Large Language Models for Software Security in the AI Cyber Challenge

Daniel Lu

Advisor: Dr. Taesoo Kim

Collaborators: Andrew Chin, Leyan Pan, SSLab

Georgia Institute of Technology

Contents

Introduction

Software security is an ever-increasing concern as our world becomes more connected. Our society has built a huge reliance on digital systems: our education, healthcare, entertainment, and financial systems are all built on software services. Many critical infrastructure systems, like power grids, are often controlled by software and connected to the internet. These developments have expanded the surface reachable by malicious actors, and in the last years we have seen attacks on oil pipelines (Easterly, 2023), hospitals (Ivanova, 2023), journalists (Marczak et al., 2023), and much more.

The volume of software being written is increasing rapidly, and the question of how to ensure its security *at scale* is becoming more and more important. The end-to-end process of reviewing software for security vulnerabilities and fixing them can be very time-consuming, and often requires a lot of expertise. An interesting question is whether we can create software capable of this process, namely reasoning about security vulnerabilities and patching them automatically. This was the goal of the Cyber Grand Challenge (CGC), a contest held by DARPA in 2016. DARPA (2016) explains that the CGC invited competitors to create systems to do just this, and hoped to establish a "lasting R&D community for automated cyber defense."

The majority of competitors used traditional static and dynamic analysis techniques—like symbolic execution and fuzzing—to find vulnerabilities and generate patches. But in the five years since the contest, machine learning has become more popular in the security community and shows promise in security tasks, such as malware detection and similarity analysis (Ucci et al., 2019). Additionally, Godefroid et al. (2017) showed that deep learning can be used to guide fuzzing, which is the automated process of generating inputs to find vulnerabilities in a program. Further, the release of GPT-3 (Brown et al., 2020) and Codex (Chen et al., 2021) in 2020 and 2021, respectively, sparked additional interest in the use of language models for code-related tasks.

In the last three years, there has been a surge of interest—both from academic

research and industry—in the use of language models in general. OpenAI has continued releasing larger and larger language models, such as the GPT-4 (and built on top of it, ChatGPT) (OpenAI et al., 2023). DeepMind has released its own language model, Gemini (Team et al., 2023). And Meta has developed Llama2, which unlike the previous two models, has publically available weights (Touvron et al., 2023).

Following this increased interest in large language models, DARPA announced the AI Cyber Challenge, or AIxCC (DARPA, 2023a). Similar to the CGC, the AIxCC is a contest that asks teams to design AI systems that can secure software on a large scale. But the scope of AIxCC is much larger than that of the CGC. While the CGC gave competitors individual binaries to analyze, DARPA (2023b) explains that AIxCC teams will receive Challenge Projects written in a variety of programming languages. Each of these projects contains vulnerabilities in a pre-defined set of software weaknesses. The Cyber Reasoning System submitted by each team will be judged on its ability to autonomously

1. produce test cases that demonstrate the presence of vulnerabilities,
2. generate proofs that it "understands" the weaknesses, and
3. submit patches that fix the vulnerabilities.

Teams are additionally rewarded for their ability to perform well on a diverse set of software weaknesses and languages.

In collaboration with the Korea Advanced Institute of Science and Technology, Pohang University of Science and Technology, and Samsung Research America, Georgia Tech is participating in this contest.

Literature review

Previous contest results

The Cyber Grand Challenge tested competitors in an attack/defense scenario similar to a capture-the-flag contest. Each team had the goal of patching vulnerabilities in a set of

Challenge Binaries, while proving vulnerabilities in those of other teams. In the final event, the top three teams were Mayhem, Xandra, and Mechaphish.

Avgerinos et al. (2018) describes the approach taken by Mayhem, which is the system that won the contest. To patch binaries, Mayhem identifies locations of possible crashes and chooses other locations with heuristics. Then, it inserts checks that memory is not accessed at unusual addresses. These patches are performed either by replacing entire functions—which requires a tricky adjustment of addresses and offsets—or by replacing branches with jumps into a custom section containing necessary checks. To find bugs in the services, Mayhem uses a combination of symbolic execution and fuzzing. Notably, Mayhem guides fuzzing using the patching infrastructure, which allowed it to find vulnerabilities more effectively than other teams. Finally, Mayhem includes exploit generation components which used crashing inputs to produce the necessary proof-of-vulnerability, either dynamically through mutations or statically via constraint solving.

Nguyen-Tuong et al. (2018) explains Xandra's approach. For defending, Xandra uses Zipr, a static binary rewriting tool (Hiser et al., 2023). With this tool, Xandra inserts checks in dangerous functions to mitigate modifications to control flow. Additionally, it performs bound inference for arrays on the stack to add safety checks. For attacking, Xandra also uses a combination of symbolic execution and fuzzing. Notably, its symbolic execution engine Grace2 is used to guide the fuzzer by mutating generated inputs.

Shoshitaishvili et al. (2018) outlines the direction taken by Mechaphish or Mechanical Phish. For patching, Mechanical Phish used general binary hardening techniques like return pointer encryption. For attacking, it—just like the other two teams—used a combination of symbolic execution and fuzzing.

These three teams also mention strategy as a key component of their systems. For example, Avgerinos et al. (2018) describes mitigations taken by Mayhem against binaries submitted by adversarial teams. However, the AIxCC contest differs substantially from the CGC in this regard, as there is limited interaction between teams.

Language models for software security

There has been recent interest in the use of language models for writing code. In 2021, OpenAI introduced Codex, a language model based on GPT-3 but fine-tuned on GitHub code (Chen et al., 2021). This introduces promising applications in automated program repair. Previous work in this field often utilizes fix patterns, which are examples of potentially problematic snippets that can be matched against buggy code to generate patches (Liu et al., 2019). However, this approach is not very extensible and fairly naive.

Jin et al. (2023) introduces InferFix, a program repair and static analysis tool for fixing bugs in Java and C# programs. InferFix relies on a database of historic bugs and patches collected from GitHub. After buggy code is localized using the Infer (Facebook, 2023) tool, InferFix finds semantically similar patches in the database and uses a large language model to generate a patch. Then, the patch is validated by running the Infer static analyzer and unit tests. Similarly, Xia et al. (2023b) examines the use of large language models in three patch generation settings: producing entire functions, code infilling, and single line generation. It tests the program repair using a number of language models on benchmarks in a variety of languages, including Defects4J (Java), QuixBugs-Py (Python), and ManyBugs (C). The paper finds that directly applying LLMs to the automated program repair setting outperforms existing baselines without any changes or fine-tuning.

