.
Performance Efficiency	Time-behavior	degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
	Resource Utilization	degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.
	Capacity	degree to which the maximum limits of the product or system, parameter meet requirements.
Compatibility	Co-existence	degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
	Interoperability	degree to which two or more systems, products or components can exchange information and use the information that has been exchanged.
Usability	Appropriateness recognisability	degree to which users can recognize whether a product or system is appropriate for their needs.
	Learnability	degree to which a product or system enables the user to learn how to use it with effectiveness, efficiency in emergency situations.
	Operability	degree to which a product or system is easy to operate, control and appropriate to use.
	User error protection	degree to which a product or system protects users against making errors.
	User interface aesthetics	degree to which a user interface enables pleasing and satisfying interaction for the user.
	Accessibility	degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.
Reliability	Maturity	degree to which a system, product or component meets needs for reliability under normal operation.
	Availability	degree to which a product or system is operational and accessible when required for use.
	Fault tolerance	degree to which a system, product or component operates as intended despite the presence of hardware or software faults.
	Recoverability	degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.
Security	Confidentiality	degree to which the prototype ensures that data are accessible only to those authorized to have access.
	Integrity	degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.
	Non-repudiation	degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later.
	Accountability	degree to which the actions of an entity can be traced uniquely to the entity.
	Authenticity	degree to which the identity of a subject or resource can be proved to be the one claimed.
Maintainability	Modularity	degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
	Reusability	degree to which an asset can be used in more than one system, or in building other assets.
	Analyzability	degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
	Modifiability	degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
	Testability	degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.
Portability	Adaptability	degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments.
	Installability	degree of effectiveness and efficiency in which a product or system can be successfully installed and/or uninstalled in a specified environment.
	Replaceability	degree to which a product can replace another specified software product for the same purpose in the same environment.

Quality in Use - ISO/IEC 25010

Characteristics	Sub-Characteristics	Definition
Effectiveness		accuracy and completeness with which users achieve specified goals
Efficiency		resources expended in relation to the accuracy and completeness with which users achieve goals
Satisfaction	Usefulness	degree to which a user is satisfied with their perceived achievement of pragmatic goals, including the resultsof use and the consequences of use
	Trust	degree to which a user or other stakeholder has confidence that a product or system will behave as intended
	Pleasure	degree to which a user obtains pleasure from fulfilling their personal needs

Appendix 4 - UAT Sample

```
{
"/home/klee/POC/program4/testoob.c":
        [
        {"main":["number1"]}
        ],
"/home/klee/POC/program5/test.c":
        [
        {"":[""]}
        ],
"/home/klee/cb-multios/challenges/Palindrome/src/service.c":
        [
        {"cgc_check":["string"]
        ]
}
```

4. Press Ctrl+S to save the file. Then, press Ctrl+X to exit the nano editor
5. Enter `nano klee_command.json`
6. Enter the following details into the file. Make sure all characters are correct

Full klee_flags command:

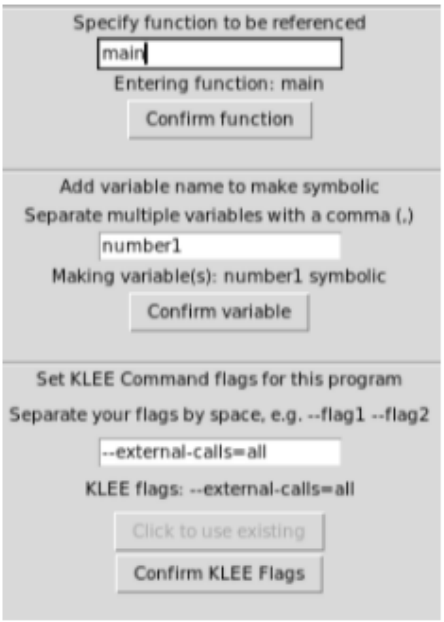
```
-link-llvm-lib=/home/klee/cb-multios/build64-backup/include/libcgc.so.bc
-link-llvm-lib=/home/klee/cb-multios/build64-backup/include/tiny-AES128-C
/libtiny-AES128-C.so.bc --simplify-sym-indices --write-cvcs --write-cov
--output-module --disable-inlining --optimize --use-forked-solver
--use-cex-cache --only-output-states-covering-new
--max-sym-array-size=4096 --max-time=300 --watchdog
--max-memory-inhibit=false --max-static-fork-pct=1
--max-static-solve-pct=1 --max-static-cpfork-pct=1 --switch-type=internal
--search=random-path --search=nurs:covnew --use-batching-search
--batch-instructions=10000 --posix-runtime Palindrome.bc --sym-stdin 128
--sym-stdout
```

