PPA Coursework 1

Dimitrios Bouras, Karthik Ambu, Tommy Marroni

1 Overview of the Assignment

In this coursework, we are using program analysis tools on open-source projects, to find bugs in those projects and classify their outputs into true negatives, false negatives, false positives, and true positives. We have opted to use 4 different tools to analyze 5 C and 1 Java repos.

1.1 Project Selection

We have selected helper or utility projects from GitHub, all with over 100 stars, using the last commit of the main branch to perform our analysis. We focused on small simple to understand projects so that we could perform manual bug searches ourselves in order to identify false negative errors and so that false positives could also be identified by an examination of the errors that the tools found.

The projects we have opted to analyze are the following:

- Kilo: a small text editor in less than 1K lines of C code. It does not depend on any library (not even curses) and uses fairly standard VT100 (and similar terminals) escape sequences.
- Tiny-regex: a small and portable Regex library written in C, which supports a subset of the syntax and semantics of the Python standard library implementation (the re-module).
- Slre: an ISO C library that implements a subset of Perl regular expression syntax
- Sds: a string library for C, which improves the limited libc string handling functionalities by adding heap allocated strings that are easy to use, binary safe and computationally efficient.
- c_hashmap: a simple hashmap for C, which assists developers with ingesting JSON or similar map structures and using them with ease.
- Tinify-java: a java client for the Tinify API, which is used to compress images intelligently. It can be used as a dependency by other software to easily compress images.

1.2 Tool Selection

We have selected 4 different analysis tools to perform our analysis:

- Infer Static Analyzer: an open-source static code analyzer designed to identify potential software defects, security vulnerabilities, and other issues in Java or C/C++/Objective-C source code. We selected it due to its ease of use, deep code analysis capabilities and due to the fact that one of the leading tech companies in the world Meta, trusts it to perform code analysis on their products.
- SpotBugs Analyser: a platform-independent, static analysis tool primarily used for finding bugs in Java codebases. It operates by analysing Java bytecode rather than the source code directly, enabling it to identify potential issues such as null pointer dereferences, thread safety problems, and resource leaks. We selected this tool due to it's ease of use and deep analysis of Java bytecode. It also provides clear information and fix suggestions on each bug.
- cppcheck: an open-source static code analysis tool designed for detecting potential issues and vulnerabilities in C and C++ codebases. We selected it due to its user-friendly interface as well as the wide range of checks (including customisable ones) that it provides.
- KLEE: a symbolic execution engine designed for analyzing and verifying software. Its core advantage lies in its ability to perform symbolic execution, which explores multiple paths

through a program's execution by treating inputs symbolically rather than concretely. KLEE supports C and C++ code and is significantly more complex to use than the previous analyzers. The reason it was selected was that it offers thorough bug detection and path exploration due to its dynamic nature. We also had some experience using LLVM and Clang, thus KLEE would be a good fit for our team since LLVM is its compatible compiler.

2 Violations identified

In this section, we provide a summary of our experience with the tools, including the total number of bugs found per tool as well as a classification into false negatives, false positives, and true positives. The methodology used for that classification is presented in 2.1 ,the results will be presented in table format in section 2.2 and individual observations for each tool are summarized in sections 2.3.1, 2.3.2, 2.3.3 and 2.3.4.

2.1 Inter-rater agreement approach

In our analysis, we used inter-rater agreement to ensure accurate and consistent classification of violations in code. We could not be certain that the leveraged tools detected all bugs in the repositories or correctly classified all violations. To ensure accuracy, each team member independently reviewed the tool-generated findings as true or false positives and also manually checked the code for missed violations. Small and comprehensible GitHub repositories were crucial for this. Violations missed by the tools discovered during the manual analysis were labeled as false negatives, as well as discrepancies between tools in detecting violations. In cases where one tool mistakenly classified a violation (false positive) and another tool did not identify this violation it was deemed a true negative in regards to the second tool. In our analysis, the concept of true negatives is too abstract as it applies to all functional lines of code not flagged as violations.

After this initial classification, we conducted cross-checking sessions for team members to compare classifications and resolve disagreements through discussion, not majority vote. The agreed-upon classifications are presented in table format in section 2.2.