A number of recent publications have demonstrated that large language models are effective in improving program analysis. One area of interest is in decompiler research. Xu et al. (2023) introduces LmPa, a framework for inferring variable names from decompiled code. It does so by prompting ChatGPT to repeatedly generate variable names and propagating them through the code. These names were evaluated against a set of ground truth names by both conducting a user study for developer preferences and utilizing a score function for evaluating similarity. This approach was found to significantly outperform previous deep learning-based approaches. Another paper explores the use of language

models for taint analysis. Liu et al. (2023) proposes LATTE, a fully automated framework that helps large language models reason about security vulnerabilities through a taint analysis approach. In particular, after a decompilation process, LATTE uses a language model to analyze external functions for security-sensitive calls. After producing call chains that lead to those sinks, LATTE uses the LLM to identify sources of input and checks whether they exist in any chains. If so, it constructs prompts for those flows and uses them with the LLM to produce a vulnerability analysis. This work addresses both limitiations of taint analysis (such as manual work required for transfer rules) and of language models (such as small context windows).

Finally, there has been work in the use of language models to guide fuzzing. Although traditional fuzzers have been proven effective for finding bugs, Xia et al. (2023a) argues that they face a number of challenges. Namely, that they suffer from

1. being tightly coupled with a system and a language, since developing a fuzzer requires time-consuming architecture-specific work;
2. having limited support for evolving systems with newer features; and
3. experiencing restricted ability to cover major parts of the input space.

So, the paper proposes Fuzz4All, which uses a large language model for both input generation and mutation. Given documentation of the system to test, Fuzz4All iteratively generates prompts to produce program inputs. Xia et al. (2023a) found that compared to existing fuzzers, Fuzz4All was able to achieve greater coverage on specific systems in the same amount of time while generalizing to all the languages tested: C, C++, SMT, Go, Java, and Python. Additionally, in the 24-hour campaign, the paper found that the coverage achieved by Fuzz4All kept increasing even when fuzzers like Csmith, Fuzz4All, and Hephaestus had already plateaued.

Overall, this research demonstrates that large language models can be effective in a variety of software security tasks, and provides promising directions to explore for the AIxCC contest.

Methods

Benchmark Collection

Once the system is fully developed, we expect to evaluate it on a number of test projects. To this end, we have collected a number of open-source projects with annotated vulnerabilities and test cases. These test cases include both functional correctness tests and tests that demonstrate the presence of security vulnerabilities. These allow, among other things, for patches to be evaluated for correctness and effectiveness.

One specific focus of the benchmark collection is client-side vulnerabilities. These are currently not out-of-scope for the AIxCC contest, but evaluating these bug classes—like XSS and CSRF—might require more infrastructure than checking for server-side application or memory-safety vulnerabilities. This is because evaluating client-side vulnerabilities often requires a browser for rendering HTML and executing JavaScript. To address this, we wrote a testing framework to more easily write test cases for client-side bugs.

Specifically, the framework provides the test cases a Python API that allows them to

- interact with the web server through HTTP requests;
- start new web servers to simulate attackers or third-party services; and
- render web pages as a victim browser would.

The client process and server processes are run in parallel, and communicate over the local network. To avoid interactions between test cases and other services, these are isolated with network namespaces.

Peripheral Components

We would like to apply large language models to create fuzzing harnesses and aid in input generation. A crucial step in this process—and most applications of language models—is prompt engineering. This involves designing and structuring queries to the

language model to most effectively frame the task and provide necessary context. Among other steps, this requires testing a variety of prompts and evaluating their results on a variety of models. So, to improve the efficiency of this process, we developed a playground for prompt engineering. Largely based on OpenPlayground (Friedman, 2024), a tool that allows the side-by-side comparison of a number of different models, our application provides a user interface allowing users to interact with the contest's permitted models and compare their results.

A significant problem to address is that large language models often require contextual information and data to generate meaningful results, but it may not be possible to provide this context upfront. As an example in a different domain, consider a model that makes clothing recommendations. In the process of recommending an outfit, the model might need to know the user's preferences, their occupation, and the weather. But how might the data about the weather be provided? One solution to this problem is to use *plugins*, which are small programs that the model can interact with through special tokens in the output. These tokens instruct the system to include extra data in the context. In the previous example, a plugin might respond to queries about the weather in specific cities.

In our system, a model that seeks to understand code may benefit from additional information about the target program. This might include the language it is written in, the content of files, and what dependencies it has. Although some of this data can easily be conveyed through an initial prompt, it may be infeasible to fit everything that *might be needed* into the query.

Plugins would be able to provide this data on demand. To enable their integration—and crucially, to simplify the process of writing future ones—we developed a plugin system that integrates with the playground. This plugin system provides a Python library for writing plugins as functions, and converts those functions to APIs accessible by the language model and user interface components.

Language Model Plugins

A particular plugin that we developed integrates tools typically meant for source code editing. A vast number of programming languages have servers that provide features like code completion and diagnostics. Many of these servers implement the Language Server Protocol (LSP), which specifies a standard interface for integrating these features into code editors. Some implementations include rust-analyzer for Rust, clangd for C and C++, Pylance for Python, and tsserver for TypeScript. This protocol exposes features that are particularly useful for our system. For example, given an identifier like a function call or variable name, these LSP servers can provide type information and the locations of definitions and references. In the context of generating fuzzing harnesses, this data might help the language model find where a function is defined or understand what arguments it expects.

Because this protocol provides a uniform interface for a variety of languages, it allowed us to write a single plugin that could be used across different settings. But a particular challenge in letting this plugin be effectively used was providing the language model tools for interacting with the codebase and localizing data. In particular, it often only makes sense to query the language server when there is something unknown in a *specific* piece of code or piece of functionality. Further, many interactions with the language server are parameterized by the current file and cursor position—for instance, the line and column number of a specific identifier.

These challenges were addressed by additional plugins. First, we included functionality for navigating the codebase, such as listing directory structure and searching for file content. We also included plugins for reading specific sections of code, such as the context surrounding a specific line in a specific file. This is not quite enough, because these plugins do not provide enough data for the language model to properly determine the line number and column number of code elements of interest: in practice, the language model attempts to guess these values and is often incorrect.

To provide assistance in creating queries, we developed plugins to provide an interface for specifically locating code constructs. Using Tree-sitter, an incremental parsing library, we are able to parse the codebase and look for function definitions, function calls, and variable declarations by name. So, when the language model is preparing a query for an LSP plugin, it can first query these plugins to determine the specific line and column numbers of the constructs it is interested in. This approach takes advantage of the fact that Tree-sitter includes parsers for a wide variety of languages. Just like with the LSP plugin, this can be easily extended to a variety of languages.