UAT 2 - Version A template

Once inside KLEE image command line:

1. At home directory `/home/klee`, make a new directory by entering the command line `mkdir JSON`
2. At home directory, enter `cd FYP/FYP/UAT`
3. Inside ~/FYP/FYP/UAT, enter `export DISPLAY=<copied IPv4 address>:0.0`
4. Enter `python3 gui.py`

5. Click "*Add New Test*". When the second window pops up, click "*Select file*" and choose the following file at the directory `**/home/klee/POC/program4/testoob.c**`
6. Type the following inputs in the fields. Make sure all characters are correct



7. Click "*Confirm Add Test*"
8. Repeat the same steps (step 4 - step 6) for the next two test cases

UAT 2 - Version B template

Appendix 5 - UAT Findings

Main Data Summary		
	Mean time taken	Stdev time taken
Version A	839	72.84420209
Version B	485.4	103.6116682
Conclusion	The Wilcoxon test done shows that the mean total time taken for users to complete both tasks in version A is more or less the same as the mean total time taken for users to complete both in version B (p = 0.1165).	

Screenshot of UAT 1 finding summary on time taken

	Mean rating	Stdev rating
Version A	3.2	0.7888106377
Version B	1.9	0.7378647874
Conclusion	The Wilcoxon test done shows that the mean ranking of version A is not the same as the mean ranking if version B (p = 0.0002). On further analysis, we found that the mean ranking of version A is significantly higher than the mean ranking of version B (p = 0.9998). NOTE: Higher in ranking means more difficult to use As the p-values are larger than the significance level of 0.05, we can conclude that the results have a strong presumption towards the null hypothesis.	

Screenshot of UAT 1 finding on difficulty rating

Main Data Summary		
	Mean time taken	Stdev time taken
Version A	621	68.0098849
Version B	569.1	44.49831457
Conclusion	The Wilcoxon test done shows that the mean total time taken for users to complete both tasks in version A is more or less the same as the mean total time taken for users to complete both in version B (p = 0.1165).	

Screenshot of UAT 2 finding summary on time taken

	Mean rating	Stdev rating
Version A	3.3	0.8232726023
Version B	2.2	0.9189365835
Conclusion	The Wilcoxon test done shows that the mean ranking of version A is not the same as the mean ranking if version B (p = 0.0002). On further analysis, we found that the mean ranking of version A is significantly higher than the mean ranking of version B (p = 0.9998). NOTE: Higher in ranking means more difficult to use As the p-values are larger than the significance level of 0.05, we can conclude that the results have a strong presumption towards the null hypothesis.	

Screenshot of UAT 2 finding summary on difficulty rating

Conclusions from UAT Findings

From our first UAT,
The initial time taken is $839\ seconds\ \approx\ 14\ minutes$
The initial difficulty reading is 3.2
The final time taken is $485.4\ seconds\ \approx\ 8\ minutes$
The final difficulty rating is 1.9

Hence,
Percentage time reduced is $\frac{(14-8)}{14} \times\ 100\%\ \approx\ 42\%$
Percentage difficulty rating reduced is $\frac{(3.2-1.9)}{3.2} \times\ 100\%\ \approx\ 40\%$

From our second UAT,
The initial time taken is $621\ seconds\ \approx\ 10\ minutes$
The initial difficulty rating is 3.3
The final time taken is $569.1\ seconds\ \approx\ 9\ minutes$
The final difficulty rating is 2.2

Hence,
Percentage time reduced is $\frac{(10-9)}{10} \times\ 100\%\ \approx\ 10\%$
Percentage difficulty rating reduced is $\frac{(3.3-2.2)}{3.3} \times\ 100\%\ \approx\ 33\%$

