PatchFuzzer: Effecient Patch Testing by Coverage-Guided Fuzzing for Continuous Integration Environment

Kyuwon Cho
Department of Computer Science and Engineering, Sungkyunkwan University
Suwon, Korea
kyuwon.cho@skku.edu

Byeongsu Woo
College of Computing, Sungkyunkwan University
Suwon, Korea
wbs79@skku.edu

Youngseok Kim
College of Computing, Sungkyunkwan University
Suwon, Korea
kys00514@skku.edu

Abstract

It is important to test and find the bugs in software engineering. Developers use unit tests to find inherent bugs. However, because of the complexity of modern software, it becomes harder to test it only with unit testing. Recently, researchers focus on the fuzzing - fully automatic and scalable technique that can deal with the complex and large software. However, current researches and industrial projects only focus on single state of development. Software developers almost use the version management system like Git even during the internal period of development, but current works do not care about the information like versioning or patch contents, they only care about the final program. For this reason, the fuzzing does not fit for rapidly patched software like open-source projects. In this paper, we design new patch-aware fuzzer, the *Patchfuzzer*. Our new fuzzer leverages the patch information and adjusts the priority of code coverage so it can test the patched code with high probabilistic.

Keywords: Fuzzing, Software Engineering

1 Introduction

In modern software development lifecycle, *patch* is an important process along with fixing bugs or enhancing functionality. However, patches may introduce new bugs to the software. For example, the Heartbleed[3] occurred because of just one commit, "Support for TLS/DTLS heartbeats"[1]. This commit is implementation of TLS heartbeat on openssl, and contains 561 lines of change. But only one reviewer reviewed this commit, and made a disaster over the entire internet. This is why we need a test of a patch. Modern software developers also recognize this problem, and create new technique named CI/CD which tests the software and delivers it automatically. The testing in CI/CD inevitably relies on a unit test. Unfortunately, the unit test is made by developers manually, thus we cannot ensure that it covers all cases. There are some fuzzers like OSS-fuzz[4], integrated into CI/CD framework to test software. However, the OSS-fuzz only fuzzes the entire software, and it does not focus on a single patch. Some related works[7, 9] introduce the symbolic execution to the patch test, but symbolic execution is inappropriate for large and complex modern software because of its limitation such as inefficient constraints solving power and path explosion. So we suggest new fuzzer for the patch test.

In this project, we design the new fuzzer framework for testing the software patch. Our work shows that the patched area can be efficiently explored by coverage measurement modification. We expect that our work will be integrated in the current state-of-the-art CI/CD pipeline so it will test the patch automatically. We focus mainly on the software's functionality improvement patch, so analysing the security patches like Zhao *et al.* [11] is orthogonal to our work. We guess the patch itself contains the bug or the bug is driven by the patch. However, we can evaluate the security patch whether the security patch introduces other bugs or not.

2 Background

2.1 Fuzzing

Fuzzing is the act of providing a variety of inputs for a binary executable and observe the behaviour of the executable based on the inputs. By testing a large amount of inputs, it is possible for the developers or adversaries to find out inputs that crash or cause unintended code execution. However, there is a limit where actual human developers can make up interesting input values that might generate the wanted crash or unintended behaviour. Therefore, a fuzzing software can be utilized that will generate a various inputs that will be used for inputs to the binary executable. There are many ways to categorize fuzzers, however in this paper two major factors will be discussed. The categories are whether the fuzzer is aware of the input structure, and whether the fuzzer considers the program structure of the binary executable.

If a fuzzer is aware of the input structure, then it will generate a valid input that will generally not be rejected right away by the executable and create some kind of trace. These fuzzers are categorized as smart fuzzers, while dumb fuzzers are fuzzers that do not consider the input model. In consequence, dumb fuzzers possibly generate inputs that do

[1] https://github.com/openssl/openssl/commit/bd6941cfaa31ee8a3f8661cb98227a5cbcc0f9f3

not match the general inputs of a executable and are directly disregarded / rejected by the executable.

Another way of categorizing fuzzers is based on their consideration of the structure of the executable. If a fuzzer does not consider the structure of the executable at all, it will simply generate completely random inputs. These kind of fuzzers are called black-box fuzzers. White-box fuzzers on the other hand, carefully manufacture their input using calculations based on analysis of the program for higher code coverage. Grey-box fuzzing is a combination of white-box fuzzing and black-box fuzzing, where the fuzzer does not use heavy calculation and analysis, but rather uses comparably simple instrumentation.

The instrumentation will provide the fuzzer the execution sequence of basic blocks which can in turn be utilized for generation of inputs of higher code coverage. The main fuzzing software we will be utilizing and comparing are PatchFuzzer to is the AFL fuzzer. The AFL fuzzer is a dumb fuzzer in that it modifies the seeds provided to it for the generation of new inputs. Initially, the seed value should be a number of valid inputs for the executable for a reasonable performance. Also, AFL fuzzer is a grey-box fuzzer, the details of the instrumentation process will be provided later in this paper.