Candidate Harness Generation

A potential direction for bug finding, as exhibited by previous contest participants, is utilizing fuzzing. In general, many fuzzing frameworks require code that provides an interface to interact with the tested program—a fuzzing harness. An interesting question is how we can create these harnesses autonomously, without expert knowledge of the program. One approach is to examine the codebase's test suite, and use particular test cases as a starting point for understanding how the program behaves and how to interact with it. In particular, we can utilize a large language model and modify an existing test case to feed in the inputs generated by a fuzzing framework of choice, such as LibFuzzer. Given a test case, this process is done in a few steps.

First, the large language model is queried to separate the test case into components. An observation we make is that test cases often follow similar formats: they begin with some code that sets up the environment and program state, then some code that interacts with the program to test the desired functionality (and checks that outputs are correct), and finally some code that cleans up the final program state. We use the language model to identify where the functional testing code begins and ends to separate it from the surrounding context.

Next, we perform another query to the language model to generate a candidate

harness. Specifically, we ask it to generate code to replace the original functional testing code in the test case to utilize the fuzzing framework's inputs. For example, if a test case calls a function with a particular argument, we might want to replace that argument with data that comes from the fuzzer. We also allow the language model to leave special annotations to denote where the setup and cleanup code should be placed.

Finally, this generated code is patched into the original test case. Both the original test case and output code are parsed with Tree-sitter. Then, we extract the new testing code and any helper functions generated by the language model and properly construct the new test case, including the setup and cleanup code. This process also allows us to format the code properly and ensure that it contains the required data and names for the fuzzing framework to run.

Code Generation Loop

Though the previous components might generate candidate fuzzing harnesses that contain correct *ideas* for how fuzzing inputs can be fed into the program, the generated code tends to be unlikely to compile. Our goal is to modify these candidates to properly compile and run.

While there exist a number of existing approaches to code repair, we chose to continue leveraging large language models. Our approach was fairly simple: we attempt to compile a candidate harness, and if it fails, we create a prompt containing the previous attempt and compile errors. We then query the language model to generate a new candidate harness that fixes the errors. This process is repeated until the code properly compiles.

In order to improve the efficiency of this process and improve the quality of the generated code, we also built additional stages into this code generation loop. This is because we noticed compilation errors were sometimes fixed in ways that were not ideal. For instance, the original harness candidate might include a reference to a helper function that was not defined. Sometimes, the compile loop would result in a harness that defined

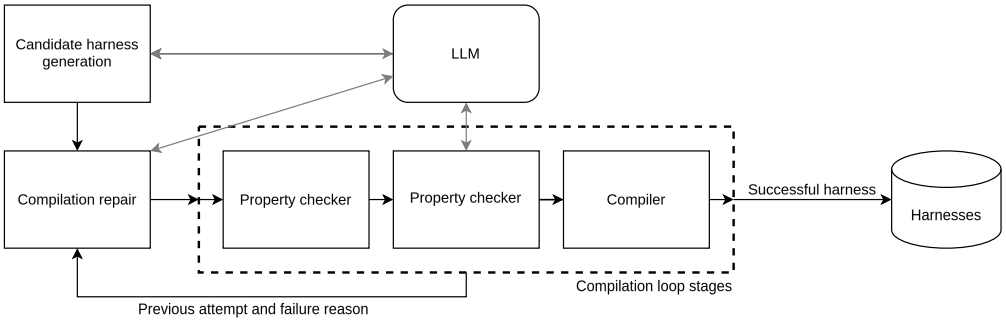

Figure 1. Code generation loop example with two intermediate stages

the helper function, but left it as an empty stub. While this did mean the compilation loop converged, the final output was less effective as a fuzzing harness.

So, we added stages to the code generation loop that would check for properties of the generated code. Some of these operate on the syntax level—we check that there don't exist empty functions, or that the fuzzing inputs are actually used. Others make additional queries to the language model to ensure that the generated code semantically makes sense. As with the final compilation step, if any of these stages fail, we create a new prompt that includes a description of the problem and the previous attempt. Then, we query the language model to generate a new candidate harness.

Parallelization

One observation we made was that performing the initial candidate harness generation is fairly quick, but may require many attempts to produce a meaningful result. This process is very amenable to parallelization, as each attempt is completely independent. In order to prioritize the most promising candidates, we expanded on the quality checks in the code generation loop to assign scores to candidates. Along with the "negative" checks described earlier that ensure the code is free from immediate issues, we added "positive" checks that prioritize candidates with certain properties. For example, harnesses that are more similar to original test cases are less likely to contain completely hallucinated, nonsensical code. We introduced additional negative tests as well, especially

for issues that can only be fixed in the original generation stage—recall that we only have the language model generate the functional testing code, and leave comments for the setup and cleanup code. If the generated harness does not correctly include these annotations (perhaps by giving extraneous ones), the harness is unlikely to work at all.

Data Collection and Analysis

We analyzed these components qualitatively, as they are all components of a large system. For the harness generation and code generation loop in particular, we were especially interested in how they might perform in an early example project provided by DARPA, namely a patched version of the Linux kernel. In order to get a sense for how well the system might perform, we collected a number of Linux kernel CVEs with associated proof-of-concept exploits. Then, by treating these proof-of-concept exploits *as test cases*, we can use them as inputs to the fuzzing harness generation system. The output can be evaluated by a human to determine whether the modifications are meaningful, and whether the same vulnerabilities can be reproduced with fuzzing inputs.

Once the entirety of the system is functional, it will be tested and evaluated in two main ways. Most immediately, we compiled a large benchmark dataset consisting of open-source projects and annotated vulnerabilities. These include test cases for program correctness, as well as test cases that demonstrate the existence of security vulnerabilities. This dataset will be used to evaluate the system in two ways: first, to check the efficacy of this component for finding vulnerabilities; and second, to evaluate whether the component is effective in the end-to-end process of producing patches. These can be determined by the percentage of vulnerabilities found and the percentage of patches that are correct, respectively.

Additionally, the AIxCC contest has provided a number of Challenge Projects to set expectations for the contest. After testing against the benchmark dataset, we will test the system against these Challenge Projects to evaluate its performance. Further, during the

preliminary rounds of the contest near the end of the semester, we will test the system against the projects featured in those competitions.

Results

Language Model Plugins

We qualitatively examined the effectiveness of our plugin system on some more general tasks, then some tasks that relate more directly to code analysis.

