Kgent: Kernel Extensions Large Language Model Agent

Yusheng Zheng*
Imperial College London
London, United Kingdom
yusheng.zheng@imperial.ac.uk

Yiwei Yang*
University of California, Santa Cruz
Santa Cruz, California, USA
yyang363@ucsc.edu

Maolin Chen
eunomia-bpf Community
Shanghai, China
agaaain.try@gmail.com

Andrew Quinn
University of California, Santa Cruz
Santa Cruz, California, USA
aquinn@ucsc.edu

Abstract

The extended Berkeley Packet Filters (eBPF) ecosystem allows for the extension of Linux and Windows kernels, but writing eBPF programs is challenging due to the required knowledge of OS internals and programming limitations enforced by the eBPF verifier. These limitations ensure that only expert kernel developers can extend their kernels, making it difficult for junior sys admins, patch makers, and DevOps personnel to maintain extensions. This paper presents Kgent, an alternative framework that alleviates the difficulty of writing an eBPF program by allowing Kernel Extensions to be written in Natural language. Kgent uses recent advances in large language models (LLMs) to synthesize an eBPF program given a user's English language prompt. To ensure that LLM's output is semantically equivalent to the user's prompt, Kgent employs a combination of LLM-empowered program comprehension, symbolic execution, and a series of feedback loops. Kgent's key novelty is the combination of these techniques. In particular, the system uses symbolic execution in a novel structure that allows it to combine the results of program synthesis and program comprehension and build on the recent success that LLMs have shown for each of these tasks individually.

To evaluate Kgent, we develop a new corpus of natural language prompts for eBPF programs. We show that Kgent produces correct eBPF programs on 80%—which is an improvement of a factor of 2.67 compared to GPT-4 program synthesis baseline. Moreover, we find that Kgent very rarely synthesizes "false positive" eBPF programs—i.e., eBPF programs that Kgent verifies as correct but manual inspection reveals to be semantically incorrect for the input prompt. The code for Kgent is publicly accessible at https://github.com/eunomia-bpf/KEN.

*Both authors contributed equally to this research.

eBPF '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0712-4/24/08
https://doi.org/10.1145/3672197.3673434

CCS Concepts

• **Software and its engineering** → **Formal software verification**;
• **Computing methodologies** → **Natural language processing**.

Keywords

Large Language Model, eBPF, Symbolic Execution

ACM Reference Format:
Yusheng Zheng, Yiwei Yang, Maolin Chen, and Andrew Quinn. 2024. Kgent: Kernel Extensions Large Language Model Agent. In *Workshop on eBPF and Kernel Extensions (eBPF '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3672197.3673434

1 Introduction

Developers are increasingly tasked with modifying and extending operating system kernels to improve performance, security, and reliability, or introduce new features to their systems. Extended Berkeley Packet Filters (eBPF) have emerged as the de facto method for extending an operating system, with recent support for both Linux and Windows [11]. eBPF programs inject new logic that is executed before or after existing kernel logic to observe or modify the kernel's behavior. eBPF programs were originally used to trace network traffic, but the ecosystem now provides sufficient power to implement a variety of features including performance monitoring performance [17, 18], detecting intrusion detection [2, 21], and application-specific logic [15, 22, 35, 37].

Unfortunately, eBPF programs are difficult to write correctly. Implementing an eBPF program requires intimate knowledge of kernel internals to identify where to inject logic [9]. Additionally, the eBPF verifier, intended to prevent unsafe eBPF programs from executing on a system, imposes a number of unfortunate programming constraints on eBPF programmers: programs can only use limited control flow (e.g., loops must have constant bounds) and limited data accesses (e.g., the program cannot access arbitrary memory). Consequently, eBPF is not even a Turing Complete language [23].

In this paper, we present Kgent, **K**ernel Extensions LLM A**gent** that alleviates the difficulty of eBPF. A user can then extend their kernel by injecting the eBPF program produced by Kgent without having to understand eBPF or a kernel's internals.

2 Background

This section describes background on the key techniques that Kgent employs.