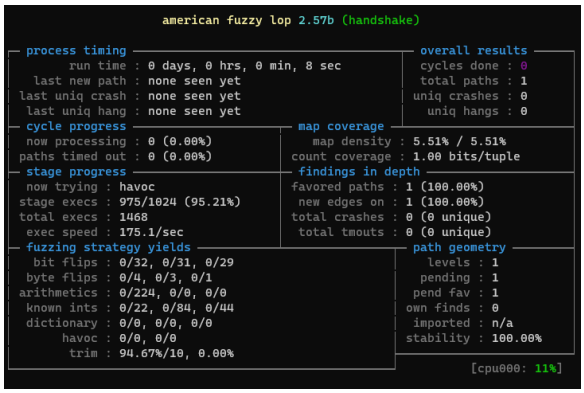

Figure 1. American Fuzzy Lop

2.2 Code-Coverage measurement in AFL

AFL records the total code area that executed. This recording action is called code coverage. AFL use this code coverage to make next input for test and find the bugs in target program efficiently. When AFL gets the higher code coverage than previous execution, it means it found new path and get a high chance to find bugs, AFL saves the input that trigger the new path and uses it to the next mutation process to generate new inputs.

2.2.1 Basic block coverage and Edge coverage. There are two methods in measuring code coverage. One is basic block coverage and the other is edge coverage. Basic block coverage records the executed basic blocks. This method is straightforward and easy to implement. However, it is hard to distinguish when the program executes same basic blocks but executes them in different order. For example, let's assume that there are three basic blocks, `A`, `B`, and `C`. In the first execution, the program executes the basic blocks in order of `A`, `B`, and `C`. In the second execution, the program executes the basic blocks in order of `A`, `C`, and `B`. Clearly, the first case and the second case are different cases. However, they executes same basic blocks. Thus, basic block coverage is hard to distinguish the two executions in the above case.

On the other hand, edge coverage records the executed edges. Edge means jump operation between two basic blocks. For example, in the above case, if a fuzzer uses edge coverage, the fuzzer records `A -> B`, and `B -> C` in the first execution and records `A -> C`, and `C -> B` in the second execution. Even though the two cases execute the same basic blocks, edge coverage can distinguish between the two cases.

2.2.2 Coverage measurement instrumentation. AFL uses edge coverage mechanism. To implement the edge coverage, AFL uses AFL compiler and bitmap. AFL compiler instrument every basic blocks by inserting coverage measurement code.

```
1 BB0:
2     %0 = load prev_loc
3     %1 = xor %0, cur_loc
4     %2 = load bitmap[%1]
5     %3 = add %2, 1
6     store bitmap[%1], %3
7     ...
```

Listing 1. Pseudo LLVM IR of AFL code coverage Instrumentation.

listing 1 shows pseudo code of instrumentation code. The instrumentation code does xor operation with ID of the previous basic block and the ID of the current basic block. The ID is random number that decided in compile time. Then, the instrumentation code increases the value in the related bitmap area. `prev_loc`, and `cur_loc` in the pseudo code are IDs of source basic block and destination basic block, not addresses of them. The ID values are determined in compile time randomly. The bitmap is a shared memory storing coverage information. Instrumentation code in the target program records coverage information in the bitmap. When the target program is terminated, AFL checks the bitmap and compares it with previous coverage information. If there are new paths in the bitmap, AFL stores the input data into the queue. In addition, AFL reuses the input data in the future fuzzing process.

2.2.3 Bitmap checking and input mutating. The numbers in the bitmap area are divided into 8 buckets, which are 1, 2, 3, 4-7, 8-15, 16-31, 32-63, 64-127, and 128+. By using this

method, AFL fuzzer can manage the change of code coverage more simply. AFL checks whether a hit time of an edge moves to another bucket. For example, if an edge was executed 101 times after in a execution sequence after the edge was previously found in another sequence 100 times, AFL will not consider the change a significant change. However, if the edge hit count is 32 and the previous maximum edge hit count was 12, then AFL will consider this change as a significant change. Mutations of initial test cases that produce new state transitions mentioned above are then added to an input queue.

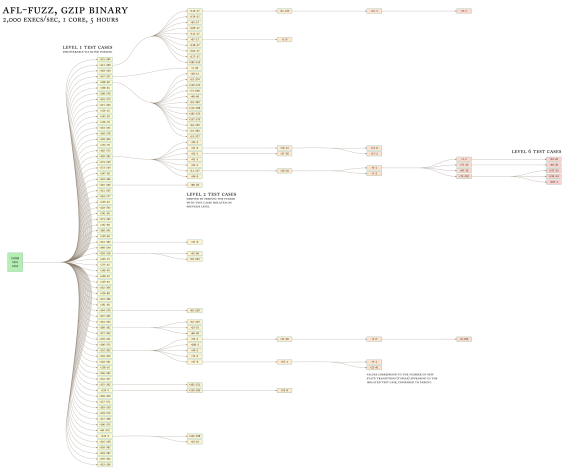

Figure 2. Example of evolution of input queue in AFL.

