Google

Google Security Engineering Technical Report
31 January, 2024

AI-powered patching: the future of automated vulnerability fixes

Jan Nowakowski, Jan Keller
jnowakowski@google.com, jakl@google.com

As AI continues to advance at rapid speed, so has its ability to unearth hidden security vulnerabilities in all types of software. Every bug uncovered is an opportunity to patch and strengthen code—but as detection continues to improve, we need to be prepared with new automated solutions that bolster our ability to fix those bugs. That's why our Secure AI Framework (SAIF) [1] includes a fundamental pillar addressing the need to "automate defenses to keep pace with new and existing threats."

This paper shares lessons from our experience leveraging AI to scale our ability to fix bugs, specifically those found by sanitizers in C/C++, Java, and Go code. By automating a pipeline to prompt Large Language Models (LLMs) to generate code fixes for human review, we have harnessed our Gemini [2] model to successfully fix 15% of sanitizer bugs discovered during unit tests, resulting in hundreds of bugs patched. Given the large number of sanitizer bugs found each year, this seemingly modest success rate will with time save significant engineering effort. We expect this success rate to continually improve and anticipate that LLMs can be used to fix bugs in various languages across the software development lifecycle.

LLMs vs. sanitizer bugs

LLMs are well known for their ability to produce language based on patterns and training. Since code is a type of language, LLMs have also proven adept at tackling coding problems. In this vein, we aimed the generative abilities of LLMs at memory safety bugs that were found by sanitizers, a class of tools first introduced by Google in 2012 [3] and now widely used across the industry to test code as it runs.

While Google has promoted the shift to memory-safe languages, like Rust, that are more secure by design, many undetected vulnerabilities persist in legacy code and continue to be uncovered by sanitizers. These tools catch elusive bugs that traditional pre-commit testing misses, thereby revealing issues in production code that could lead to crashes, data corruption, and even exploitable vulnerabilities that allow an attacker to execute arbitrary code.

Since the bugs are uncovered after code is merged, sanitizer testing creates a queue of bugs that are not blocking immediate forward progress, which means their median time-to-fix is longer than bugs detected before code is merged. Any large software company will have an ongoing queue of these bugs to address, and continued improvements to AI-powered bug-finding [4] will only exacerbate this issue, making AI-powered bug-fixing essential to keep pace.

To harness LLMs to generate the code needed to fix these bugs, we built an automated bug-finding-to-fixing pipeline.

An LLM-powered pipeline

An end-to-end solution needs a pipeline to:

1. Find vulnerabilities
2. Isolate and reproduce them
3. Use LLMs to create fixes
4. Test the fixes
5. Surface the best fix for human review and submission

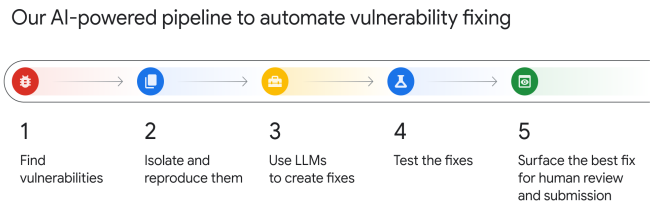

Figure 1: AI-powered patching pipeline

Large companies generate enough sanitizer bugs to justify the effort to automate this process, but each step could also be carried out manually in smaller contexts. The following

discussion addresses our approach and points out alternative considerations for other situations.

Step 1: Finding sanitizer bugs

The process of detecting and reproducing sanitizer bugs is environment-specific, but for the purposes of this LLM pipeline, there are two main considerations:

- Preserve all information from the test run, particularly the stack trace, as it might be useful for prompting the LLM to determine the fix.
- Ensure the service can run easily reproducible tests—both to catch non-deterministic bugs that need multiple attempts to surface, and so that tests can be rerun to be sure an error hasn't already been fixed if there's a delay between detection and a fix attempt.

At Google, our detection service is language agnostic and runs during off-peak hours to find sanitizer bugs in all C/C++, Java, and Go projects.

