AutoPatch: Automated Vulnerable Code Patching with AFL, ASan, and GPT

CS 487 Final Project Report

Simone Bevilacqua, Bridget Danaher, Lily Leith

December 6, 2024

Introduction

Fuzzing and address sanitization are useful tools for detecting and identifying bugs. However, manually parsing the output from these tools and creating patches can be time consuming and complex, especially when new patches reap new bugs. We introduce AutoPatch, a method for automated patching and testing of C code using a memory error detector, a fuzzer, and a large language model (LLM).

AutoPatch Goals

1. Bug Descriptions

To describe the goals of AutoPatch we first describe the types of errors we are interested in:

Syntactic errors: these errors violate the language rules and prevent compilation. Some examples are typos and missing semicolons. Syntactic errors are discovered at compilation time.

Runtime errors: these errors do not prevent compilation, and instead are discovered at runtime and cause the program to crash if not handled correctly. When input is required by the program, it's possible for variations in input to cause runtime errors, such as divide by zero when the user specifies the divisor. The use of a fuzzer helps expose these runtime errors by feeding the program diverse input.

Semantic errors: these errors do not violate the language rules, and may not prevent compilation or cause runtime crashes. They encompass errors in the logic of the program, and result in incorrect results. Examples of semantic errors include the use of incorrect variables for computations and incorrect conditionals. These are the most difficult bugs to patch because *(1)* they are not flagged by the fuzzer, *(2)* an attempt to patch more complex

semantic errors them requires knowing the purpose of the program, and *(3)* patch attempts may incorrectly change the logic of the program in an undesirable way.

2. Bugs and Our Metrics

The goal of AutoPatch is to detect and patch syntactic errors, runtime errors, and make the best attempt at patching semantic (logical) errors in C code. To deem a patch successful, AutoPatch requires that there are no syntactic or runtime errors. The detection of semantic errors is more nuanced, and requires human analysis of the patched code.

Tools

1. Address Sanitizer

We chose to use Google's Address Sanitizer (ASan). ASan can detect a variety of memory-specific errors in C, including use after free, heap overflow, and stack overflow. ASan is used by compiling source code with the **-fsanitize=address** flag, which generates a log of errors or warnings caught by ASan, and compiles the code into an executable if no compilation errors exist.

2. American Fuzzy Lop

For the fuzzer, we chose to use Google's American Fuzzy Lop (AFL). AFL provides coverage-based fuzzing to generate runtime crashes in all reachable areas of code, and can be configured for use on multiple operating systems. AFL is used by compiling source code with the AFL compiler, followed by running this code with a specified input file. During runtime, the input file seed is mutated to attempt to reach all possible control flows of the program. Crashes are logged with the crash type and the input that caused the crash.

3. GPT-4o mini

We chose OpenAI's GPT-4o mini as the LLM for AutoPatch. GPT-4o mini is adept at contextualizing errors in code, along with being lightweight and more cost effective than other previous models. In AutoPatch, we prompt GPT to create a patch for buggy code using the output information from ASan and AFL. The exact wording of the prompt is crucial for a successful patch. We build the following prompt to direct GPT's patch to align with the goals of the project based on the following conditions:

- **Always Included (beginning):** "Here's a piece of code: (insert code)."
- **If the code is syntatically correct:** "The sanitizer detected these issues: (insert sanitizer output)"
- **If crashes have been found:** "The fuzzer detected some crashes, here are some inputs that caused the crashes: (insert crashes list)"

- **If the Fuzzer halted before it finished executing:** "The code crashes regardless of the input it is given."
- **Always Included (end):** Please provide a patch to fix this issue. Don't change the meaning at all, keep it simple, don't add any comments and solve the issues in the easiest possible way"

AutoPatch Methodology

The control flow of the AutoPatch program is as follows, assuming a buggy target program:

1. Buggy code is compiled using ASan.
   (a) If code does not successfully compile, prompt GPT to generate a patch using the source code and the errors from ASan. Return to step 1 with the patched code as the new target.
2. Compile the program with AFL.
3. Run the program with AFL for a specified number of seconds.
   (a) If crashes are logged by AFL, then prompt GPT to generate a patch using the source code, the ASan log, and the crash information from AFL. Return to step 1 with the patched code as the new target.
   (b) If there are no crashes logged, then the code is bug-free.

Some changes exist between the flow of original source code testing and patched code testing:

- If the code being tested is the original source code, AFL runs for 30 seconds. If the code is a GPT-generated patch, AFL runs for 30, 60, and 120 seconds to ensure maximal code coverage before approving the patch.
- The program allows three attempts at patching code with the above steps before declaring failure. We believe that if a successful patch is not generated after three attempts, it is unlikely that GPT will succeed on subsequent attempts.

Testing

To test AutoPatch, we compiled a codebase of buggy C code that encompasses the following types of bugs:

- Stack overflow
- Heap overflow
- Use after free
- Control flow hijacking

- Format String

Each buggy source code is analyzed using the process described above.

Evaluation

To evaluate AutoPatch, we calculated the success rate for patching syntactic and runtime errors and for semantic errors.

1. Syntactic and Runtime Errors

AutoPatch has a 100% success rate on patching syntactic and runtime errors found by ASan and AFL. However, for some source codes, the input needed to cause a crash is extremely specific and would require extensive mutations of the input. These kinds of input include the memory addresses of various pointers on the stack, along with calculations needing to be made to determine buffer sizes and things of that nature. Without being able to guide the fuzzer to take specific input that is close to crash-causing input, it is extremely intensive/not possible with our constraints to generate these inputs.

2. Semantic Errors

AutoPatch's attempts to correct semantic errors vary in their success. For example, use after free errors are corrected if the program is using the free'd variable for a purpose after the free, but is not corrected if, say, the address of the pointer is printed after the free. Thus, like mentioned above, human intervention is needed to determine if the LLM was able to patch the semantic issue, and quantifying a metric of success would be inaccurate due to it being subjected by the programmer.

Next Steps

Unfortunately, we weren't to maximize the potential of AutoPatch due to the timeline constraints of this project, but we have multiple ideas for ways to expand the capabilities of AutoPatch that we thought were worth noting.

1. Improvements and Goals

Here we elaborate on ways to improve and continue this work.

- Expanding the language base: patching code in other languages, such as Python, Java, or C++.
- Increasing the complexity and length of the codebase files.
- Implementing the ability to handle multiple linked files with complex dependencies.
- Scaling the program by isolating the buggy section of code and sending only this part of a large program to the LLM, instead of the entire program.

2. Final Thoughts

The goal of automatically patching code is well within reach for simple programs, as shown by AutoPatch. This tool could be useful for students debugging their code, or teachers aiming to teach students simultaneously about vulnerabilities and fuzzing. Further developments, such as applying these practices to low-level software will require more complex handling.