Mutation methods of AFL include sequential bit flips and addition / subtraction along with sequential insertion of known interesting integers such as 0 or `INTMAX`. The queue entries will be evaluated again by AFL and will possibly generate more mutated inputs that will discover previously undiscovered state transitions. These inputs will be added to the input queue as well, and this process will repeat itself.

3 Related Work

Related works to our work can be attempts such as High-Coverage Symbolic Patch Testing[9]. The ultimate goal of this work is almost identical to the work that we introduce in this paper; that is patch testing with a high code coverage. However, the procedure of this attempt differs in various ways. The overall order of High-Coverage Symbolic patch Testing (further noted as patch tester) is first, identifying a target, which would usually be an uncovered code in the patched code partition. Then, the patch tester will select some kind of seed, which is a random input, or from the program test suite, or from standard symbolic execution. The input should be able to execute instructions near to the target. Then, by going through an iteration, the patch tester manipulates the seed so that the seed value will mutate the seed value at symbolic branch points to a value in which the target instruction is executed. Of course, during this procedure inconsistent executions must be avoided by carefully analyzing the input and the branches taken. Also, it is an important factor to determine how to measure distances between instructions. The patch tester uses a context sensitive, path insensitive static analyzer to calculate the distance. Another interesting point about this patch tester is that the tester only keeps tracks of the last patch explored, focusing on simplicity. The key difference this patch tester has with our implementation is that our implementation tries to get to the patched code using fuzzing while calculating and modifying weights in edges or added basic blocks. This makes our implementation more complex and heavy, but also may have results that are accurate and superior. Symbolic execution may have limitations in programs with excessive size of complicated branches where it may take up a substantial amount of unnecessary time, thus our implementation is expected to be a faster and more efficient approach to software patches that have limitations for using symbolic execution.

Other attempts that aim for the same goal as our work but implemented in different ways exist as well. One can be Coverage Based Test Suite Augmentation[2], which is quite similar to the patch tester describe above. The difference is in the implementation part where it creates requirements for test cases to perform proper execution of target code, instead of using simple coverage metrics as the patch tester does. The Coverage Based Test Suite is different from our implementation similarly as the patch tester does from our implementation. It relies on symbolic execution while it does not utilize fuzzing of any sort. This can lead to computational overhead, as well as the patch tester mention above. Due to this reason, our implementation is expected to be a more safer faster way of covering patched code, over other similar implementations.

Another related work to our work is Shadow Symbolic Execution[7]. The main purpose of this paper is to generate inputs that can trigger the new behaviours caused by a patch. For example, if the condition is changed from 'old' to 'new', there can be execution divergence behaviour or possible divergence behaviour on same executions. While various definitions of behaviour are possible, it uses a generally applicable definition of behaviour at the code-level. In the first step of the author's approach, the author annotate the patches to unify the old and the new version. After integrating two versions into a singular version, the shadow symbolic execution operates in two phases for each input that touches the patch. The first phase is concolic phase that can gather divergence points after this phase is done. The second phase is bounded symbolic execution phase to search for additional divergent behaviours on the divergence point. However, there is a clear limitation since it only makes possible inputs based on the static analysis on the code-level. Whereas, our PatchFuzzer approaches patch testing with

fuzzing making it possible to discover the bugs caused by patch as well as the inputs triggering difference behaviour.

4 PatchFuzzer

4.1 Project Goal and Assumption

Project Goal The goal of this project is finding the bugs inside the software patch. The result can be any type of bug, but bug-detecting techniques are not our interests. We totally rely on state-of-the-art bug detection techniques like *ASAN*[10], *TypeSAN*[5], *UBSAN*[8] and others. Yuseok *et al.* [6] shows that optimized sanitizers can improve the fuzzing performance, but we use the traditional sanitizer because those optimization are not for coverage, but only for the performance.

Assumption We assume that the target application is an open-source project and is well managed by Software Configuration Management (SCM) like Git so it is easy to get the patch information such as modified function name or line of code. We also assume that the target application is written in C/C++, because C/C++ produces simpler structure of Intermediate Representation (IR) than other languages. Our methodology can be used for other LLVM-compilable languages like Golang or Rust. However, the languages that have no LLVM frontend or have the LLVM frontend but the target project cannot support that frontend, will not be compatible for the Patchfuzzer.

4.2 Design

The overall structure of PatchFuzzer contains three major phases. It instruments the target program with a slightly modified AFL compiler pass in the first phase(section 4.3). The second phase is finding difference in between the patched code and the original code. In order to find the difference of the codes, we obtained the LLVM Module instance of both the original function and the patched function. This can be done by producing the bitcode of both the patched and original source codes with LLVM, then parsing them and converting them to the LLVM Module instances using LLVM library functions. Then, by comparing the modules of the two sources, the Patchfuzzer will find the difference in the source and patched source file. In the last phase of PatchFuzzer, the block weight will be adjusted. For different elements of the two modules, the weight will be modified while the other identical elements in the modules remain unchanged(section 4.5).

4.3 Coverage Instrumentation

As mentioned in listing 1, AFL assigns unique block IDs to each basic block. However, this block ID is changed in every compilation because the value is chosen randomly. Consequentially, this randomness will hurt our next step, differential finding. The reason behind this is because our framework will decide two blocks are different when two blocks have different block IDs, even if they have the exact same behaviour. To prevent this situation, we modify the afl-clang pass so that it inserts a fixed value as a block identifier initially, then we change it to a random value again in our framework.