Appendix 6 - Schedule Screenshot

Iteration	Task ID	DESCRIPTION	PLANNED DATETIME START	PLANNED DATETIME END	HOURS SPENT PLANNED	ACTUAL DATETIME START	ACTUAL DATETIME END	HOURS SPENT ACTUAL	Story Points
1	5.3	Meeting with sponsor to go through Draft 1	21/10/2020 15:00	21/10/2020 17:00	2	21/10/2020 15:00	21/10/2020 17:00	2	1
1	6	Project proposal Draft 2	01/11/2020 11:00	04/11/2020 13:00	2	04/11/2020 11:00	04/11/2020 13:00	2	3
1	7	Project proposal submission	08/11/2020 10:00	08/11/2020 13:00	3	04/11/2020 11:02	04/11/2020 13:02	2	13
1	8	Project proposal acceptance	10/10/2020 10:00	12/11/2020 13:00	3	12/10/2020 11:00	12/11/2020 13:00	2	5
					235.00			305	38
2	**1**	**Project Meeting**	11/01/2020 15:30	11/01/2020 16:30	1	11/01/2020 15:30	11/01/2020 16:30	**1**	1
		Meeting with supervisor	15/01/2020 17:30	15/01/2020 18:00	0.5	15/01/2020 17:30	15/01/2020 18:15	**0.75**	1
2	**2**	**Parsing Module design**	16/01/2020 10:00	16/01/2020 15:00	5	15/01/2020 18:30	15/01/2020 20:00	**1.5**	2
2	**3**	**Handling of simple c programme thru pycparser**	16/01/2020 15:00	16/01/2020 17:00	**2**	15/01/2020 20:00	15/01/2020 21:00	**1**	2
2	3.1	Parsing C program through parsing module	17/01/2020 10:00	17/01/2020 12:00	2	17/01/2020 10:00	17/01/2020 12:00	2	5
2	3.2	Handling of type declaration	17/01/2020 12:00	17/01/2020 13:00	1	17/01/2020 12:00	17/01/2020 13:00	1	3
2	3.3	Identifying line number of type decl	17/01/2020 13:00	17/01/2020 15:00	2	17/01/2020 13:00	17/01/2020 15:00	2	2
2	3.4	Returning AST table	17/01/2020 15:00	17/01/2020 16:00	1	17/01/2020 15:00	17/01/2020 16:00	1	3
2	3.5	Debugging for different type of variable	19/01/2020 15:00	19/01/2020 18:00	3	19/01/2020 15:00	19/01/2020 18:00	3	8
2	4	Meeting with supervisor & sponsor	21/01/2020 11:00	21/01/2020 12:00	1	21/01/2020 11:00	21/01/2020 12:00	1	5
	5	Project Meeting	22/01/2020 16:00	22/01/2020 17:00	3	22/01/2020 16:00	22/01/2020 17:00	1	1
2	**6**	**Annotation Module**	16/01/2020 10:00	16/01/2020 15:00	**5**	16/01/2020 10:00	16/01/2020 15:00	**5**	5
2	6.1	Creating fake c header	22/01/2020 15:00	25/01/2020 15:00	72	22/01/2020 15:00	27/01/2020 17:00	122	8
2	6.2	Insertion of make symbolic line	22/01/2020 17:00	22/01/2020 18:00	1	22/01/2020 17:00	22/01/2020 18:00	1	2
2	6.3	Identifying line number of type decl	22/01/2020 18:00	22/01/2020 19:00	1	22/01/2020 18:00	22/01/2020 19:00	1	2
2	7	Knowledge sharing session on debugging	22/01/2020 19:00	22/01/2020 21:00	2	22/01/2020 19:00	22/01/2020 21:00	2	8
2	11	Annotation Module	27/01/2020 17:00	27/01/2020 18:00	1	27/01/2020 17:00	27/01/2020 18:00	1	2
2	12	Push Daily Updates							
2	13	Update schedule for debugging	27/01/2020 20:00	27/01/2020 21:00	1	27/01/2020 20:00	27/01/2020 21:00	1	1
					115.5			159.25	75
3	1	Project Meeting	29/01/2020 15:30	29/01/2020 17:00	1.5	29/01/2020 15:30	29/01/2020 17:00	1.5	1
3	2	Create base for annotation module	05/02/2020 15:30	05/02/2020 17:30	2	05/02/2020 15:30	05/02/2020 17:30	2	2
3	3	Testing with DARPA	05/02/2020 15:30	05/02/2020 17:30	2	05/02/2020 15:30	05/02/2020 17:30	2	2
3	4	Identifying type array and unique handling for annotation module	05/02/2020 17:30	05/02/2020 18:30	1	05/02/2020 17:30	05/02/2020 18:30	1	3
3	5	Annotation module to handle struct	05/02/2020 17:30	05/02/2020 18:30	1	05/02/2020 17:30	05/02/2020 18:30	1	2
3	6	Annotation module to handle struct II	05/02/2020 15:30	05/02/2020 17:30	2	05/02/2020 15:30	05/02/2020 17:30	2	2
3	7	App Demo & Progress Update	11/02/2020 16:00	11/02/2020 17:00	1	11/02/2020 16:00	11/02/2020 17:00	1	8
3	8	Debugging parsing module	09/02/2020 15:00	09/02/2020 17:00	2	09/02/2020 15:00	09/02/2020 17:00	2	5