2.2 Results

	Kilo	**Tiny-regex**	**Slre**	**Sds**	**c_hashmap**	**Tinify-java**
Infer Static Analyzer	10	0	0	6	1	10
SpotBugs	-	-	-	-	-	12
cppcheck	0	0	1	3	0	-
KLEE	0	0	7	4	1	-

Table 1: Number of bugs found per tool per repo (- means the repo was not analyzed by this tool)

2.3 Tools

In this section, we delve into the practical application of the chosen static analysis tools with the goal of providing a detailed overview of our approach and the insights gained through this process. Each of these tools contributed uniquely to our findings, enhancing our understanding of the nature and frequency of various types of bugs in the code.

2.3.1 Infer Static Analyzer

Infer uses compile/build software like gcc, make or maven etc. to obtain binaries to perform static analysis. Infer provides as output along logs containing the issues that were found along with information about their location in the code.

Infer primarily observed null dereference, uninitialised variables and dead store issues for the C projects, and null dereference and thread safety issues for the Java projects. Most of the

Bug Counts for Different Repositories and Tools					
Repository	Tool	False Positives	False Negatives	True Positives	True Negatives
kilo	Infer	1	0	9	0
kilo	cppcheck	0	9	0	1
kilo	KLEE	0	9	0	1
Tiny-regex	Infer	0	0	0	0
Tiny-regex	cppcheck	0	0	0	0
Tiny-regex	KLEE	0	0	0	0
Slre	Infer	0	7	0	0
Slre	cppcheck	0	7	1	0
Slre	KLEE	0	1	7	0
Sds	Infer	2	2	4	5
Sds	cppcheck	2	5	1	5
Sds	KLEE	3	5	1	4
c_hashmap	Infer	0	1	1	1
c_hashmap	cppcheck	0	2	0	1
c_hashmap	KLEE	1	1	1	0
Tinify-java	Infer	0	11	10	1
Tinify-java	SpotBugs	1	10	11	0

Table 2: Bug Counts and Types for Different Repositories and Tools

information of the issues provided were valid, with a few exceptions, which mostly concerned uninitialised values. A simple example of where the static analyser could fail is when a variable is initialised in the beginning of the loop, with the variable defined outside of it.

2.3.2 SpotBugs Analyser

SpotBugs analyzes JAR files that require the Java Runtime Environment for them to be executed. Running SpotBugs opens a Graphical User Interface, where we can simply provide the path to the jar file of the Java application that needs to be analysed along with the source path.

SpotBugs provides deep insights in the following categories: Bad practice, Correctness, Malicious Code, Multithreaded Correctness, Random Code, Performance, Security and Dodgy Code. For the violations found, we get a detailed description, its category and possible fixes. Almost every issue reported by SpotBugs is true positive, with a few of them being insignificant.

2.3.3 Cppcheck

In our utilisation of cppcheck our attention was mainly directed towards error flags raised. These error flags are critical as they often point to potential faults in the code that could lead to incorrect behavior or system crashes. By focusing on these error flags, we hoped to be able to identify significant issues such as memory leaks, null pointer dereferences, and out-of-bounds accesses and efficiently catalog the most impactful and potentially harmful bugs present in the code. Conversely, we chose to overlook stylistic flags, such as 'include not found' warnings, as our goal was to pinpoint bugs that could directly affect the functionality and stability of the application, not to assess the code's adherence to stylistic conventions.

2.3.4 Klee

In our approach to leverage Klee for code analysis, we primarily focused on identifying key functions within the C source code and exploring them using Klee's symbolic execution. This involved transforming the C code into LLVM bitcode using clang, allowing Klee to work with the bitcode. By employing klee_assume, we ensured that the analysis stayed within valid input boundaries, so that infeasible paths are not explored. We let Klee run for extended periods to exhaustively analyze potentially hundreds of thousands of paths(7 hours total). This thorough exploration helped identify precise error locations in both the original C code and the corresponding LLVM assembly. Furthermore, Klee's automatic test case generation pinpointed the exact inputs leading to errors. Utilizing the ktest tool, we could examine these specific inputs, and more easily debug any violations, to try to understand if they were true positives. Klee rarely produces False Positives since the bugs are for paths that have been explored dynamically and actually lead to crashes. The most common way that they could be false positives if the user makes an error and explores invalid paths. Kilo was not explored extensively due to the need for user interaction.