Knowledge base. Using the plugin API we developed, we developed plugins that integrate with Wikipedia and manpages in order to test that the language model can effectively interact plugins through the system, in general. When presented with the query "On what date did the Klaus Advanced Computing Building open?", the language model

1. queried the *get Wikipedia pages* plugin to find the Wikipedia page for the building;
2. queried the *get Wikipedia summary* plugin;
3. checked the *get Wikipedia sections* plugin to list the sections of the page;
4. and queried the *get Wikipedia section content* plugin to find the section containing the building's opening date.

Finally, it correctly generated the response "The Klaus Advanced Computing Building opened on October 26, 2006." The system prompt did contain information about how to interact with the plugins, and that the language model should use the plugins if necessary. However, it did not contain instructions for specific processes, like narrowing down the Wikipedia page to the correct section. This suggests that the system is capable of some limited reasoning and planning when interacting with plugins. The system exhibited similar behavior when queried about more technical topics, such as "How does the V8 JavaScript engine handle garbage collection?" and "What are example programming languages that compile to the JVM?".

As another example, when asked what command line flag instructs the `grep` command to ignore case, the language model utilized the *get manpage* plugin to locate the `grep` manage, then correctly identified the `-i` flag as the one that ignores case.

Code navigation. One setting where the system struggled was in navigating codebases without direct instructions. A number of language server protocol plugins were made available, including ones for listing the directory structure; searching for file content; retrieving specific sections of code; locating symbols by name; and finding references and definitions of symbols. A plugin for searching through file content was intentionally omitted. When given the query "Where is the function called 'test' defined?", the behavior of the language model was extremely brittle. In one attempt, it hallucinated a plugin that would search for the function definition, and when the query was unsuccessful, it responded that the function did not exist. In another attempt, the language model indicated that it did not have the proper tools to answer the question.

Similarly, when the language model *was* equipped with the ability to search for strings in files, it performed that query and found matches in a number of files. However, instead of verifying the content of the files to check which references were definitions, it simply responded that the function was defined in each file found.

These cases suggested that the system has a limited ability to create a plan for interacting with plugins, especially when a possible solution path (such as checking files with semantically related names) is not immediately obvious.

Code navigation with instructions. An additional reason for this belief is that the language model performed considerably better when the query was structured in way that hinted at a solution. For example, when asked "The function 'test' is referenced in the file 'main.py.' What does it return?", the language model

1. queried the *get symbol location* plugin to find where the function was used in 'main.py';

2. used the *get file content* plugin to retreive the line where the function was called;

3. queried the *get symbol definition* plugin to find where the function was defined; and
4. used the *get file content* plugin to retrieve the function definition.

Then, it correctly responded that the function returned one regardless of its input. This behavior was consistent across similar queries, and other languages supported. The difference in behavior between this case and the more open-ended ones described earlier suggests that tasks that involve implicit reasoning and planning are difficult for the system.

Harness Generation

CVE	Feasible candidate harness	Compilation convergence
CVE-2020-27786	3 / 5	2 / 3
CVE-2021-41541	3 / 5	3 / 3
CVE-2022-0185	5 / 5	5 / 5
CVE-2022-0995	5 / 5	5 / 5
CVE-2022-1015	2 / 5	2 / 2
CVE-2022-2585	1 / 5	1 / 1
CVE-2022-2606	0 / 5	-
CVE-2022-32250	0 / 5	-
CVE-2023-0719	1 / 5	1 / 1
CVE-2023-2008	2 / 5	2 / 2
CVE-2023-2513	0 / 5	-
CVE-2023-2598	0 / 5	-
CVE-2023-3390	0 / 5	-
CVE-2023-6560	1 / 5	1 / 1

Table 1

Harness generation results

To evaluate the harness generation, we examined fourteen Linux kernel CVEs and

interpreted the proof-of-concept exploits as test cases. Specifically, we compiled a collection of test cases ahead of time, and excluded one (CVE-2021-43267) that was used as an example in the prompt. Then we used the harness generation system to generate fuzzing harnesses based on those test cases, and recorded the results. Specifically, we performed five candidate generation attempts on each CVE. In each attempt, we allowed the compilation loop ten iterations to converge. A brief summary is provided in Table 1. We only consider the convergence of compilation among the feasible candidates, as although infeasible candidates may lead to compiling harnesses, those harnesses were never correct.

In nine of the fourteen cases, our system was able to produce harnesses that (a) compiled, (b) reasonably utilized the fuzzing inputs, and (c) would be able to reproduce the vulnerabilities in the test cases. This means that if the output harnesses were to be used, the bug could be reproduced by a fuzzer, given the right inputs. In the remaining cases, the system failed to produce feasible harness candidates at all. Notice that the convergence of the compilation loop was never an issue—that is, if the system could produce a feasible candidate in the five attempts, it could always modify the candidate to properly compile.

Discussion

Language Model Plugins

An immediate question is how we might improve the system's ability to plan its interactions with plugins, to be more robust in the second case. To restate the problem, the system was able to perform well when the query implied a natural solution path (such as locating a Wikipedia section before checking its content) but struggled when the solution was less immediate. One promising solution was to simply ask the language model to first generate a plan for how the plugins might be used, before actually querying them. We noticed that this did improve performance, which suggests that the planning step does not need to be performed by the user. An interesting direction for future work would be to explore more principled ways of generating these plans.

Harness Generation

In the harness generation results, we see that in five of the fourteen cases, the system was not able to produce a feasible harness candidate within five attempts. While increasing the number of attempts did improve the results in some cases, it did not help in the majority of those cases. Examining the specific test cases where this method failed might provide some insight into where the approach breaks down.

One major reason that the system would fail is simply because some test cases are not suitable for simple modification. For instance, CVE-2023-2513 was a use-after-free vulnerability in ext4, caused by the handling of specific attributes in a sequence of interactions. Examining the proof-of-concept script that we treated as a test case, we see that there is no obvious way to sensibly modify the script for fuzzing. In this situation—and in fact most cases where the system failed—the test case was just not suitable for the overall approach.

Another reason for failure is that some of our proof-of-concept scripts were much closer to exploits than test cases. For example, the CVE-2023-2598 script is very involved, and contains among other things a long section for leaking kernel addresses. In the process of generating the initial candidate harness, there did not exist a unique location where the fuzzing inputs could be reasonably inserted.

Overall, on the cases that were particularly suitable for the approach—namely those that were closer to traditional unit tests and had very clear inputs to the tested functionality—the system effectively produced harnesses that would be able to reproduce the vulnerabilities.