	9	Debugging for JSON.py	10/02/2020 15;00	10/02/2020 16;00	1	10/02/2020 15;00	10/02/2020 16;00	1	3
3	10	Debugging for annotation module	13/02/2020 17:00	13/02/2020 18:00	1	13/02/2020 17:00	13/02/2020 18:00	1	3
	11	Error handling of scanf	13/02/2020 18:00	13/02/2020 19:00	1	13/02/2020 18:00	13/02/2020 19:00	1	5
3	12	Integrating Parsing and Annotation module	18/02/2020 15:00	18/02/2020 17:00	2	18/02/2020 15:00	18/02/2020 17:00	2	13
3	13	Project Meeting							1
3	14	**Test Cases**	**19/02/2020 15:00**	**19/02/2020 17:00**	**2**	**19/02/2020 15:00**	**19/02/2020 17:00**	**2**	2
3	14.1	Creation of Test Cases	19/02/2020 17:00	19/02/2020 19:00	2	19/02/2020 17:00	19/02/2020 19:00	2	1
3	15	User Testing	19/02/2020 19:00	19/02/2020 21:00	2	19/02/2020 19:00	19/02/2020 21:00	2	1
3	16	Push Daily Updates	19/02/2020 22:00	19/02/2020 22:30	0.5	19/02/2020 22:00	19/02/2020 22:30	0.5	1
3	17	Bug fix on Angled	20/02/2020 10:00	20/02/2020 12:00	2	19/02/2020 14:00	19/02/2020 15:00	1	2
3	18	Updating project schedule for parsing module	20/02/2020 10:00	20/02/2020 12:00	2	20/02/2020 10:00	20/02/2020 11:00	1	2
3	19	Regression testing	21/02/2020 10:00	21/02/2020 12:00	2	21/02/2020 10:00	21/02/2020 12:00	2	2
3	20	Update schedule for debugging	22/02/2020 10:00	22/02/2020 12:00	2	22/02/2020 10:00	22/02/2020 12:00	2	2
					32			30	63
4	7.1	Creation of Test Cases	18/3/2021 12:00	19/3/2021 10:00		18/3/2021 12:00	19/3/2021 10:00		3
4	7.2	User Testing	19/3/2021 12:00	19/3/2021 17:00		19/3/2021 12:00	19/3/2021 17:00		6
4	8	Push Daily Updates	20/3/2021 12:00	20/3/2021 17:00		20/3/2021 12:00	20/3/2021 17:00		2
4	9	Regression testing	21/3/2021 12:00	21/3/2021 17:00		21/3/2021 12:00	21/3/2021 17:00		3
4	10	Update schedule for debugging	21/3/2021 17:00	21/3/2021 18:00		21/3/2021 17:00	21/3/2021 18:00		5
									48
5	1	Project Meeting	23/3/2021 12:00	23/3/2021 13:00		23/3/2021 12:00	23/3/2021 13:00		1
5	2.2	**Test Cases**	23/3/2021 14:00	23/3/2021 15:00		23/3/2021 14:00	23/3/2021 15:00		5
5	2.1	Creation of Test Cases	23/3/2021 15:00	23/3/2021 16:00		23/3/2021 15:00	23/3/2021 16:00		4
5	2.2	User Testing	24/3/2021 12:00	24/3/2021 18:00		24/3/2021 12:00	24/3/2021 18:00		6
5	3	Push Daily Updates	24/3/2021 18:00	24/3/2021 19:00		24/3/2021 18:00	24/3/2021 19:00		1
5	4	Updating project schedule	24/3/2021 19:00	24/3/2021 20:00		24/3/2021 19:00	24/3/2021 20:00		1
5	5	Regression testing	25/3/2021 12:00	25/3/2021 14:00		25/3/2021 12:00	25/3/2021 14:00		3
5	6	GUI prep json function	25/3/2021 14:00	26/3/2021 15:00		25/3/2021 14:00	26/3/2021 15:00		4
5	7	GUI create config_template.json function and write to	26/3/2021 15:00	27/3/2021 17:00		26/3/2021 15:00	27/3/2021 17:00		5
5	8	UAT KLEENOTATION (with GUI)	28/3/2021 12:00	28/3/2021 15:00		28/3/2021 12:00	28/3/2021 15:00		5
5	9	Update schedule for debugging	28/3/2021 15:00	28/3/2021 16:00		28/3/2021 15:00	28/3/2021 16:00		1
									36
6	1	Project Meeting	29/3/2021 14:00	29/3/2021 15:00		29/3/2021 14:00	29/3/2021 15:00		1
6	2	Bug Fix	29/3/2021 15:00	30/3/2021 17:00		29/3/2021 15:00	30/3/2021 17:00		4
6	3	GUI Style	31/3/2021 12:00	31/3/2021 15:00		31/3/2021 12:00	31/3/2021 15:00		3
6	4	input validations check for GUI	1/4/2021 12:00	1/4/2021 14:00		1/4/2021 12:00	1/4/2021 14:00		3
6	5	UAT validation	1/4/2021 14:00	2/4/2021 12:00		1/4/2021 14:00	2/4/2021 12:00		6
6	6	Code clean up	3/4/2021 12:00	4/4/2021 15:00		3/4/2021 12:00	4/4/2021 15:00		7
6	7	Finalize Requirements.txt	5/4/2021 12:00	5/4/2021 15:00		5/4/2021 12:00	5/4/2021 15:00		4
6	8	User Guide Readme	6/4/2021 15:00	7/4/2021 15:00		6/4/2021 15:00	7/4/2021 15:00		5