Few-shot In-Context Learning: systems that employ in-context Learning adapt an LLM to a new target area by adding minimal contextual data to an LLM (e.g., they prompt "few shots" LLM with a small corpus of particularly pertinent inputs) and carefully crafted prompts rather than training a custom LLM on a large dataset. In-context learning is more cost-effective and can new data more quickly than traditional alternatives. The key challenges in employing in-context learning in a new synthesis domain involve identifying a corpus of critical examples and crafting a prompt strategy that guides the underlying LLM [4, 26].

Feedback Loops: many systems employ a feedback-driven approach to validate an LLM's output against some specification and pass feedback to the model to refine the output. For example, HarmonyOS developers [33], and TPUv4 designers [29] use unit tests to evaluate whether an LLM's output is correct. The fundamental challenge in developing such ground truth unit tests—manually crafting a set of unit tests that can sufficiently guide the LLM is likely as difficult as manually writing the ideal synthesized output. Developers in the aforementioned examples were able to use unit tests that were already created for their use cases. Additionally, Self-Debugging includes a mechanism to provide feedback by using an LLM to explain the behavior of synthesized programs. The system refines its output by iteratively synthesizing programs, explaining the synthesized program checking its unit-test outputs, and then iteratively re-synthesizing.

Automated Program Comprehension: automated program comprehension determines the properties of a program's execution without requiring developer effort. Early work [24] uses a counter-example-driven approach, which observes a program under test and derives invariants that hold over all executions. Recent works show that LLMs are effective at program comprehension tasks [1, 12, 13, 25, 34, 38], but their output is typically an English description of the program.

Symbolic Execution: automated program verification has emerged to ensure that programs are correct without requiring a developer to write a copious amount of tests. There is a wide range of existing solutions including model checking [7, 28], fuzzing [14], and symbolic execution (symbex) [3, 19]. Kgent uses symbex because symbolic execution has been shown to be highly effective in finding issues both in operating systems [6] and user-programs [3, 19].

A symbolic execution engine reasons the behavior of a function or program to determine whether the program upholds specific properties (e.g., memory safety). A symbex engine associates a symbolic value with each variable in the program. As the program executes, the engine gathers constraints on the symbolic values (e.g., variable x is less than 3). During execution, the engine uses an SMT solver to determine if a given property (e.g., "this function will be executed under this path condition?") can be violated given the constraints that the engine gathered. Thus, symbolic execution can reason about large classes of inputs to a function or program without needing to execute each input individually.

3 System Design

This section describes the design and implementation of Kgent; Figure 1 depicts the system's key components and how they interact.

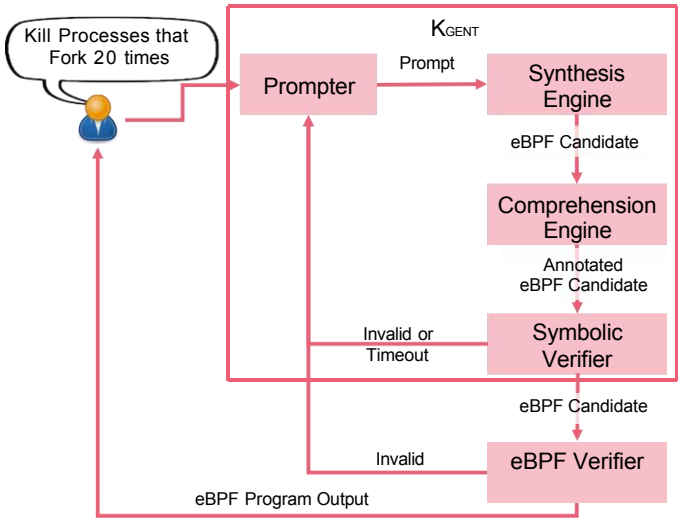

Figure 1: The Workflow of Kgent

Kgent consists of four main components: a Prompter (section 3.2), responsible for constructing prompts that generate eBPF programs; a Synthesis Engine (section 3.3), responsible for synthesizing a candidate eBPF program given a natural language prompt; a Comprehension Engine (section 3.4), responsible for annotating a candidate eBPF program with accurate Hoare-logic conditions for each of the kernel functions with which the candidate interacts; and the Symbolic Verifier (section 3.5), responsible for ensuring that the candidate eBPF program upholds the Hoare-logic annotations. The system also uses the existing eBPF verifier to ensure that the eBPF candidate meets basic security criteria.