4.4 Differential Finding

In this section, we describe the tools that find the difference between two basic blocks. In patch files, the names of modified function are usually included. So our tool leverages the **FunctionComparator**, which is a class included in LLVM. Unfortunately, **FunctionComparator** only provides the result if the behaviour of two functions is same or not, so we inherit and generate a new class named **DiffMarker**. Our tool marks entire blocks in function that are generated in a patch. For modified functions, **DiffMarker** class takes two functions. One function is the original function, and the other function is a patched function. We want to compare two basic blocks and see whether they are same or not. Fortunately, **FunctionComparator** provides a method named *cmpBasicBlocks*, which does the exact work we needed. We just used that method for the first implementation step, but we found that it is imprecise in that they generate false negatives, which means that two basic blocks are the same but the method evaluates those as different. For the reason, we add an additional check if the *cmpBasicBlocks* has failed. In current implementation, we convert the LLVM IR instruction to string and compare them. We will describe the limitation of this technique in section 7.1. To detect the modification between two functions, we use brute-force algorithm to compare the entire basic blocks in patched and original code.

```
patched = {}

search(f_origin f_patched):
    for BBp in f_patched:
        bool flag = false;
        for BBo in f_origin:
            if(cmpBasicBlock(BBp, BBo) == true)
                flag = true;
                break;

        if flag == false:
            // The basic block does not belong to the
                original function
            // It means that this block is patched.
            patched.push(BBp);
```

Listing 2. Algorithm for comparison of two functions

Because of huge size of the target binary, it is infeasible to compare the entire basic blocks by using brute-force algorithm. However, we compare the basic blocks per the function granularity, so the brute-force is also feasible and makes the comparison in time. We loop the basic blocks in

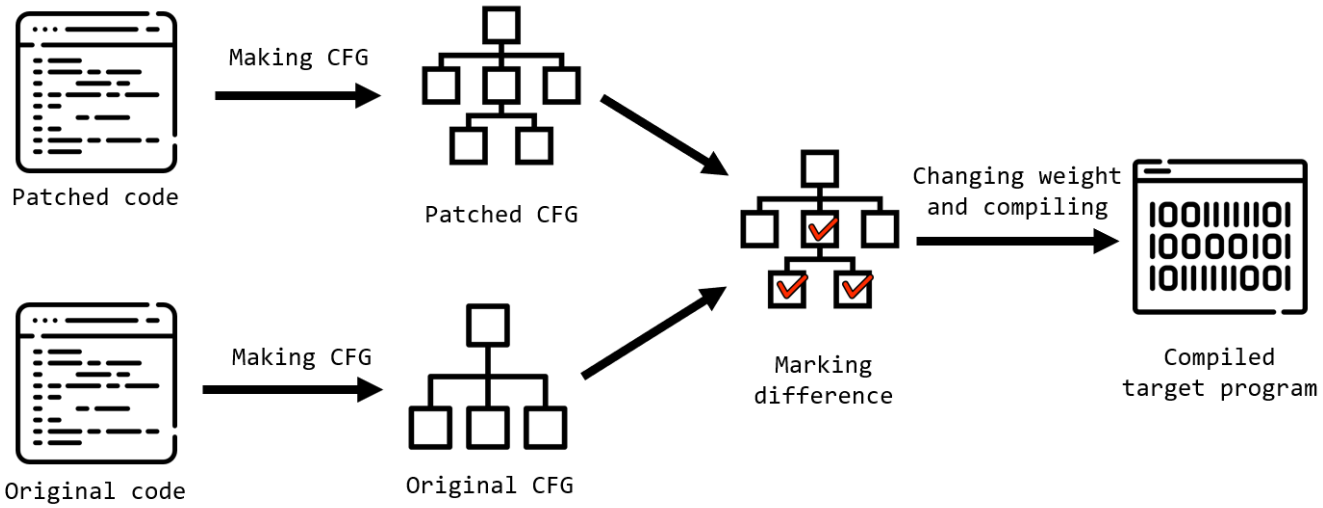

Figure 3. Overall architecture of PatchFuzzer

the patched function, and compare it to entire blocks in the original function. If the two blocks match, we break the loop because it means the contents of the block are same, and it means there is no patch on this block. However, some blocks may have no pair block in the original function. It means that the blocks are newly introduced by some patches, so we add this block to patched basic block list.

After we find the modified basic blocks, we mark that these blocks are more important than the other blocks. We try to change the coverage weight as will be described in next section(section 4.5). To achieve this, we need to find the bitmap incremental instruction in basic block. However, AFL instrumentation is always inserted in the top of the basic block, so we can assume that the first add instruction is for bitmap incremental instruction. So we find it and change the operand to mark the modified block.