Step 2: Reproducing and isolating bugs

LLMs do not have infinite "memory," known as context length, so the prompt for the LLM is limited in length. That means we needed to isolate the specific code that needs fixing to create a concise prompt for the LLM. Since most files fit within our LLM's context length limits, we chose to isolate relevant code at the file level, choosing the most likely file to need modification and including its entire contents in the prompt. If developers do not have access to a model with sufficient context length, prompts could instead use smaller pieces of code such as single functions or class definitions.

The code that triggers the sanitizer is not necessarily the same code that needs to be modified to correct the error, so to determine the file to isolate, we initially tried a simple heuristic of selecting the first file on the stack trace that is in the same directory as the test file. We tested whether we could ask the LLM itself to tell us which file from the stack trace should be fixed based on the sanitizer error, but that did not work as well as we had hoped (though with proper training, an LLM could, in principle, do this successfully).

In the end, we chose to train a small, custom ML model to select which file to fix, using thousands of similar bugs from the past as training data. That model returns a score for each file indicating the likelihood that the file contains the code that needs to be modified to fix the error. This score determines the fix strategy: which files to prompt the LLM to try to fix, and in which order and permutations.

Step 3: Generating fixes with LLMs

From our experience, it was not worth spending lots of time on prompt engineering for individual bugs in this context—the results weren't significantly different given the effort and resources needed. We used several slightly different prompts depending on the LLM queried, but all of them followed this structure:

```
You are a Senior Software Engineer tasked
with fixing sanitizer errors. Please fix them.
... code
// Please fix the <error_type> error originating
here.
... LOC pointed to by the stack trace
... code
```

Figure 2: Example prompt used in the experiment

In particular, we found this prompt to return better results than the approach of: "Here is the stack trace, here is the code, please fix." At the moment, it seems difficult for LLMs to connect the dots between the code and stack trace, since the latter has function names that could be abstract; the models performed better when shown exactly where something went wrong.

But which model?

We began our experiment using a base LLM trained on text and code, and a smaller, coding-specific T5X [5]-based model trained on different tasks, which achieved around 5% successful fixes as a baseline. After exploring several options, we found the best results were generated by a Gemini-based model trained on a number of coding tasks, including a training dataset with several thousand examples of previous sanitizer fixes. This model, which is similar in capabilities as the publicly available Gemini Pro [6], enabled us to automatically fix 15% of vulnerabilities in the experiment, and we look forward to pushing this number even higher.

Other large corporations likely also have internal code-specific models, while smaller companies can use the T5X framework, Parameter Efficient Prompt Tuning [7] of existing LLMs, or publicly. available LLMs such as Gemini (available via Bard [8], Google AI Studio [9], and Google Cloud Vertex AI [10]). For those that do not have resources or training data to fine-tune an LLM to their codebase, an option is to use few-shot prompting [11] to show the model how a. few similar bugs were fixed in their code in the past. With the speed at which

the field is progressing, a primary consideration is building a pipeline that easily incorporates new models—for example, by providing the model's name as a command line parameter.

Step 4: Testing the LLM-generated fixes

To test the LLM's solutions, we needed an automated way to create commits from their generated output in order to run automated tests on the modified code. LLMs, especially those not trained for coding, often add extraneous details to the code they produce. While sometimes helpful, such as for commit messages, these details can complicate the process of generating and testing a patch. To address this issue, you can use few-shot prompting to give the LLM examples of your desired output structure, and/or request that generated code is enclosed by special symbols to help filter for the correct lines.

In addition, since LLMs may not output the entire file or function after modifying it, we needed to locate the insertion point for the code modifications. One way to do this is to prompt the LLM to include several lines of code before and after the modification, so simple text analysis can match the right location based on the unchanged lines.

With these two methods, the LLM's generated solution results in an automated commit that's ready for testing and sanitizer checks to catch coding errors or potential AI hallucinations. Based on these tests, we found that different models performed better on different types of errors, so we constructed the pipeline to prompt several models sequentially, giving each model a few attempts before moving on to the next model if no solution was found.

Step 5: Surfacing the best fixes for human review and approvals

These tests and sanitizer checks are only a first step to address the possibility of hallucinations. At the current state of technology, an ML-generated fix—even if it passes all of the tests—must be reviewed by humans. For additional safety, we employed a double human filter on top of the automated analysis: in the first round, we rejected approximately 10-20% of the generated commits as either false positives that did not actually fix the problem or bad solutions that reduced the code quality. We then sent the remaining generated commits to code owners for final validation.