Kgent uses in-context learning (section 2) to augment existing LLMs without requiring training of an entirely new model. This design allows Kgent to remain LLM agnostic. Supporting in-context learning requires two technical contributions: prompting strategies that guide the LLMs to produce correct output and new datasets for the Synthesis and Comprehension Engines.

Kgent uses a feedback-driven approach to iteratively synthesize a correct eBPF program. Kgent includes two such feedback loops. First, if the Symbolic Verifier determines that a candidate eBPF program from the Synthesis Engine does not uphold the Hoare-logic conditions from the Comprehension Engine or if the Symbolic Verifier times out, then the Verifier will pass the failure back to the Prompter. This feedback loop allows both Engines additional chances to synthesize correct output. The second feedback is from the eBPF verifier to the Prompter and is taken when the eBPF verifier does not verify that the synthesized eBPF program is safe.

Each of Kgent's components is an adoption of state-of-the-art techniques to the problem of eBPF program synthesis. Kgent's key novelty lies in its combination of these techniques which empowers a useful and efficient program synthesis tool. In particular, Kgent's use of symbolic execution to combine the results of program synthesis and program comprehension is a novel structure that allows Kgent to build on the power that LLMs have shown on both tasks individually.

The rest of this section proceeds as follows. We first describe the workflow of synthesizing an eBPF program for a user prompt (section 3.1). Then, we discuss each of Kgent's components (section 3.2–section 3.6). Finally, we describe Kgent's implementation details (section 3.7).

3.1 Workflow

The high-level workflow for synthesizing an eBPF program in Kgent is as follows. The user issues a prompt to the Prompter, which forwards the user's prompt to Kgent's Synthesis Engine. The Synthesis Engine consults an LLM to synthesize a candidate eBPF program based upon the user's input. Kgent passes the candidate eBPF program to the Comprehension Engine, which consults an LLM to annotate the candidate eBPF program with Hoare-logic pre- and post-conditions for each of the Kernel functions that is referenced in the candidate eBPF program. Kgent passes this annotated eBPF candidate to its Symbolic Verifier, which validates that the synthesized eBPF program satisfies the Hoare-logic properties. If the Symbolic Verifier determines that the eBPF program does not uphold the semantic properties, or if the Symbolic Verifier times out, then the Symbolic Verifier passes its output to the Prompter to begin another iteration of Kgent. If the Symbolic Verifier succeeds, it passes the eBPF program to the eBPF verifier to validate the program's safety properties. The eBPF verifier will pass its error message to the Prompter to begin another iteration of Kgent if the eBPF program is deemed unsafe.

If Kgent fails to synthesize a verified eBPF program after a configured number of trials (3 is the default), the system will re-prompt the user to include additional information. Anecdotally, we observe that including small additional semantic hints (e.g., hinting at the expected size of some variables) can often resolve Kgent's synthesis issues.

3.2 Prompter

The Prompter takes the input prompt from the user and specially formats it to pass to the synthesis engine. In particular, the Prompter adds boilerplate text instructing Kgent to produce an eBPF program written for bpftrace framework. Additionally, the Prompter appends all error messages that it has received for the current synthesis task by all feedback loops. For example, if the Symbolic Verifier failed in a first iteration with message failure1 and the eBPF verifier failed in a second iteration with message failure2, then the Prompter will append both failure1 and failure2 to its message before sending it to the Synthesis Engine.

3.3 Synthesis Engine

The Synthesis Engine takes a natural language prompt and consults an LLM to generate a candidate eBPF program. The engine uses LangChain [20] as a mechanism for interacting with an arbitrary LLM, which allows Kgent to support a variety of privacy-cost-performance tradeoffs.