References

Avgerinos, T., Brumley, D., Davis, J., Goulden, R., Nighswander, T., Rebert, A., and Williamson, N. (2018). The Mayhem Cyber Reasoning System. *IEEE Security & Privacy*, 16(2):52–60. Conference Name: IEEE Security & Privacy.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language Models are Few-Shot Learners. arXiv:2005.14165 [cs].

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs].

DARPA (2016). Cyber Grand Challenge.

DARPA (2023a). AIxCC.

DARPA (2023b). Artificial Intelligence Cyber Challenge Request for Comments.

Easterly, J. (2023). The Attack on Colonial Pipeline: What We've Learned & What We've Done Over the Past Two Years | CISA.

Facebook (2023). facebook/infer. original-date: 2015-01-26T11:19:13Z.

Friedman, N. (2024). nat/openplayground. original-date: 2023-02-26T22:58:17Z.

Godefroid, P., Peleg, H., and Singh, R. (2017). Learn&Fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59, Urbana, IL. IEEE.

Hiser, J. D., Nguyen-Tuong, A., and Davidson, J. W. (2023). Zipr: A High-Impact, Robust, Open-source, Multi-platform, Static Binary Rewriter. arXiv:2312.00714 [cs].

Ivanova, I. (2023). HCA Healthcare says hackers stole data on 11 million patients - CBS News.

Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N., and Svyatkovskiy, A. (2023). InferFix: End-to-End Program Repair with LLMs. arXiv:2303.07263 [cs].

Liu, K., Koyuncu, A., Kim, D., and Bissyandé, T. F. (2019). TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 31–42, New York, NY, USA. Association for Computing Machinery.

Liu, P., Sun, C., Zheng, Y., Feng, X., Qin, C., Wang, Y., Li, Z., and Sun, L. (2023). Harnessing the Power of LLM to Support Binary Taint Analysis. arXiv:2310.08275 [cs].

LLVM Developer Group (2024). clangd/clangd. original-date: 2019-01-14T22:07:51Z.

Marczak, B., Scott-Railton, J., Razzak, B. A., and Deibert, R. (2023). Triple Threat: NSO Group's Pegasus Spyware Returns in 2022 with a Trio of iOS 15 and iOS 16 Zero-Click Exploit Chains. Technical report, Citizen Lab, University of Toronto. Section: Targeted Threats.

Microsoft (2024a). microsoft/pylance-release. original-date: 2020-05-21T16:50:15Z.

Microsoft (2024b). microsoft/TypeScript. original-date: 2014-06-17T15:28:39Z.

Nguyen-Tuong, A., Melski, D., Davidson, J. W., Co, M., Hawkins, W., Hiser, J. D., Morris, D., Nguyen, D., and Rizzi, E. (2018). Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge. *IEEE Security & Privacy*, 16(2):42–51. Conference Name: IEEE Security & Privacy.

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., Bello, I., Berdine, J., Bernadett-Shapiro, G., Berner, C., Bogdonoff, L., Boiko, O., Boyd, M., Brakman, A.-L., Brockman, G., Brooks, T., Brundage, M., Button, K., Cai, T., Campbell, R., Cann, A., Carey, B., Carlson, C., Carmichael, R., Chan, B., Chang, C., Chantzis, F., Chen, D., Chen, S., Chen, R., Chen, J., Chen, M., Chess, B., Cho, C., Chu, C., Chung, H. W., Cummings, D., Currier, J., Dai, Y., Decareaux, C., Degry, T., Deutsch, N., Deville, D., Dhar, A., Dohan, D., Dowling, S., Dunning, S., Ecoffet, A., Eleti, A., Eloundou, T., Farhi, D., Fedus, L., Felix, N., Fishman, S. P., Forte, J., Fulford, I., Gao, L., Georges, E., Gibson, C., Goel, V., Gogineni, T., Goh, G., Gontijo-Lopes, R., Gordon, J., Grafstein, M., Gray, S., Greene, R., Gross, J., Gu, S. S., Guo, Y., Hallacy, C., Han, J., Harris, J., He, Y., Heaton, M., Heidecke, J., Hesse, C., Hickey, A., Hickey, W., Hoeschele, P., Houghton, B., Hsu, K., Hu, S., Hu, X., Huizinga, J., Jain, S., Jain, S., Jang, J., Jiang, A., Jiang, R., Jin, H., Jin, D., Jomoto, S., Jonn, B., Jun, H., Kaftan, T., Kaiser, L., Kamali, A., Kanitscheider, I., Keskar, N. S., Khan, T., Kilpatrick, L., Kim, J. W., Kim, C., Kim, Y., Kirchner, H., Kiros, J., Knight, M., Kokotajlo, D., Kondraciuk, L., Kondrich, A., Konstantinidis, A., Kosic, K., Krueger, G., Kuo, V., Lampe, M., Lan, I., Lee, T., Leike, J., Leung, J., Levy, D., Li, C. M., Lim, R., Lin, M., Lin, S., Litwin, M., Lopez, T., Lowe, R., Lue, P., Makanju, A., Malfacini, K., Manning, S., Markov, T., Markovski, Y., Martin, B., Mayer, K., Mayne, A., McGrew, B., McKinney, S. M.,

McLeavey, C., McMillan, P., McNeil, J., Medina, D., Mehta, A., Menick, J., Metz, L., Mishchenko, A., Mishkin, P., Monaco, V., Morikawa, E., Mossing, D., Mu, T., Murati, M., Murk, O., Mély, D., Nair, A., Nakano, R., Nayak, R., Neelakantan, A., Ngo, R., Noh, H., Ouyang, L., O'Keefe, C., Pachocki, J., Paino, A., Palermo, J., Pantuliano, A., Parascandolo, G., Parish, J., Parparita, E., Passos, A., Pavlov, M., Peng, A., Perelman, A., Peres, F. d. A. B., Petrov, M., Pinto, H. P. d. O., Michael, Pokorny, Pokrass, M., Pong, V., Powell, T., Power, A., Power, B., Proehl, E., Puri, R., Radford, A., Rae, J., Ramesh, A., Raymond, C., Real, F., Rimbach, K., Ross, C., Rotsted, B., Roussez, H., Ryder, N., Saltarelli, M., Sanders, T., Santurkar, S., Sastry, G., Schmidt, H., Schnurr, D., Schulman, J., Selsam, D., Sheppard, K., Sherbakov, T., Shieh, J., Shoker, S., Shyam, P., Sidor, S., Sigler, E., Simens, M., Sitkin, J., Slama, K., Sohl, I., Sokolowsky, B., Song, Y., Staudacher, N., Such, F. P., Summers, N., Sutskever, I., Tang, J., Tezak, N., Thompson, M., Tillet, P., Tootoonchian, A., Tseng, E., Tuggle, P., Turley, N., Tworek, J., Uribe, J. F. C., Vallone, A., Vijayvergiya, A., Voss, C., Wainwright, C., Wang, J. J., Wang, A., Wang, B., Ward, J., Wei, J., Weinmann, C. J., Welihinda, A., Welinder, P., Weng, J., Weng, L., Wiethoff, M., Willner, D., Winter, C., Wolrich, S., Wong, H., Workman, L., Wu, S., Wu, J., Wu, M., Xiao, K., Xu, T., Yoo, S., Yu, K., Yuan, Q., Zaremba, W., Zellers, R., Zhang, C., Zhang, M., Zhao, S., Zheng, T., Zhuang, J., Zhuk, W., and Zoph, B. (2023). GPT-4 Technical Report. arXiv:2303.08774 [cs].

