**CSEE 6863**

**mb4270 - ssd2149**

**FINAL REPORT**

**DESCRIPTION OF THE PROJECT**

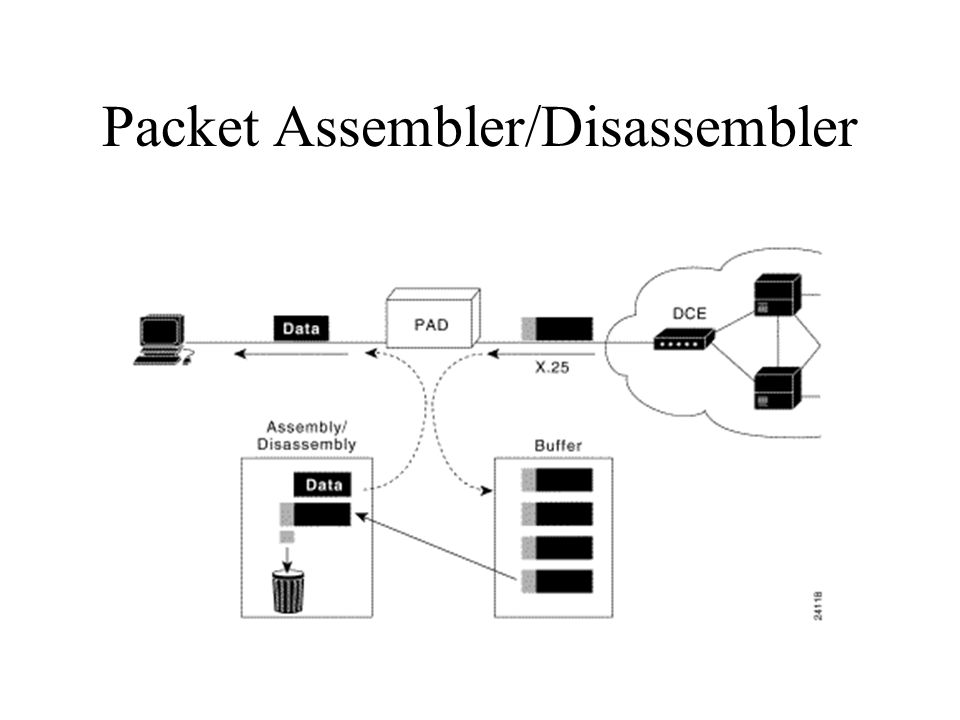
In this project, we verified a software application, named wireshark. This software is used by the Wireshark Foundation and is responsible for PPP packet disassembly.

The piece of the software that we are studying includes two files:

1. Packet-ppp.c : This file has routines for packet disassembly
2. Proto.c : This contains routines for protocol tree, with protocol being used here as P2P protocol

These protocols were used inside wireshark 1.8.0 and are written by Gerald Combs of Wireshark team. Packet-ppp uses concept of point-to-point packet disassembler. This disassembler enables a hosts to connect to packet switched networks, like IP and X.25. The application has three functions. 1. It buffers data until it is ready for processing and use. 2. It acts as an assembler, where it breaks strings of data into individual packets and adds the appropriate headers to transmit over the network. 3. It acts as a packet disassembler, where it decodes individual packets in order to obtain the appropriate data from the sender.

This code is written in C language and the code length is about 15K+ lines. The codebase also includes header files. The codebase has a couple of missing functions which we filled up for by adding function stubs, while making sure that the functionality of the codebase remained unaffected.



**WORK DISTRIBUTION**

Both of us worked on all aspects on the project. We met throughout the semester multiple times to work on every stage of the project, together.