3 Example of Bugs

Each static analysis tool offered a unique perspective for examining the code, uncovering a wide range of bugs, varying from basic to intricate. Our objective in this section is clear and direct: to showcase a variety of bugs that were identified, providing an illustrative snapshot of the range of issues that can be detected by these tools.

3.1 Infer - Kilo

The issue at line 384 in kilo.c is a "Null Dereference" issue which refers to a pointer that could be null being dereferenced, which can lead to a potential crash. A pointer 'row→h1' is allocated memory in line 383 using realloc(), and is provided to memset() function in line 384. Since 'row→h1' can be null, and memset() dereferences this pointer, Infer's static analysis correctly flags this as a potential issue.

Listing 1: kilo.c:382-384

```
382  void editorUpdateSyntax(erow *row) {
383      row->hl = realloc(row->hl,row->rsize);
384      memset(row->hl,HL_NORMAL,row->rsize);
```

3.2 Infer - SDS

The issue described at line 618 in sds.c is a "Dead Store" issue which refers to a value that is written into a variable but never used. Here, the developers initialise a character pointer 'f' with parameter 'fmt'. Shortly after, in line 627, the pointer 'f' is assigned the variable 'fmt', which leads Infer's static analysis to speculate as a dead value, but is in fact, not an issue (False Positive) as it is the same value here that's rewritten.

Listing 2: sds.c:616-632

```
616  sds sdscatfmt(sds s, char const *fmt, ...) {
617      size_t initlen = sdslen(s);
618      const char *f = fmt;
...      ...

627      f = fmt;    /* Next format specifier byte to process. */
628      i = initlen; /* Position of the next byte to write to dest str. */
629      while(*f) {
...          ...
```

3.3 Infer - Tinify-java

The issue described at line 25 in Tinify.java is a "Thread Safety Violation" issue which usually refers to Read/Write race in asynchronous, parallel threads, potentially causing data loss. Here, at line 25, the "client" property of the class Tinify is being read to be returned by the "client()" member function. But, in line 31, the same property "client" is written into, potentially causing a Read/Write race by a Background thread.

Listing 3: Tinify.java:13-32

```
    public static Client client() {
            ...

            return client;
        }
    }

    public static void setKey(final String key) {
        Tinify.key = key;
        client = null;
    }
```

3.4 SpotBugs - Tinify-java

The issue described at line 21 in Result.java is a ”Malicious Code Vulnerability” issue, which refers to code that can be modified to insert malicious code leading to security issues. In this line, a private final member of a class is simply returned, on call of the method toBuffer(). This, is an issue, further elaborated as ”Exposes internal representation” by SpotBugs Analyzer. It explains that here the object can be accessed to be modified freely from code outside the class, causing potential bugs. It also suggests to return a copy of the member instead of the object itself.

Listing 4: Result.java:8-22

```
  8  public class Result extends ResultMeta {
  9      private final byte[] data;
...      ...

 20      public final byte[] toBuffer() {
 21          return data;
 22      }
```

3.5 Klee - Sds:

The error in the source code file sds.c is traced back to the memcpy (line 29 of the memcpy.c file) function in the libc library . It is an out of bounds pointer error ultimately affecting the sdscatlen function. The sdscatlen function is invoked with 2 sds strings that have been created from klee symbolic values and with a len parameter that is also a symbolic value. The problem occurs in a combination where the second input string (char arrays is null and the len parameter is not zero. the erroneous line of code is :

Listing 5: sds.c:399-404

```
    sdscatlen(sds s, const void *t, size_t len) {
    size_t curlen = sdslen(s);
    s = sdsMakeRoomFor(s,len);
    if (s == NULL) return NULL;
    memcpy(s+curlen, t, len);
```

The input string that caused this error was :

```
parameter1 : b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00'
parameter2 : b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
parameter3:  b'\x02\x00\x00\x00\x00\x00\x00\x00'
parameter3 as int : 2
```

The main problem is that we try to copy to the first string a second string. The second string is of size 0 but the len provided as input is of a larger size so an out of bounds pointer error happens. To solve this issue a check should have been performed earlier to make sure that the length of the second string is equal to len (if it was lower no error would occur either).

3.6 Infer - c_hashmap

The issue at line 37 in main.c is a "Null Dereference" issue which refers to a pointer that could be null being dereferenced, which can lead to a potential crash. A pointer 'value' is allocated memory in line 35 using malloc(), and in line 37, a pointer pointed by value 'value→number' is being assigned the variable 'index'. Since 'value' can be null, and this pointer is dereferenced in line 37, Infer's static analysis correctly flags this as a potential issue.