Rust Foundation (2024). rust-lang/rust-analyzer. original-date: 2017-12-21T19:24:42Z.

Shoshitaishvili, Y., Bianchi, A., Borgolte, K., Cama, A., Corbetta, J., Disperati, F., Dutcher, A., Grosen, J., Grosen, P., Machiry, A., Salls, C., Stephens, N., Wang, R., and Vigna, G. (2018). Mechanical Phish: Resilient Autonomous Hacking. *IEEE Security & Privacy*, 16(2):12–22. Conference Name: IEEE Security & Privacy.

Team, G., Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., Millican, K., Silver, D., Petrov, S., Johnson, M., Antonoglou,

I., Schrittwieser, J., Glaese, A., Chen, J., Pitler, E., Lillicrap, T., Lazaridou, A., Firat, O., Molloy, J., Isard, M., Barham, P. R., Hennigan, T., Lee, B., Viola, F., Reynolds, M., Xu, Y., Doherty, R., Collins, E., Meyer, C., Rutherford, E., Moreira, E., Ayoub, K., Goel, M., Tucker, G., Piqueras, E., Krikun, M., Barr, I., Savinov, N., Danihelka, I., Roelofs, B., White, A., Andreassen, A., von Glehn, T., Yagati, L., Kazemi, M., Gonzalez, L., Khalman, M., Sygnowski, J., Frechette, A., Smith, C., Culp, L., Proleev, L., Luan, Y., Chen, X., Lottes, J., Schucher, N., Lebron, F., Rrustemi, A., Clay, N., Crone, P., Kocisky, T., Zhao, J., Perz, B., Yu, D., Howard, H., Bloniarz, A., Rae, J. W., Lu, H., Sifre, L., Maggioni, M., Alcober, F., Garrette, D., Barnes, M., Thakoor, S., Austin, J., Barth-Maron, G., Wong, W., Joshi, R., Chaabouni, R., Fatiha, D., Ahuja, A., Liu, R., Li, Y., Cogan, S., Chen, J., Jia, C., Gu, C., Zhang, Q., Grimstad, J., Hartman, A. J., Chadwick, M., Tomar, G. S., Garcia, X., Senter, E., Taropa, E., Pillai, T. S., Devlin, J., Laskin, M., Casas, D. d. L., Valter, D., Tao, C., Blanco, L., Badia, A. P., Reitter, D., Chen, M., Brennan, J., Rivera, C., Brin, S., Iqbal, S., Surita, G., Labanowski, J., Rao, A., Winkler, S., Parisotto, E., Gu, Y., Olszewska, K., Zhang, Y., Addanki, R., Miech, A., Louis, A., Shafey, L. E., Teplyashin, D., Brown, G., Catt, E., Attaluri, N., Balaguer, J., Xiang, J., Wang, P., Ashwood, Z., Briukhov, A., Webson, A., Ganapathy, S., Sanghavi, S., Kannan, A., Chang, M.-W., Stjerngren, A., Djolonga, J., Sun, Y., Bapna, A., Aitchison, M., Pejman, P., Michalewski, H., Yu, T., Wang, C., Love, J., Ahn, J., Bloxwich, D., Han, K., Humphreys, P., Sellam, T., Bradbury, J., Godbole, V., Samangooei, S., Damoc, B., Kaskasoli, A., Arnold, S. M. R., Vasudevan, V., Agrawal, S., Riesa, J., Lepikhin, D., Tanburn, R., Srinivasan, S., Lim, H., Hodkinson, S., Shyam, P., Ferret, J., Hand, S., Garg, A., Paine, T. L., Li, J., Li, Y., Giang, M., Neitz, A., Abbas, Z., York, S., Reid, M., Cole, E., Chowdhery, A., Das, D., Rogozińska, D., Nikolaev, V., Sprechmann, P., Nado, Z., Zilka, L., Prost, F., He, L., Monteiro, M., Mishra, G., Welty, C., Newlan, J., Jia, D., Allamanis, M., Hu, C. H., de Liedekerke, R., Gilmer, J., Saroufim, C., Rijhwani, S., Hou, S., Shrivastava, D., Baddepudi, A., Goldin, A., Ozturel,