**ISSUES**

We ran into a few issues during our development. The first biggest issue was getting our code to compile. Even though we were focusing on two classes (packet-ppp.c & proto.c), the code used by these files was spread all over the wireshark application. Many of the functions were declared externally. Hence, we needed to clean up the messy code, by adding a bunch of appropriate headers, path and directories. We created new files with “fake functions”, in order to get the code to compile and run. Even though this is not the ideal solution, the functions that we “faked” were not needed for our purposes. The code required specific flags to compile and work. This was due to the fact that certain libraries (in our case glib) tend to hide themselves in certain directories and the compiler must be told where to find them. We were able to compile the code, even though compilation happened with more than a few warnings. This was the most time consuming step of our project since it took multiple days to complete.

The next hindrance in our project was getting the clang-tidy tool to work accordingly. It is easy to install and work, however the difficult part came in integrating it with our code. The code required certain flags in order to work. The tool did not support such flags. We had to use separate methods like using compile\_databases or giving the required flags manually to the tool.

The final step in our project was to get the cbmc tool to work. Even though it is easy to run, we had to once again give it certain flags manually in order to compile. Also, we had to learn about entry points and all the possible flags needed for CBMC since we wanted to catch a particular bug.

**TOOL COMPARISON: CLANG-TIDY VS CBMC**

**CLANG-TIDY**

Clang-Tidy is a static code analyzer. It can be used to find many “simple bugs”. Clang-tidy was not going to help us find error that caused a crash in this case. It helped us find errors that made the code less “messy” and “dirty”. Clang-tidy is used as a plugin by many IDE’s in order to force developers to right better and cleaner clode. Some of the bugs were the following:

1. The code has various cases where it accesses pointers of structs without checking if the struct is valid (not null) beforehand:

2. There are cases where the code checks values in order to execute if/else statements. However, the variables never get initialized or changed.

3. There are many cases of unused variables. Such variables never get read or used.

4. There are cases where certain jump conditions are always true.



Our evaluation for the clang-tidy tool was the following:

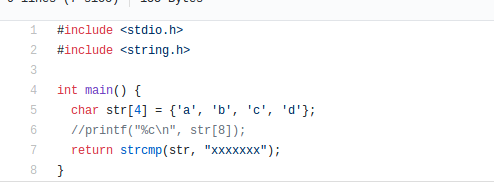
1. **Utility:** The tool reports many simple bugs. It also reports many false bugs as well. The tool is useful in finding bugs like undeclared variables, unused variables, deference to null pointers, reference to pointers without checking if they are valid etc.
2. **Performance:** The tool had very high performance. It was able to analyze more than 10k lines of code in less than 2 seconds. However, as mentioned above, it reports many false bugs, which was the biggest issue. The tool itself is not very intelligent.
3. **Easiness:** In terms of installation, the tool was easy to get running. In most linux distributions it can be found by cloning a repository and adding it as a command line argument. The tool itself is easy to run. The challenge came when integrating the tool to our code since the code requires certain compile flags (-Wall ‘pkg-config --cflags --libs glib 2.0’) to run. Those compile flags were not supported in the tool so we had to find alternative ways to compile the code and then run the static code analyzer.
4. **Features:** The most interesting feature about clang-tidy is that it doesn’t only detect the errors but can also be used to fix the errors automatically using -fix-errors flag in front of the clang-tidy command mentioned above.

**CBMC**

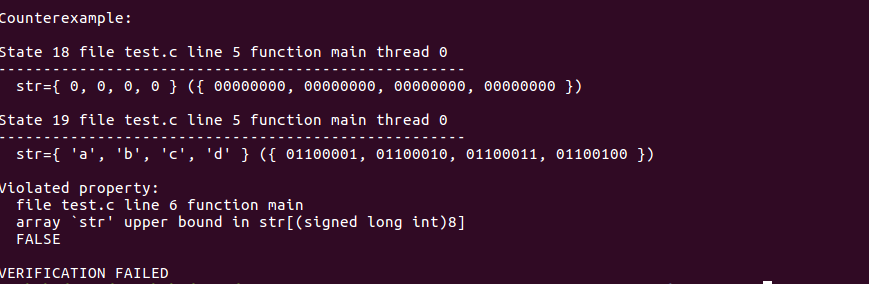
For the formal verification based approach we selected CBMC, which is a bounded model checker for C and C++ programs. CBMC supports most of the compiler extensions including gcc, which is used in this project. CBMC is widely used to check array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. CBMC unwinds all the loop it encounters and passes the resulting equation to a decision procedure. It also checks the consistency of C and C++ programs with the verilog counterparts. CBMC supports dynamic memory allocation and can be used as a command line tool or can be added to eclipse IDE as an extension. In the CBMC the transitional relation of complex state machine and its specification are jointly unwound to obtain a Boolean formula that is satisfiable if there exists an error trace in the program. This formula is then checked using SAT procedure, after converting it into CNF. SAT procedure this way can be used to generate the counterexample which is then presented to the user. Tool also checks that sufficient unwinding is done, just to make sure that longer counterexample doesn’t exist. Unwinding assertions are used for this purpose. These assertions are extremely important as they are the ones that determine if more unwinding is required or not.