Like other systems [5], the Synthesis Engine uses a VectorDB (e.g., Milvus [31]) to enable in-context learning. The engine stores prompt-eBPF pairs from the eBPFNLDataset dataset (see below) in its VectorDB. On each query, the engine uses the VectorDB to identify the prompt-eBPF pairs that are most similar to the user prompt and includes these pairs as examples of correct input-output pairs in its query to the large language model. Thus, Kgent is similar to few-shot learning [32]. Kgent also updates its VectorDB after each synthesis, which allows the system to learn from its successful and failed eBPF syntheses.

By specifying the desired syntax in the LLM prompt, we find that the Synthesis Engine almost always synthesizes the correct syntax. Our results show that it also often synthesizes correct semantics (section 5).

eBPFNLDataset—an eBPF synthesis Dataset. The Synthesis Engine is empowered by eBPFNLDataset, a novel dataset of 145 natural language prompts paired with corresponding eBPF programs, 79 of which are bpftrace programs and 66 of which are libbpf programs. eBPFNLDataset was gathered from two sources. First, 65 pairs (39 bpftrace, 26 libbpf) come from a popular eBPF developer blog [16]. The other 80 pairs (40 bpftrace, 40 libbpf) are hardwritten based upon examples from well-known open-source eBPF project repositories, such as bcc[27], bpftrace[30], ebpf-exporter[8], or bpftime[36].

3.4 Comprehension Engine

The Comprehension Engine annotates eBPF candidate programs with Hoare-logic pre- and post-conditions using an LLM prompt that includes the eBPF function, developer prompt, and conditions from the KernelCompDataset. This approach allows the engine to utilize the developer's prompt, smooth out inaccuracies in the automatically generated KernelCompDataset, and learn from its mistakes and successes by updating an internal VectorDB. Empirical evaluationsection 5 shows that the Comprehension Engine does not nefariously adjust pre-/post-conditions to tautologically verify all candidate programs, likely due to the slow learning of the VectorDB.

KernelCompDataset—a dataset of Hoare-logic contracts for kernel functions. Manually creating a dataset of Hoare-logic contracts for every eBPF instrumentable kernel function would be infeasible. There are hundreds of such functions, and the functions themselves change with each new kernel release. Thus, we chose to generate KernelCompDataset using an automated approach that we can then adapt to new kernel versions. We use a regular expression to identify every function that has a symbol in the kernel. To approximate the function's semantics, we use a regular expression to find any comments immediately before the function. We pass the function prototype, source code, and approximate semantics into an LLM using a prompt that asks for Z3 compatible conditions for each of the eBPF helpers. We store the approximate semantics, prototype, and the output from the LLM in a JSON format in the KernelCompDataset. If detected, developers could fix inaccuracies from the automated approach, although we have not needed to do so in Kgent.

3.5 Symbolic Verifier

The symbolic verifier uses symbolic execution to validate the annotated eBPF candidate program produced the Comprehension Engine. If the symbolic verifier determines that the program upholds the assert statements, it removes the assert/assume statements and passes

Prompt: Write a bpftrace program to trace tcp_connect events for both IPv4 and IPv6 connection attempts, display the source and destination IP addresses and the source and destination ports in host byte order.

Figure 2: A prompt passed into KGENT instructing it to print basic connection information for all TCP connect attempts.

```
"kretprobe:tcp_connect_init": {
  "pre": {
    "sk": "!=null",
  }
}
```