A., Cassirer, A., Xu, Y., Sohn, D., Sachan, D., Amplayo, R. K., Swanson, C., Petrova, D., Narayan, S., Guez, A., Brahma, S., Landon, J., Patel, M., Zhao, R., Villela, K., Wang, L., Jia, W., Rahtz, M., Giménez, M., Yeung, L., Lin, H., Keeling, J., Georgiev, P., Mincu, D., Wu, B., Haykal, S., Saputro, R., Vodrahalli, K., Qin, J., Cankara, Z., Sharma, A., Fernando, N., Hawkins, W., Neyshabur, B., Kim, S., Hutter, A., Agrawal, P., Castro-Ros, A., Driessche, G. v. d., Wang, T., Yang, F., Chang, S.-y., Komarek, P., McIlroy, R., Lučić, M., Zhang, G., Farhan, W., Sharman, M., Natsev, P., Michel, P., Cheng, Y., Bansal, Y., Qiao, S., Cao, K., Shakeri, S., Butterfield, C., Chung, J., Rubenstein, P. K., Agrawal, S., Mensch, A., Soparkar, K., Lenc, K., Chung, T., Pope, A., Maggiore, L., Kay, J., Jhakra, P., Wang, S., Maynez, J., Phuong, M., Tobin, T., Tacchetti, A., Trebacz, M., Robinson, K., Katariya, Y., Riedel, S., Bailey, P., Xiao, K., Ghelani, N., Aroyo, L., Slone, A., Houlsby, N., Xiong, X., Yang, Z., Gribovskaya, E., Adler, J., Wirth, M., Lee, L., Li, M., Kagohara, T., Pavagadhi, J., Bridgers, S., Bortsova, A., Ghemawat, S., Ahmed, Z., Liu, T., Powell, R., Bolina, V., Iinuma, M., Zablotskaia, P., Besley, J., Chung, D.-W., Dozat, T., Comanescu, R., Si, X., Greer, J., Su, G., Polacek, M., Kaufman, R. L., Tokumine, S., Hu, H., Buchatskaya, E., Miao, Y., Elhawaty, M., Siddhant, A., Tomasev, N., Xing, J., Greer, C., Miller, H., Ashraf, S., Roy, A., Zhang, Z., Ma, A., Filos, A., Besta, M., Blevins, R., Klimenko, T., Yeh, C.-K., Changpinyo, S., Mu, J., Chang, O., Pajarskas, M., Muir, C., Cohen, V., Lan, C. L., Haridasan, K., Marathe, A., Hansen, S., Douglas, S., Samuel, R., Wang, M., Austin, S., Lan, C., Jiang, J., Chiu, J., Lorenzo, J. A., Sjösund, L. L., Cevey, S., Gleicher, Z., Avrahami, T., Boral, A., Srinivasan, H., Selo, V., May, R., Aisopos, K., Hussenot, L., Soares, L. B., Baumli, K., Chang, M. B., Recasens, A., Caine, B., Pritzel, A., Pavetic, F., Pardo, F., Gergely, A., Frye, J., Ramasesh, V., Horgan, D., Badola, K., Kassner, N., Roy, S., Dyer, E., Campos, V., Tomala, A., Tang, Y., Badawy, D. E., White, E., Mustafa, B., Lang, O., Jindal, A., Vikram, S., Gong, Z., Caelles, S., Hemsley, R., Thornton, G., Feng, F., Stokowiec, W., Zheng, C., Thacker, P., Unlu, C., Zhang, Z.,

Saleh, M., Svensson, J., Bileschi, M., Patil, P., Anand, A., Ring, R., Tsihlas, K., Vezer, A., Selvi, M., Shevlane, T., Rodriguez, M., Kwiatkowski, T., Daruki, S., Rong, K., Dafoe, A., FitzGerald, N., Gu-Lemberg, K., Khan, M., Hendricks, L. A., Pellat, M., Feinberg, V., Cobon-Kerr, J., Sainath, T., Rauh, M., Hashemi, S. H., Ives, R., Hasson, Y., Li, Y., Noland, E., Cao, Y., Byrd, N., Hou, L., Wang, Q., Sottiaux, T., Paganini, M., Lespiau, J.-B., Moufarek, A., Hassan, S., Shivakumar, K., van Amersfoort, J., Mandhane, A., Joshi, P., Goyal, A., Tung, M., Brock, A., Sheahan, H., Misra, V., Li, C., Rakićević, N., Dehghani, M., Liu, F., Mittal, S., Oh, J., Noury, S., Sezener, E., Huot, F., Lamm, M., De Cao, N., Chen, C., Elsayed, G., Chi, E., Mahdieh, M., Tenney, I., Hua, N., Petrychenko, I., Kane, P., Scandinaro, D., Jain, R., Uesato, J., Datta, R., Sadovsky, A., Bunyan, O., Rabiej, D., Wu, S., Zhang, J., Vasudevan, G., Leurent, E., Alnahlawi, M., Georgescu, I., Wei, N., Zheng, I., Chan, B., Rabinovitch, P. G., Stanczyk, P., Zhang, Y., Steiner, D., Naskar, S., Azzam, M., Johnson, M., Paszke, A., Chiu, C.-C., Elias, J. S., Mohiuddin, A., Muhammad, F., Miao, J., Lee, A., Vieillard, N., Potluri, S., Park, J., Davoodi, E., Zhang, J., Stanway, J., Garmon, D., Karmarkar, A., Dong, Z., Lee, J., Kumar, A., Zhou, L., Evens, J., Isaac, W., Chen, Z., Jia, J., Levskaya, A., Zhu, Z., Gorgolewski, C., Grabowski, P., Mao, Y., Magni, A., Yao, K., Snaider, J., Casagrande, N., Suganthan, P., Palmer, E., Irving, G., Loper, E., Faruqui, M., Arkatkar, I., Chen, N., Shafran, I., Fink, M., Castaño, A., Giannoumis, I., Kim, W., Rybiński, M., Sreevatsa, A., Prendki, J., Soergel, D., Goedeckemeyer, A., Gierke, W., Jafari, M., Gaba, M., Wiesner, J., Wright, D. G., Wei, Y., Vashisht, H., Kulizhskaya, Y., Hoover, J., Le, M., Li, L., Iwuanyanwu, C., Liu, L., Ramirez, K., Khorlin, A., Cui, A., LIN, T., Georgiev, M., Wu, M., Aguilar, R., Pallo, K., Chakladar, A., Repina, A., Wu, X., van der Weide, T., Ponnapalli, P., Kaplan, C., Simsa, J., Li, S., Dousse, O., Yang, F., Piper, J., Ie, N., Lui, M., Pasumarthi, R., Lintz, N., Vijayakumar, A., Thiet, L. N., Andor, D., Valenzuela, P., Paduraru, C., Peng, D., Lee, K., Zhang, S., Greene, S., Nguyen, D. D., Kurylowicz, P., Velury, S., Krause, S., Hardin, C., Dixon, L., Janzer, L., Choo, K., Feng,