1. **Utility:** CBMC takes the names of .c files as command line arguments and then translates the program and merges the function definitions from various .c files, just like a linker. It then performs symbolic simulation (procedure mentioned above). CBMC will not report “false bugs” as the static code analyzer did. It checks for specific assertions specified by the user. Another interesting feature of CBMC is it doesn’t report unnecessary warnings, which can be annoying to the end user.
2. **Performance:** The tool performance wasn’t as great as the static code analyzer. It takes a long time since it needs to add assertions after key instructions, as well as requires unwinding the loops in the code. Checking a state space that is too wide will make the tool run for too long of a time, which is a common case for many formal verification tools.
3. **Easiness:** The tool is easy to run since the tool determines the properties it needs to verify automatically. However, in order to get specific bugs, the tool must be run with specific flags.
4. **Other Features:** CBMC supports many flags as shown in the reference documents. Some of them are: *–show-claims which allows you to print properties.* --function allows you to determine entry points. --bounds-check or no-bounds-check allows you to tell the tool if you want out of bounds check or not. Many of the features like pointer checks, assertions, number of loop unwindings and others can be added as per requirement.

We ran into some issues when running CBMC. We found out that CBMC can’t detect the null termination in array of characters, although it can detect the errors related to the bounds of the array. We tried out the following sample code to verify this:



In this code, we don’t get any verification error via CBMC, even though the character array isn’t null terminated. Although, if we remove the comment at the line 6, we get the array bound error and a counterexample is generated by CBMC, as seen below:



Hence, we concluded that CBMC can’t detect if character array is null-terminated or not. However, cmbc was successful in determining the array out of bounds of our project, which was the main goal. The output is shown below:

|  |
| --- |
| file packet-ppp.c line 3142 function dissect\_ccp\_bsdcomp\_opt: array index designator 1 out of bounds (1)  CONVERSION ERROR |

**CODEBASE & BUG**

As any major technology company, the wireshark company constantly strives to improve their product. Recently they rewrote the code for their dissector functions. However, whenever they use the dissector technology, wireshark crashes completely. The application will stop working and the only working solution is to restart the application completely. It is a bug that was submitted recently so it does not seem that the developers have found the solution just yet. We used a static code analyzer and a formal verification tool in order to find what the problem that is like. Reports for the bugs and crashes can be found under the list of references. The temporary solution that was implemented in order to keep the application running was to turn off dissector features completely. Using our tools we were able to find the bug, even though we are not sure what is causing it. In packet-ppp.c, an array of pointers is declared in a static function as shown below:

|  |
| --- |
| static void dissect\_ccp\_bsdcomp\_opt(const ip\_tcp\_opt \*optp, tvbuff\_t \*tvb, int offset,  guint length, packet\_info \*pinfo , proto\_tree \*tree)  {  static const int\* vd\_fields[] = {  &hf\_ccp\_opt\_vd\_vers,  &hf\_ccp\_opt\_vd\_dict,  };  …  } |