Figure 3: JSON for the matching element in KERNELCOMPDATASET for the kernel probe

```
1  kprobe:tcp_connect {
2    $sk = (struct sock *) arg0;
3    assume($sk != 0);
4    assume($sk->___sk_common.skc_rcv_saddr != 0);
5    assume($sk->___sk_common.skc_daddr != 0);
6    assume($sk->___sk_common.skc_num >= 0);
7    assume($sk->___sk_common.skc_dport >= 0);
8    assume(sizeof($sk->___sk_common.skc_rcv_saddr) == 4
     ↪ || sizeof($sk->___sk_common.skc_rcv_saddr) == 16);
9    assume(sizeof($sk->___sk_common.skc_daddr) == 4 ||
     ↪ sizeof($sk->___sk_common.skc_daddr) == 16);
10   $saddr = ntop(2, $sk->___sk_common.skc_rcv_saddr);
11   $daddr = ntop(2, $sk->___sk_common.skc_daddr);
12   $sport = ($sk->___sk_common.skc_num);
13   $dport = ($sk->___sk_common.skc_dport);
14   printf("TCP connect: %s:%d -> %s:%d\n", $saddr, $sport,
     ↪ $daddr, $dport);
15   assert($dport ==
     ↪ bswap($sk->___sk_common.skc_dport));
16   assert($sport == bswap($sk->___sk_common.skc_num));
17 }
```

Figure 4: Annotated candidate eBPF program produced by the Comprehension Engine for verification. The original synthesized eBPF program is shown in gray. Pre-conditions inferred from KERNELCOMPDATASET are shown in pink. Pre-conditions inferred from the user prompt are shown in yellow. Finally, post-conditions inferred from the user's prompt are shown in blue.

```
kprobe:tcp_connect {
  $saddr = ntop(2, $sk->___sk_common.skc_rcv_saddr);
  $daddr = ntop(2, $sk->___sk_common.skc_daddr);
  $sport = (bswap($sk->___sk_common.skc_num));
  $dport = (bswap($sk->___sk_common.skc_dport));
  printf("TCP connect: %s:%d -> %s:%d\n", $saddr, $sport,
  ↪ $daddr, $dport);
}
```

Figure 5: The output of the Synthesis Engine on the second iteration for the prompt of fig. 2 and the error message for the symbolic verification failure of fig. 4

the candidate eBPF program to the eBPF verifier. If the symbolic verifier finds an assertion statement that is not upheld or times out, then it passes its error message to the Prompter.

3.6 eBPF Verifier

KGENT uses the existing eBPF Verifier from the operating system. Namely, the eBPF Verifier validates that the eBPF candidate program produced by the Symbolic Verifier contains no unbounded loops or arbitrary memory accesses. If the eBPF verifier is unable to assure that the candidate program is safe, it passes its error message back to the Prompter. Otherwise, the eBPF verifier passes the eBPF candidate program to the user as its final synthesized eBPF program.

3.7 Implementation

We implement KGENT, EBPFNLDATASET, and KERNELCOMPDATASET using 4244 LOC in Python, we add 51 LOC to the bpftrace compiler to add support for assume and assert functions. KGENT uses SeaHorn [19] as its Symbolic Verifier and Z3 [10] as its backend SMT solver. SeaHorn symbolically executes LLMV IR; KGENT uses SeaHorn by compiling the output of the Comprehension Engine to LLVM IR.

KGENT supports a multitude of LLM, but employs GPT-4 API by default. Our current KGENT prototype only works on Linux. KGENT can synthesize eBPF programs for both libbpf and bpftrace, but it defaults to bpftrace because we found that KGENT is more effective at synthesizing bpftrace programs (section 5).

4 Case Study

This section describes a case study of using KGENT to provide an example of how each of KGENT's steps works. We crafted this scenario specifically to illustrate KGENT's behavior; it is not a test from EBPFNLDATASET.

In this case study, a user wishes to display information to their terminal about all of the TCP connections that are made to their webserver so that they can detect denial of service attacks that are made against it. Kernel extensions offer a good mechanism for performing full system telemetry. However, since the developer is not a Linux kernel expert, they turn to KGENT to help them build their eBPF program. Figure 2 shows the developer's prompt that the developer issues to KGENT. They ask the system to trace all IPv4 and IPv6 tcp_connect events and specify their desired output including the proper order for each port.