Z., Zhang, B., Singhal, A., Latkar, T., Zhang, M., Le, Q., Abellan, E. A., Du, D., McKinnon, D., Antropova, N., Bolukbasi, T., Keller, O., Reid, D., Finchelstein, D., Raad, M. A., Crocker, R., Hawkins, P., Dadashi, R., Gaffney, C., Lall, S., Franko, K., Filonov, E., Bulanova, A., Leblond, R., Yadav, V., Chung, S., Askham, H., Cobo, L. C., Xu, K., Fischer, F., Xu, J., Sorokin, C., Alberti, C., Lin, C.-C., Evans, C., Zhou, H., Dimitriev, A., Forbes, H., Banarse, D., Tung, Z., Liu, J., Omernick, M., Bishop, C., Kumar, C., Sterneck, R., Foley, R., Jain, R., Mishra, S., Xia, J., Bos, T., Cideron, G., Amid, E., Piccinno, F., Wang, X., Banzal, P., Gurita, P., Noga, H., Shah, P., Mankowitz, D. J., Polozov, A., Kushman, N., Krakovna, V., Brown, S., Bateni, M., Duan, D., Firoiu, V., Thotakuri, M., Natan, T., Mohananey, A., Geist, M., Mudgal, S., Girgin, S., Li, H., Ye, J., Roval, O., Tojo, R., Kwong, M., Lee-Thorp, J., Yew, C., Yuan, Q., Bagri, S., Sinopalnikov, D., Ramos, S., Mellor, J., Sharma, A., Severyn, A., Lai, J., Wu, K., Cheng, H.-T., Miller, D., Sonnerat, N., Vnukov, D., Greig, R., Beattie, J., Caveness, E., Bai, L., Eisenschlos, J., Korchemniy, A., Tsai, T., Jasarevic, M., Kong, W., Dao, P., Zheng, Z., Liu, F., Yang, F., Zhu, R., Geller, M., Teh, T. H., Sanmiya, J., Gladchenko, E., Trdin, N., Sozanschi, A., Toyama, D., Rosen, E., Tavakkol, S., Xue, L., Elkind, C., Woodman, O., Carpenter, J., Papamakarios, G., Kemp, R., Kafle, S., Grunina, T., Sinha, R., Talbert, A., Goyal, A., Wu, D., Owusu-Afriyie, D., Du, C., Thornton, C., Pont-Tuset, J., Narayana, P., Li, J., Fatehi, S., Wieting, J., Ajmeri, O., Uria, B., Zhu, T., Ko, Y., Knight, L., Héliou, A., Niu, N., Gu, S., Pang, C., Tran, D., Li, Y., Levine, N., Stolovich, A., Kalb, N., Santamaria-Fernandez, R., Goenka, S., Yustalim, W., Strudel, R., Elqursh, A., Lakshminarayanan, B., Deck, C., Upadhyay, S., Lee, H., Dusenberry, M., Li, Z., Wang, X., Levin, K., Hoffmann, R., Holtmann-Rice, D., Bachem, O., Yue, S., Arora, S., Malmi, E., Mirylenka, D., Tan, Q., Koh, C., Yeganeh, S. H., Põder, S., Zheng, S., Pongetti, F., Tariq, M., Sun, Y., Ionita, L., Seyedhosseini, M., Tafti, P., Kotikalapudi, R., Liu, Z., Gulati, A., Liu, J., Ye, X., Chrzaszcz, B., Wang, L., Sethi, N., Li, T., Brown, B., Singh, S., Fan, W., Parisi, A., Stanton, J., Kuang, C., Koverkathu, V.,

Choquette-Choo, C. A., Li, Y., Lu, T. J., Ittycheriah, A., Shroff, P., Sun, P., Varadarajan, M., Bahargam, S., Willoughby, R., Gaddy, D., Dasgupta, I., Desjardins, G., Cornero, M., Robenek, B., Mittal, B., Albrecht, B., Shenoy, A., Moiseev, F., Jacobsson, H., Ghaffarkhah, A., Rivière, M., Walton, A., Crepy, C., Parrish, A., Liu, Y., Zhou, Z., Farabet, C., Radebaugh, C., Srinivasan, P., van der Salm, C., Fidjeland, A., Scellato, S., Latorre-Chimoto, E., Klimczak-Plucińska, H., Bridson, D., de Cesare, D., Hudson, T., Mendolicchio, P., Walker, L., Morris, A., Penchev, I., Mauger, M., Guseynov, A., Reid, A., Odoom, S., Loher, L., Cotruta, V., Yenugula, M., Grewe, D., Petrushkina, A., Duerig, T., Sanchez, A., Yadlowsky, S., Shen, A., Globerson, A., Kurzrok, A., Webb, L., Dua, S., Li, D., Lahoti, P., Bhupatiraju, S., Hurt, D., Qureshi, H., Agarwal, A., Shani, T., Eyal, M., Khare, A., Belle, S. R., Wang, L., Tekur, C., Kale, M. S., Wei, J., Sang, R., Saeta, B., Liechty, T., Sun, Y., Zhao, Y., Lee, S., Nayak, P., Fritz, D., Vuyyuru, M. R., Aslanides, J., Vyas, N., Wicke, M., Ma, X., Bilal, T., Eltyshev, E., Balle, D., Martin, N., Cate, H., Manyika, J., Amiri, K., Kim, Y., Xiong, X., Kang, K., Luisier, F., Tripuraneni, N., Madras, D., Guo, M., Waters, A., Wang, O., Ainslie, J., Baldridge, J., Zhang, H., Pruthi, G., Bauer, J., Yang, F., Mansour, R., Gelman, J., Xu, Y., Polovets, G., Liu, J., Cai, H., Chen, W., Sheng, X., Xue, E., Ozair, S., Yu, A., Angermueller, C., Li, X., Wang, W., Wiesinger, J., Koukoumidis, E., Tian, Y., Iyer, A., Gurumurthy, M., Goldenson, M., Shah, P., Blake, M. K., Yu, H., Urbanowicz, A., Palomaki, J., Fernando, C., Brooks, K., Durden, K., Mehta, H., Momchev, N., Rahimtoroghi, E., Georgaki, M., Raul, A., Ruder, S., Redshaw, M., Lee, J., Jalan, K., Li, D., Perng, G., Hechtman, B., Schuh, P., Nasr, M., Chen, M., Milan, K., Mikulik, V., Strohman, T., Franco, J., Green, T., Hassabis, D., Kavukcuoglu, K., Dean, J., and Vinyals, O. (2023). Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs].

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V.,

Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. (2023). Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs].

Ucci, D., Aniello, L., and Baldoni, R. (2019). Survey of machine learning techniques for malware analysis. *Computers & Security*, 81:123–147.

Xia, C. S., Paltenghi, M., Tian, J. L., Pradel, M., and Zhang, L. (2023a). Universal Fuzzing via Large Language Models. arXiv:2308.04748 [cs].

Xia, C. S., Wei, Y., and Zhang, L. (2023b). Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. ISSN: 1558-1225.

Xu, X., Zhang, Z., Feng, S., Ye, Y., Su, Z., Jiang, N., Cheng, S., Tan, L., and Zhang, X. (2023). LmPa: Improving Decompilation by Synergy of Large Language Model and Program Analysis. arXiv:2306.02546 [cs].