Approximately 95% of the commits sent to code owners were accepted without discussion. This was a higher acceptance rate than human-generated code changes, which often provoke questions and comments. This could be in part because we had thoroughly filtered out bad suggestions by the time they reached the final review stage. But we also want to highlight the possibility that reviewers may have had greater trust in the solutions because they were generated by technology. To address this possibility, developers should be made aware of the potential errors LLMs can produce and be instructed to evaluate the suggestions rigorously.

For example, humans sometimes add temporary code changes with "TODOs" to return to after addressing higher priority tasks. If these examples are not filtered from the training data, the LLM will learn those patterns and suggest similar provisional fixes. We also saw generated suggestions where the code was "fixed" by removing a test that was failing. In another example, the suggested solution made the code run sequentially and added a comment specifying, "cannot run in parallel because it causes a data race." As with anything else, we can improve the quality of LLM-generated fixes by improving the quality of the sample solutions used for training.

Results

At the time of writing, we've accepted several hundred of these LLM-generated commits into Google's codebase, with another several hundred in the process of being validated and submitted. Instead of a software engineer spending an average of two hours to create each of these commits, the necessary patches are now automatically created in seconds.

Perhaps unsurprisingly, we've seen the best success rate in fixing errors stemming from the use of an uninitialized value, a relatively simple fix. But the LLM-generated fixes didn't target only simple errors. They also, for example, effectively initialized matrices and images using the appropriate library methods. In order of the highest fix success rate, the most commonly fixed sanitizer errors fell into four types:

1. Using uninitialized values
2. Data races
3. Buffer overflows
4. Temporal memory errors (e.g. use-after-scope)

Though a 15% success rate might sound low, many thousands of new bugs are found each year, and automating the fixes for even a small fraction of them saves months of engineering effort—meaning that potential security vulnerabilities are closed even faster. We expect improvements to continue pushing that number higher.

Looking ahead

While these initial results are promising, this is just a first step toward a future of AI-powered, automated bug patching. We're currently working on expanding capabilities to include multi-file fixes and to integrate multiple bug sources into the pipeline.

The potential extends beyond just sanitizer fixes. This technology can be applied to any bug caught after code submission, whether unearthed by fuzzing, triggered by a dependency change, or turned up by any process that produces an analyzable error, such as a stack trace. This means bugs across the software lifecycle will become fair game for not only automated detection, but also automated patching using LLMs to generate the fixes. As described in the SAIF implementation guide [12], using AI to automate time-consuming tasks, reduce toil, and speed up defensive mechanisms is an important part of building a safer AI future.

If you're interested in trying a similar pipeline in your organization, you can start by manually prompting an available LLM to fix a sanitizer error and using the model's suggestions as a starting point for the fix. If you have enough bugs to warrant building an automated pipeline, each step described above can be gradually converted from manual entry to an automated step, including fine-tuning an LLM to your specific codebase. We're eager to see what the future holds for unlocking the potential for AI to create end-to-end, automated bug detection and patching solutions.

Acknowledgements

Thank you to Ilya Cherny and Antoine Baudoux who worked with the authors to implement this pipeline.

References

[1] https://safety.google/cybersecurity-advancements/saif.

[2] https://deepmind.google/technologies/gemini.

[3] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf, 2012.

[4] https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html.

[5] https://github.com/google-research/t5x.

[6] https://deepmind.google/technologies/gemini/#capabilities.

[7] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. https://arxiv.org/abs/2104.08691, 2021.

[8] http://bard.google.com.

[9] https://makersuite.google.com.

[10] https://cloud.google.com/vertex-ai.

[11] Tom B. Brown et al. Language models are few-shot learners. https://arxiv.org/abs/2005.14165, 2020.

[12] https://services.google.com/fh/files/blogs/google_secure_ai_framework_approach.pdf.

Jan Nowakowski is a Machine Learning Software Engineer and **Jan Keller** is a Technical Program Manager. Both work in the field of machine learning for security.