5 Evaluation

KGENT employs GPT-4 API 2023 September version as its LLM. Additionally, unless otherwise specified, we train the system using KERNELCOMPDATASET and the set of prompt-eBPF program pairs from

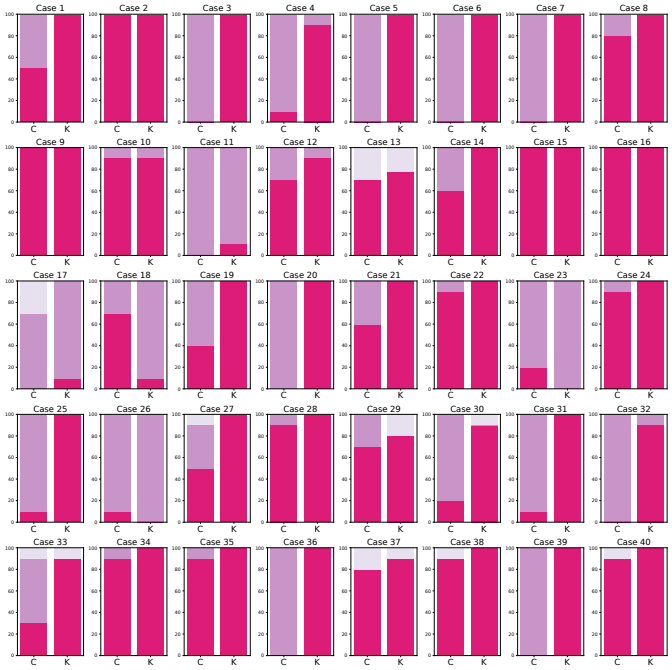

Figure 6: The per-prompt effectiveness of KGENT (K) and a GPT-4 (C). Each bar chart shows the percentage of time that KGENT/GPT-4 synthesizes an accurate(red), false negative (pink), or false positive(grey) eBPF program for each prompt over ten trials.

eBPFNLDATASET that were scrapped from the popular web blog [16] and test the system using the set of prompt-eBPF program pairs from eBPFNLDATASET that we created anew. We create a baseline that synthesizes eBPF programs using a single prompting of GPT-4 and verifies the output eBPF program by using the built-in eBPF verifier.

Since each prompt can be correctly implemented in multiple ways, we manually inspect KGENT's synthesized eBPF program for each prompt to determine if the output correctly implements the prompt. We calculate the *Accuracy (A)* of KGENT for each experiment, which is the fraction of prompts for which KGENT synthesized a correct eBPF program.

To understand the consequence of KGENT's incorrect outputs, we split the prompts for which KGENT fails to correctly synthesize an eBPF program into two categories: *False Negative (FNs)*, which are the percentage of prompts for which KGENT fails to synthesize a verified eBPF program, and *False Positives (FPs)*, which are the percentage of prompts for which KGENT synthesizes a verified eBPF program that does not correctly implement the prompt. Conceptually, FPs represent a safety violation since a developer using KGENT may extend their kernel incorrectly when KGENT produces a false positive. In contrast, FNs represent a liveness violation since a developer is effectively unable to use KGENT for such prompts.

System	A	FP	FN
GPT-4 few shot	30%	2.5%	67.5%
GPT-4+Feedback	60%	7.5%	32.5%
GPT-4+Feedback+Symbex	77.5%	5%	17.5%
Human Expertise	72.5%	2.5%	25%
KGENT	80%	2.5%	17.5%

Table 1: The Breakdown Accuracy Analysis of KGENT

5.1 KGENT Effectiveness

For each prompt in the eBPFNLDATASET test set, fig. 6 shows a bar depicting the percentage of time that KGENT and the GPT-4 baseline are accurate, produce a false positive, and produce false negative across 10 iterations of the prompt. Inspecting the results for each individual test case, we observe that KGENT generates a correct program more often than the baseline in 37 of the 40 test cases. In 11 of 40 test cases, KGENT improves the accuracy rate by more than a factor of 9 (i.e., the accuracy rate improves from at most 10% in the baseline to at least 90% in KGENT). In addition, KGENT improves on the false positive rate in 6 of the 9 test cases in which either system observes a false positive.