4.5 Code Coverage Modification

We explain the AFL code coverage instrumentation in section 2.2. As listing 1 describes, in the entry of the basic blocks, the bitmap is increased by 1 in line 5. We define the increased value as *weight of block*. We guess that if we change the weight of basic blocks, the fuzzer will get better coverage for the changed basic block area. For example, if we change the weight of specific basic block 1 to 2, the fuzzer will get the tendency to hit that block. Based on this assumption, we change the weights of patched basic block that were found in section 4.4 and measure the coverage of those blocks. After trying some various weight modification, we found out that the current AFL fuzzer gets better coverage on the smaller block weights. So we set the common weight of basic block as 2, and decrease the weight of the patched area as 1. We will show the various weight modification in section 6.

5 Implementation

5.1 Instrumenting Whole Program

To build the common program by AFL is easy, just by using the afl-clang as the compiler is enough. However, for our framework, it is much more complicated because we need to analyze the patch, so it couldn't be done in compiler pass. So we created a new standalone program using the LLVM library and feed the program an original bitcode file and the patched bitcode file. Generating the bitcode of single C file is an easy task, but in large modern software, generating a bitcode of a single C file, modifying it, then feeding it to the final compilation step is a hard task without the compiler pass. So we use *wllvm*[2], which creates the bitcode of the whole program, not a single C file. Using this tool, we can get the whole program's bitcode including the static linked binary so we can use that bitcode as input of our framework.

5.2 Patch Tracking

To find the differential between previous code and patched code without any patch information is inefficient and not feasible because of the large the bitcode size. We use the patch information to extract which functions are modified and added or deleted.

```
1 diff --git a/ssl/s3_clnt.c b/ssl/s3_clnt.c
2 index 8874ce8818..ff1cff23d0 100644
3 --- a/ssl/s3_clnt.c
```

[2]https://github.com/travitch/whole-program-llvm

```
+++ b/ssl/s3_clnt.c
@@ -203,6 +203,18 @@ int ssl3_connect(SSL *s)
    s->in_handshake++;
    if (!SSL_in_init(s)  SSL_in_before(s))
        SSL_clear(s);

+#ifndef OPENSSL_NO_HEARTBEATS
+ /* If we're awaiting a HeartbeatResponse, pretend we
+ * already got and don't await it anymore, because
+ * Heartbeats don't make sense during handshakes anyway.
+ */
+ if (s->tlsext_hb_pending)
+  {
+  s->tlsext_hb_pending = 0;
+  s->tlsext_hb_seq++;
+  }
+#endif
+
    for (;;)
       {
       state=s->state;

diff --git a/apps/s_client.c b/apps/s_client.c
index b72e505fb1..b0702ce6fe 100644
--- a/apps/s_client.c
+++ b/apps/s_client.c
@@ -1862,6 +1862,14 @@ printf("read=%d pending=%d peek=%d\n",k,SSL_pending(con),SSL_peek(con,zbuf,
               SSL_renegotiate(con);
               cbuf_len=0;
               }
+#ifndef OPENSSL_NO_HEARTBEATS
+   else if ((!c_ign_eof) && (cbuf[0] == 'B'))
+     {
+    BIO_printf(bio_err,"HEARTBEATING\n");
+    SSL_heartbeat(con);
+    cbuf_len=0;
+    }
+#endif
```

Listing 3. Part of git patch contents for TLS heartbeat protocol support in openssl

Git provides a strong patch generation tool, *format-patch*. This tool generates the differential between two commits including modified contents, filename, commit hash and function name. In listing 3 line 5, the modified function name is followed after the second @@ with the signature and arguments name. We can parse this patch file with regular expression to get a function name. However, sometimes the format-patch miss to get a actual function name, and put an odd value in the function name place like line 29. For this case, we will aggressively compare the entire functions in that file. This is acceptable because a lot of modern software manage each file has only one functionality, so there is not that many functions in a single file. However, updating comments like copyright year or updating the include header generates a lot of false positives so we filter those cases. We will show the total number of functions aggressively compared in section 6.1

6 Evaluation

6.1 Number of Functions Patch Parser Process

	On-place	Overestimated
libpng	6	499
libxml	9	500
coreutils	3	443
openssl	53	757
qemu	111	1447
linux-5.18.0	47	1821

Table 1. Number of functions that our patch parser find out from the patch file.

Git format-patch sometimes does not catch the patched function name, so we aggressively estimate that entire functions in that client should be compared. In this section, we will show the actual number of functions that are overestimated by this logic. To get the number, we select the open-source projects and generate the patch file for recent 100 commits by format-patch tool and run our parser to evaluate the number of change. In table 1, on-place means that the format-patch correctly catch the function name, so we do not need aggressive logic while overestimated means that the format-patch does not get the function name so it counts the functions in the file. The result show that format-patch is not good for catching the function name. In 100 commits, format-patch only catch the 3 functions in coreutils. However, the aggressively overestimated functions are also not that many. Linux kernel has maximum number of overestimated functions, but it is 18 functions per one commit in average. It is not that big number and Patchfuzzer can easily deal with that small number of functions.