The two values used in the array are declared globally in packet-ppp.c and their values are -1 both. However, this array of pointers is causing a buffer overflow, which causes the whole application to crash. After compiling the code, an extra garbage value gets added to the array of pointers shown above. The following print commands were added after the declaration of the array.

|  |
| --- |
| printf("%p\n", vd\_fields[0]);  printf("%p\n", vd\_fields[1]);  printf("%p\n", vd\_fields[2]);  printf("%d\n", \*vd\_fields[0]);  printf("%d\n", \*vd\_fields[1]);  printf("%d\n", \*vd\_fields[2]); |

Surprisingly, the output was the following:

|  |
| --- |
| 0x62067c  0x620680  0x620180  -1  -1  (-nil) |

After compiling, one extra pointer gets added to the array. Compiling in different machines yielded different memory locations stored in the array. However, in certain cases the value stored in the extra pointer was NULL, which is what is likely causing the application to crash. Those values are used and read later in the application by other classes. We tried many things to try to fix the issue. Setting the array size to two did not help either. Even if the array is declared with a specific size, the extra value gets added. We were surprised by this, we were expecting the compiler to give us an error but surprisingly it did not.

|  |
| --- |
| static const int\* vd\_fields[2] = {  &hf\_ccp\_opt\_vd\_vers,  &hf\_ccp\_opt\_vd\_dict,  }; |

The output was the following:

|  |
| --- |
| 0x62067c  0x620680  0x620180  -1  -1  -1 |

We do not know why the issue is being caused, since an array that is declared locally should not change or by affected by code somewhere else. We tried NULL terminating the array, reducing and increasing its size, declaring it as non static and others. In all cases, extra garbage values get appended to the end of the array. What we do know however, is that the crashes experienced by the application are being caused by this array. When the garbage value is read, it turns out to be NULL most of the times.

**USERGUIDE**

In order to recreate the project, the following commands must be run:

|  |
| --- |
| pkg-config --cflags --libs glib-2.0 |

The command above will give the locations of glib files. The output of such commands must be used to run the tools and let the compiler know of the path it needs to include. The output will look like this:

|  |
| --- |
| -I/usr/include/glib-2.0 -I/usr/lib/x86\_64-linux-gnu/glib-2.0/include -lglib-2.0 |

To compile the code:

|  |
| --- |
| gcc packet-ppp.c -Wall `pkg-config --cflags --libs glib-2.0` |

To run clang-tidy:

The output of the first command will be referred to as “output” in the following instructions:

|  |
| --- |
| clang-tidy packet-ppp.c -- output |

Alternatively, a compile database can be used to give the tool the right paths and directories.

To run cbmc:

|  |
| --- |
| cbmc -I/usr/include/glib-2.0 -I/usr/lib64/glib-2.0/include packet-ppp.c --function dissect\_ccp\_bsdcomp\_opt --bounds-check --unwind 1 |

In the case of cbmc, we added our output for the first command. This is because the last lglib string must be removed from the output of the first command, as linking isn’t permitted in CBMC. The --function flags indicates the entry point. The --bounds-check flags tells it to look for buffer overflows. The unwind flags tells the tool how many times iterations it should unwind the loop (array filling with variable size), so as to find an error. The CBMC command can be repeated with different unwind flag values to make robust bug detection.

**REFERENCES**

1. Code repository: <https://samate.nist.gov/SRD/view_testcase.php?tID=231344> (lines of code : 15k+)
2. About the PPP dissector crash bug: <https://www.wireshark.org/lists/wireshark-bugs/201305/msg00259.html>
3. How wireshark works through dissectors: <https://www.wireshark.org/docs/wsdg_html_chunked/ChapterDissection.html>
4. More about bug #8638 : <https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=8638>
5. Clang : <https://clang.llvm.org/>
6. CBMC : [http://www.cprover.org/cbmc](http://www.cprover.org/cbmc/)/
7. CPRover user manual: <http://www.cprover.org/cbmc/doc/manual.pdf>
8. Internal working of CBMC: <http://www.kroening.com/papers/tacas2004.pdf>