5.2 The Effectiveness of KGENT's Design Decisions

We first evaluate the impact of each of KGENT's high-level design decisions, then describe an experiment showing the benefit of KGENT's comprehension and symbolic execution engines, and finally describe the impact of synthesizing eBPF programs to use bpftrace instead of libbpf.

5.2.1 Effectiveness of Kgent's High-Level Design Decisions. Table table 1 shows how each high-level design decision impacts KGENT's effectiveness. Each row in the table represents a different configuration. We start from the GPT-4 few shot baseline and apply KGENT's design features in the the order of largest impact on accuracy. Namely, we first include KGENT's model-guided feedback, then KGENT's comprehension and symbolic execution components (symbex), and finally add training using the blog-gathered portion of the eBPFNLDATASET dataset. We note that it is not meaningful to separate KGENT's comprehension engine from its symbolic execution components.

The results indicate that model-guided feedback plays a large role in improving the accuracy of KGENT, as it improves the accuracy by a factor of 2 (from 30%to 60%). However, this increase in accuracy comes with a factor of 3 increase in false positive rate (from 2.5% to 7.5%). Including the comprehension engine and symbolic execution component also improves KGENT's effectiveness substantially—accuracy improves to 77.5%, while the false positive rate moves to 5%. Including the eBPFNLDATASET dataset in training comes with a relatively small impact on KGENT's accuracy—it only improves by 2.5%. However, training using the eBPFNLDATASET dataset does bring KGENT's false positive rate back down to the baseline of 2.5%

5.2.2 Effectiveness of Kgent's Comprehension and Symbolic Execution Engine. To identify the power of KGENT's automated reasoning, we create a baseline that replaces the comprehension and symbolic execution engines with developer expertise. We augment

the eBPFNLDataset test set with correct kprobe and kretprobe locations for each prompt and use this augmented dataset as input to the GPT-4 + feedback + dataset baseline, producing a *human expertise* baseline. Kgent demonstrates higher accuracy without additional false positives compared to the human expertise baseline, despite not directly passing the output of the comprehension and symbolic execution engines into its synthesis engine. We hypothesize that Kgent's feedback loop, initiated on each symbolic execution failure, provides the synthesis engine with sufficient knowledge to replace the human expertise from the baseline, revealing a powerful synergy between Kgent's feedback feature and its automated reasoning features.

5.2.3 Effectiveness of using bpftrace instead of libbpf. We compare the synthesis of eBPF programs using bpftrace and libbpf. Although libbpf offers more flexibility, it also introduces more programming complexity. Due to state explosion caused by frequent helper function usage in libbpf programs, Kgent's symbolic execution engine rarely terminates on them. When not using Kgent's comprehension and symbolic execution engines, the accuracy of libbpf programs is 37.5%, while bpftrace programs achieve 60% accuracy. Additionally, libbpf programs take an average of 2.16 seconds to synthesize, compared to 1 second for bpftrace programs. We conclude that bpftrace is a better synthesis target for our eBPF use cases.

6 Conclusion

In conclusion, we presented Kgent, a system that helps developers extend their kernels with eBPF by using symbolic execution to integrate results from LLM-based program synthesis and comprehension. Additionally, we produce datasets valuable for building and evaluating Kgent and future eBPF program synthesis work. Future work involves enhancing Kgent's performance and usability, expanding its datasets, and exploring real-world applications to further advance eBPF program synthesis.

References

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[2] Maximilian Bachl, Joachim Fabini, and Tanja Zseby. 2021. A flow-based IDS using Machine Learning in eBPF. *CoRR* abs/2102.09980 (2021). arXiv:2102.09980 https://arxiv.org/abs/2102.09980

[3] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[4] Yiannis Charalambous, Norbert Tihanyi, Ridhi Jain, Youcheng Sun, Mohamed Amine Ferrag, and Lucas C Cordeiro. 2023. A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. *arXiv preprint arXiv:2305.14752* (2023).

[5] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[7] Edmund M. Clarke and E. Allen Emerson. 1982. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, Dexter Kozen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–71.