6.2 Weight Adjustment

We conducted experiments to figure out which weight makes the fuzzer cover the patched region faster. AFL-training[3] is the repository that provides the training course about the AFL usage. The repository includes some challenges about trying to write the harness and run the fuzzer to find a bug in a real-world software like openssl, the SSL library or libXML,

[3] https://github.com/mykter/afl-training

the XML parser. We used the heartbleed vulnerability example in openssl for our experiment because the heartbleed is one of the most representative examples of vulnerable patch.

OS	Ubuntu 20.04
CPU	Intel(R) Core(TM) i7-8550U
Memory	DDR4 2666MHz 20GB

Table 2. Experiment Setup

We did checkout to the commit implementing heartbeat protocol which also introduced heartbleed. We built the openssl library with AFL instrumentation compiler and *ASAN* then wrote and compiled the harness program in the challenges directory to fuzz the openssl library. Among the patched part, the vulnerable function is *tls1_process_heartbeat*.

6.2.1 First Experiment: Comparing the Time until Finding Crash. First of all, we compared the time spent to find the crash between harness with baseline openssl library and modified library. The modified library is the library whose weights of patched area were adjusted to 2, the baseline is 1. We fuzzed 100 times on each harness to get relatively reliable experimental result because the mutation process in fuzzer has randomness. We assumed that comparing the time spent to find the crash is fine because the vulnerable part is in patched area. We additionally thought the time is short meaning that the fuzzer covered the patched area preferentially. However, we additionally guessed that the patched harness covered the patched area preferentially even though the time spent to find the crash is slow. This means fuzzer covered the patched area first, but the covered area was not a vulnerable area. In this case, even though fuzzer covers the patched area as we want, the time spent to find the crash is slow. Thus, we conducted another experiments using another patched harness. The second patched harness is the harness whose weights of only the vulnerable function `tls1_process_heartbeat` were adjusted to 2. The table 3 shows the result of the experiment.

Harness	Baseline	Modified	Vulnerable within modified
Average	142.64	148.19	140.68
Standard deviation	70.94	71.34	94.52
Top 10%	89	88	86
Top 90%	246	255	233

Unit: seconds

Table 3. Experiment I. **Baseline** is harness compiled with unmodified AFL compiler. **Modified** is the library whose weights of patched area are 2. **Vulnerable with modified** is the library whose weights of vulnerable function are 2. **Top 10%** shows the first decile of the result and **Top 90%** shows the ninth decile of the result.

As a result, **Vulnerable with modified** is faster than **Baseline**. However, **Modified** is slower than **Baseline**. Standard deviations of the result are big because there are some random factors in the input generation process of the fuzzing such as mutation methods, and mutated values.

However, the reduced average time is only 2 seconds in the experiment I. We additionally conduct another experiment to reduce time more than 2 seconds. In the experiment II, we adjusted the weights to 8 to figure out whether the time consumption reduces or not. The table 4 shows the result of the another experiment.

Harness	Weight 8	Vulnerable within weight 8
Average	157.21	150.31
Standard deviation	99.50	84.57
Top 10%	93	89
Top 90%	259	241

Unit: seconds

Table 4. Experiment II. **Weight 8** is the harness whose weights of patched area are 8. **Vulnerable within weight 8** is the harness whose weights of vulnerable function are 8. **Top 10%** shows the first decile of the result and **Top 90%** shows the ninth decile of the result.

As we expected, **Vulnerable within weight 8** is faster than **Weight 8**. However, the results are slower overall than the experiment I. We can figure out that the higher weight does not ensure the faster fuzzing.

From the experiment I, we can figure out the change of weight value affects the code coverage of fuzzer. However we cannot ensure what values make the fuzzer cover the patched region faster than original harness. This is because, the fuzzer took more time to find a crash when we marked all of the patched regions. And also there was no meaningful difference when the fuzzer executed the harness marked only vulnerable region even though it took less time than original harness. Thus, we concluded that we should conduct more sophisticated experiment.

6.2.2 Second Experiment: Comparing the Hit Time. To get more accurate results, we modified the instrumentation code of AFL compiler. We made another shared memory region to store the hit count of patched regions. And we inserted the recording function call to the beginning of the instrumentation code in the patched basic blocks. The recording function in the patched area will increase the hit counter of patched region stored in the shared memory region. The shared memory stores the overall accumulated hit count value during the entire fuzzing process. The fuzzer stores the hit count to a file in the output directory and updates the accumulated hit counts on every 200 milliseconds.

The fuzzer starts recording the hit counts when the fuzzer executes the patched area first time. This means, when the

hit count is zero, the fuzzer do not record the value to the output file. Because the time consumed to find patched area is realm of luck and out of our interest. All we want to see is just how fast the patched area is covered after the first time that the fuzzer executes the patched regions. We fuzzed 100 times for each cases and calculated the average of results in every 200 millisecond to overcome the randomness of fuzzing.