Appendix 7 - Meeting Minutes Example Screenshot

Internal Meeting 5/2/21

Points Discussed:

Technical Updates

- User has to specify the argv

Todo Tasks:

- Slides -> Celine
    - Intermediate testing
        - goals / features
        - Expected output
        - Actual output
- Use case
- Problem description
- Problem examples
- Bug tracker -> Qi Xiang + Celine
- Schedule -> Qi Xiang
- What is our value to the sponsor
- Team roles
- Design documents
    - Solutn archi, solutn model -> Qi Xiang
- Algorithm, Framework, Data challenges, Architecture, Design patterns, Flexible vs Hard

Appendix 8 - Unit Testing Documents

Parser (Links to an external google document)
Annotate (Links to an external google document)
Compiler (Links to an external google document)
Analysis (Links to an external google document)
Graphical User Interface (Links to an external google document)

References

Aldrich, J., & Le Goues, C. (2018). *Program Analysis (Spring 2018)*. https://www.cs.cmu.edu/~aldrich/courses/17-355-18sp/notes/notes14-symbolic-execution.pdf

Cadar, C., & Nowack, M. (2020). *KLEE symbolic execution engine in 2019*. https://link.springer.com/article/10.1007/s10009-020-00570-3

Cockburn, A. (2004). *Crystal clear a human-powered methodology for small teams*.

Dudina, I. A., & Belevantsev, A. A. (2017). Using static symbolic execution to detect buffer overflows. *Programming and Computer Software, 43*(5), 277-288. doi:10.1134/s0361768817050024

Kundel, D. (2020, June 11). Introduction to abstract syntax trees. Retrieved April 28, 2021, from https://www.twilio.com/blog/abstract-syntax-trees

MITRE, (2021). CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'). https://cwe.mitre.org/data/definitions/121.html

MITRE, (2021). CWE-121: Stack-based Buffer Overflow. https://cwe.mitre.org/data/definitions/121.html

SSH protocol is the standard for strong authentication, secure connection, and encrypted file Transfers. we developed it. (n.d.). Retrieved April 28, 2021, from https://www.ssh.com/academy/ssh/protocol

Swinhoe, D. (2019, February 13). What is a man-in-the-middle ATTACK? How MitM attacks work and how to prevent them. Retrieved April 28, 2021, from https://www.csoonline.com/article/3340117/what-is-a-man-in-the-middle-attack-how-mitm-attacks-work-and-how-to-prevent-them.html