[8] Cloudflare. 2023. ebpf_exporter: eBPF-based exporter for Prometheus. GitHub repository. https://github.com/cloudflare/ebpf_exporter.

[9] Alibaba Cloud Native Community. 2023. Seven Core Issues about eBPF. https://www.alibabacloud.com/blog/seven-core-issues-about-ebpf_599668.

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[11] eBPF for Windows Contributors. 2023. eBPF for Windows. https://github.com/microsoft/ebpf-for-windows.

[12] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing. arXiv:2104.02443 [cs.SE]

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong (YIMING), Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of EMNLP 2020*. https://www.microsoft.com/en-us/research/publication/codebert-a-pre-trained-model-for-programming-and-natural-languages/

[14] fuzzing book author. [n. d.]. The fuzzing book: Concolic Fuzzing. https://www.fuzzingbook.org/beta/html/SymbolicFuzzer.html.

[15] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. {BMC}: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 487–501.

[16] Brenden Gregg. 2001. Brenden Gregg's Homepage. https://www.brendangregg.com/.

[17] Brenden Gregg. 2016. Linux Extended BPF (eBPF) Tracing Tools. https://www.brendangregg.com/ebpf.html.

[18] Brendan Gregg. 2021. Computing Performance. (2021).

[19] Arie Gurfinkel, Temesghen Kahsai, and Jorge A Navas. 2015. SeaHorn: A framework for verifying C programs (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 447–450.

[20] Robusta Intellegence. [n. d.]. LangChain. https://www.langchain.com/.

[21] Jinghao Jia, Michael V Le, Salman Ahmed, Dan Williams, and Hani Jamjoom. 2023. Practical and Flexible Kernel CFI Enforcement using eBPF. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*. 84–85.

[22] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. 2023. Programmable System Call Security with eBPF. *arXiv preprint arXiv:2302.10366* (2023).

[23] Andrea Mayer, Pierpaolo Loreti, Lorenzo Bracciale, Paolo Lungaroni, Stefano Salsano, and Clarence Filsfils. 2021. Performance Monitoring with H^2: Hybrid Kernel/eBPF data plane for SRv6 based Hybrid SDN. *Computer Networks* 185 (2021), 107705. https://doi.org/10.1016/j.comnet.2020.107705

[24] Jeff H. Perkins and Michael D. Ernst. 2004. Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering* (Newport Beach, CA, USA) *(SIGSOFT '04/FSE-12)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/1029894.1029901

[25] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645* (2021).

[26] Gabriel Poesia, Kanishk Gandhi, Eric Zelikman, and Noah D Goodman. 2023. Certified Reasoning with Language Models. *arXiv preprint arXiv:2306.04031* (2023).

[27] IO Visor Project. 2023. BPF Compiler Collection (bcc). Available: https://github.com/iovisor/bcc.

[28] J. P. Queille and J. Sifakis. 1982. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, Mariangiola Dezani-Ciancaglini and Ugo Montanari (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–351.

[29] Agam Shah. [n. d.]. Google TPU v5e AI Chip Debuts after Controversial Origins. https://www.enterpriseai.news/2023/08/31/google-tpu-v5e-ai-chip-debuts-after-controversial-origins/.

[30] IO Visor. 2023. bpftrace: High-level tracing language for Linux eBPF. GitHub repository. https://github.com/iovisor/bpftrace.

[31] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[32] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)* 53, 3 (2020), 1–34.

[33] Wikipedia. [n. d.]. The Wikipedia of HarmonyOS. https://en.wikipedia.org/wiki/HarmonyOS.

[34] Hongqiu Wu, Hai Zhao, and Min Zhang. 2020. Code summarization with structure-induced transformer. *arXiv preprint arXiv:2012.14710* (2020).
[35] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. 2023. {λ-IO}: A Unified {IO} Stack for Computational Storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 347–362.
[36] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, and Andrew Quinn. 2023. bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions. arXiv:2311.07923 [cs.OS]
[37] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 375–393. https://www.usenix.org/conference/osdi22/presentation/zhong
[38] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318* (2021).