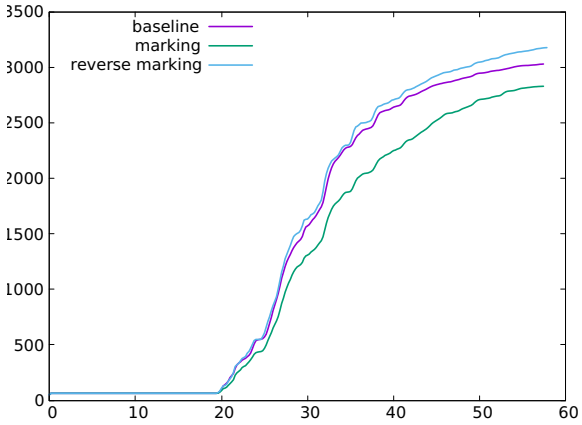

Figure 4. The average hit count of patched basic blocks region. X axis shows the elapsed time in seconds, Y axis shows the accumulated hit times of patched area

fig. 4 shows the result of second experiment. We measured the hit count of patched region up to 60 seconds after the first hit time. We compared the hit count of the patched region when the weight of the patched region is 1 as normal cases(*baseline*), when the weight of patched region is 2 and the weight of others is 1(*marking*), and when the weight of patched region is 1 and the weight of the others is 2(*reverse marking*).

The experiment shows that the hit count is highest when we fuzz the reverse marking binary and the hit count is lowest when we fuzz the marking binary. More specifically, after 60 seconds, *reverse marking* hit 4% more than baseline, and *marking* hit 7% less than baseline. The gap is not that big, but it is because the patched area is not that big and the evaluation time is short. We expect the gap will increase in complex patch with enough time period.

We assume that the fuzzer covers the region having higher weight faster than the region having lower weight. However, the experiment shows the opposite result. We can interpret the result of first experiment by conducting the second experiment. The increased time at the patched case in table 3 means that the fuzzer executes the patched region less than the other regions, not visiting some other patched region that does not have vulnerability. Even though the experiment gave us the opposite result of our expectation, we found the methodology to execute the patched region faster than the original fuzzer.

7 Discussion

7.1 Limitation

Basic Block Comparison. As described in section 4.4, our basic block comparison technique relies heavily on the LLVM. But in some cases, two blocks are completely same but LLVM overestimates causing generations of false negatives. To enhance the accuracy, we introduce the comparison of the IR string, but that method is very fragile. For example, just the variable addition on other basic blocks can lead the IR difference, because the virtual register numbers are changed. So in that case, our IR comparing method also generates the false negative, because the variable is introduced in other block, so the compared behaviour of blocks are completely same. To prevent this, we have to analyze the logic of basic block and compare it. However, comparing two basic blocks and finding out those are same or not can be done without heuristic way, as Francis *et al.* [1] shown. In this project, the heuristics are very simple, but we can improve this part by using other heuristics.

Patch on Out-of-Function Area. Our work only care the code-modified patch in function. However, some patch only change out-of-function, like global variable value or static variable value, and even not-code modification like configure. In this case, it cannot detect by the deferential finding step, and even it can be detected, we cannot instrument them. However, the patch that changes global or static variables may changes the usages of that variable in the function with high probabilistic, so we can expect that the bugs driven from the change of variable can be detected by our framework. But we have no chance to detect the out-of-source patch like configure.

7.2 Future Work

Analyzing the AFL Mutation and Input Generation Algorithm. We found that the blocks with lower weight have higher chance to be executed in fuzzing process. However, we just found this for the experiment and cannot explain about the fundamental reason of the phenomenon. The observed situation that lower weight value indicates high priority is contrast to our intuitive so we need the explanation about the result. We will analyze the internal algorithm of the fuzzer that manages the mutation of the input and controls the input queue to find out the reason of the result.

Using Hybrid Fuzzing with Symbolic Execution. Symbolic execution is efficient analysis method to solve the branch condition and get a high coverage. But because of path explode and high overhead of constraints solving, it is not feasible for complex modern software development. For that reason, researchers do the hybrid fuzzing, run the fuzzer in base and make a symbolic execution when the fuzzer stuck on the complex branch conditions. In our work,

we only rely on the fuzzer feedback mechanism to guide the flow to the patched area. However, with the hybrid fuzzing and symbolic execution, we can easily generate the initial seed to hit the patched area and solve the complex branch conditions.

Combining with State-of-the-Art CI Pipeline. In our PoC and evaluation, we make a standalone program to run a analysis and fuzzing in local environment. However, our final goal is run the framework in the Continuous Integration (CI) pipeline to provide a efficient testing to the opensource ecosystem. To achieve that goal, implementation of the patch parsing for the CI environment and getting the original code and patched code is needed. We can implement those things for Github Action, Jenkins or Gitlab CI for our future work.

8 Conclusion

In this project, we implemented PatchFuzzer which covers the patched area faster than the state-of-the-art fuzzer. In the differential finding part, we discovered the differences in the source codes using the LLVM analysis library. We introduced the methodology for detecting differences, analyzing the patch file to find a patched functions and comparing the basic blocks of the functions. In the code coverage modification part, we found that lower weight basic blocks have been searched with more probabilistic. We evaluated the PatchFuzzer with openssl and real-world vulnerability example and showed that it is efficient than the state-of-the-art fuzzer, AFL.

References

[1] Francis Adkins, Luke Jones, Martin Carlisle, and Jason Upchurch. Heuristic malware detection via basic block comparison. In *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, pages 11–18, 2013.