Listing 6: main.c:35-37

```
35      value = malloc(sizeof(data_struct_t));
36      snprintf(value->key_string, KEY_MAX_LENGTH, "%s%d", KEY_PREFIX, index);
37      value->number = index;
```

3.7 Klee - Slre:

The error in slre.c, specifically within the slre_replace function, is caused by improper handling of memory operations. The inputs, including a regex pattern, buffer (buf), and substring (sub), are entirely composed of null bytes. A significant issue is the buffer length (buf_len), set symbolically to 0, leading to an integer underflow. This underflow causes the memcpy function in slre_replace to attempt copying an excessively large amount of data, indicated by the third argument in memcpy being an unusually large number (we see that in the error log). Thus an out of memory access is caused.
The input string that caused this error was :

```
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

The error line in the source code is in the libc/string/memcpy.c library called by the slre_replace function. To solve this issue a check should have been performed earlier to make sure that a null buffer is not given as input and that the buffer size is larger than 0.

3.8 Cppcheck - Slre

The bug in the slre_replace function arises from a typical mistake in realloc usage. The problem emerges when realloc fails. In the event of failure, realloc returns NULL, but crucially does not automatically free the memory initially pointed to by the variable 's'. As a result, if realloc fails, 's' is assigned NULL, leading to the loss of the original memory address. This oversight results in a memory leak, as the memory previously allocated and pointed to by 's' becomes inaccessible and cannot be freed. To circumvent this issue, a robust approach involves the use of a temporary pointer. By assigning the result of realloc to a temporary pointer, you can check for a NULL return (indicating failure) without losing the original pointer. If realloc is successful, the new memory address is transferred to 's'; if it fails, the original memory address held by 's' remains accessible for proper memory deallocation.

```
CWE:401
Common realloc mistake 's' nulled but not freed upon failure.
```

3.9 Cppcheck - Sds

In the sdsMakeRoomFor function of the sds.c file, cppcheck flagged a potential memory leak associated with the newsh variable. However, thorough manual code review and tests using AddressSanitizer, a memory error detector, no leak was detected. The function appears to properly reallocat and free memory, which AddressSanitizer's runtime detection confirmed, indicating no issues. This contrast between cppcheck's static analysis and AddressSanitizer's dynamic results, along with manual verification, suggests that cppcheck's warning is likely a false positive. The function can be seen in the Appendix A

Appendices

A Function code for sdsMakeRoomFor for repo Sds

Listing 7: sds.c:204-248

```
sds sdsMakeRoomFor(sds s, size_t addlen) {
    void *sh, *newsh;
    size_t avail = sdsavail(s);
    size_t len, newlen, reqlen;
    char type, oldtype = s[-1] & SDS_TYPE_MASK;
    int hdrlen;

    /* Return ASAP if there is enough space left. */
    if (avail >= addlen) return s;

    len = sdslen(s);
    sh = (char*)s-sdsHdrSize(oldtype);
    reqlen = newlen = (len+addlen);
    if (newlen < SDS_MAX_PREALLOC)
        newlen *= 2;
    else
        newlen += SDS_MAX_PREALLOC;

    type = sdsReqType(newlen);

    /* Don't use type 5: the user is appending to the string and type 5 is
     * not able to remember empty space, so sdsMakeRoomFor() must be called
     * at every appending operation. */
    if (type == SDS_TYPE_5) type = SDS_TYPE_8;

    hdrlen = sdsHdrSize(type);
    assert(hdrlen + newlen + 1 > reqlen); /* Catch size_t overflow */
    if (oldtype==type) {
        newsh = s_realloc(sh, hdrlen+newlen+1);
        if (newsh == NULL) return NULL;
        s = (char*)newsh+hdrlen;
    } else {
        /* Since the header size changes, need to move the string forward,
         * and can't use realloc */
        newsh = s_malloc(hdrlen+newlen+1);
        if (newsh == NULL) return NULL;
        memcpy((char*)newsh+hdrlen, s, len+1);
        s_free(sh);
        s = (char*)newsh+hdrlen;
        s[-1] = type;
        sdssetlen(s, len);
    }
    sdssetalloc(s, newlen);
    return s;
}
```