[2] Prabhneet Nayyar Bharti Suri. Coverage based test suite augmentation techniques-a survey. *International Journal of Advances in Engineering Technology*, may 2011.

[3] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery.

[4] Google. Oss-fuzz. https://github.com/google/oss-fuzz, 2022. Last accessed Apr 29 , 2022,.

[5] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 517–528, 2016.

[6] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient sanitizer metadata design for fuzzing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 249–263. USENIX Association, July 2020.

[7] Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. Shadow symbolic execution for testing software patches. *ACM Trans. Softw. Eng. Methodol.*, 27(3), sep 2018.

[8] LLVM. Undefinedbehaviorsanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2022. Last accessed Apr 29 , 2022,.

[9] Paul Dan Marinescu and Cristian Cadar. High-coverage symbolic patch testing. In Alastair Donaldson and David Parker, editors, *Model Checking Software*, pages 7–21, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[10] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[11] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. *PatchScope: Memory Object Centric Patch Diffing*, page 149–165. Association for Computing Machinery, New York, NY, USA, 2020.

A Complexity of the Patch

```
From bd6941cfaa31ee8a3f8661cb98227a5cbcc0f9f3 Mon Sep 17 00:00:00 2001
From: "Dr. Stephen Henson" <steve@openssl.org>
Date: Sat, 31 Dec 2011 23:00:36 +0000
Subject: [PATCH] PR: 2658 Submitted by: Robin Seggelmann
 <seggelmann@fh-muenster.de> Reviewed by: steve

Support for TLS/DTLS heartbeats.

---
 20 files changed, 561 insertions(+), 4 deletions(-)

...
 diff --git a/ssl/d1_both.c b/ssl/d1_both.c
index 89338e9430..0a84f95711 100644
--- a/ssl/d1_both.c
+++ b/ssl/d1_both.c
@@ -1084,7 +1084,11 @@ int dtls1_read_failed(SSL *s, int code)
...
@@ -1084,7 +1084,11 @@ int dtls1_read_failed(SSL *s, int code)
...
+ if (hbtype == TLS1_HB_REQUEST)
+  {
+  unsigned char *buffer, *bp;
+  int r;
+
+  /* Allocate memory for the response, size is 1 bytes
+  * message type, plus 2 bytes payload length, plus
+  * payload, plus padding
+  */
+  buffer = OPENSSL_malloc(1 + 2 + payload + padding);
+  bp = buffer;
+
+  /* Enter response type, length and copy payload */
```

```
+  *bp++ = TLS1_HB_RESPONSE;
+  s2n(payload, bp);
+  memcpy(bp, pl, payload);
+  r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT,
     buffer, 3 + payload + padding);
...

diff --git a/ssl/t1_lib.c b/ssl/t1_lib.c
index 32f99013ad..01e8fc9c68 100644
--- a/ssl/t1_lib.c
+++ b/ssl/t1_lib.c
@@ -114,6 +114,7 @@
...
@@ -618,6 +619,20 @@ unsigned char
     *ssl_add_clienthello_tlsext(SSL *s, unsigned
     char *p, unsigned cha
           i2d_X509_EXTENSIONS(s->tlsext_ocsp_exts,
               &ret);
...
@@ -796,6 +811,20 @@ unsigned char
     *ssl_add_serverhello_tlsext(SSL *s, unsigned
     char *p, unsigned cha
...
@@ -840,6 +869,11 @@ int
     ssl_parse_clienthello_tlsext(SSL *s, unsigned
     char **p, unsigned char *d, in
...
@@ -1227,6 +1261,21 @@ int
     ssl_parse_clienthello_tlsext(SSL *s, unsigned
     char **p, unsigned char *d, in
...
@@ -1312,6 +1361,11 @@ int
     ssl_parse_serverhello_tlsext(SSL *s, unsigned
     char **p, unsigned char *d, in
...
@@ -1478,6 +1532,21 @@ int
     ssl_parse_serverhello_tlsext(SSL *s, unsigned
     char **p, unsigned char *d, in
...

@@ -2330,3 +2399,145 @@ int tls1_process_sigalgs(SSL
     *s, const unsigned char *data, int dsize)
+ if (hbtype == TLS1_HB_REQUEST)
+  {
+  unsigned char *buffer, *bp;
+  int r;
+
+  /* Allocate memory for the response, size is 1
     byte
+  * message type, plus 2 bytes payload length, plus
+  * payload, plus padding
+  */
+  buffer = OPENSSL_malloc(1 + 2 + payload +
     padding);
+  bp = buffer;
+
+  /* Enter response type, length and copy payload */
+  *bp++ = TLS1_HB_RESPONSE;
+  s2n(payload, bp);
+  memcpy(bp, pl, payload);
+  /* Random padding */
+  RAND_pseudo_bytes(p, padding);
+
+  r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT,
     buffer, 3 + payload + padding);
...
```

Listing 4. Git patch contents for TLS heartbeat protocol support in openssl. The total patch size is too large(almost 1000 line) and complex, so reviewer couldn't find the critical bug in line 36 and 76.