Pacific Northwest
NATIONAL LABORATORY

PNNL-36775

Assurance of Reasoning Enabled Systems (ARES)

MARS Initiative Report

September 2024

Andrés Márquez
Tim J Stavenger
Ted C Fujimoto

U.S. DEPARTMENT OF ENERGY

Prepared for the U.S. Department of Energy
Under contract DE-AC05-76RL01830

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from
the Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062
www.osti.gov
ph: (865) 576-8401
fox: (865) 576-5728
email: reports@osti.gov

Available to the public from the National Technical Information Service
5301 Shawnee Rd., Alexandria, VA 22312
ph: (800) 553-NTIS (6847)
or (703) 605-6000
email: info@ntis.gov
Online ordering: http://www.ntis.gov

PNNL-36775

Assurance of Reasoning Enabled Systems (ARES)

MARS Initiative Report

September 2024

PI Andrés Márquez
Ted C Fujimoto
Tim J Stavenger

Prepared for
the U.S. Department of Energy
Under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99354

Abstract

ARES was in part motivated by the determination of President's Council of Advisors on Science and Technology (PCAST) on May 13th, 2023 that published a set of inquiries:

In an era in which convincing images, audio, and text can be generated with ease on a massive scale, how can we ensure reliable access to verifiable, trustworthy information? How can we be certain that a particular piece of media is genuinely from the claimed source? What technologies, policies, and infrastructure can be developed to detect and counter AI-generated disinformation?

In an effort to automatically analyze and patch/optimize code the work in this report describes various neural Machine Learning (ML) analysis engine implementations to assist in situations where source code is deficient or completely lacking to decompile (lift) binary code to 'C'. The goal is to gradually reduce human intervention. To this end, two Large Language Model (LLM) variants (Code LLama 2, LLama 3.1 and Starcoder1, Starcoder 2) where finetuned with 'before/after' code pairs on the OpenBLAS library. LLama trained on the lowering process, Starcoder trained on the lifting process with National Security Agency's (NSA) open-source Ghidra decompiler assist. The inferencing test results indicate correctness for only very short sequences for Starcoder 2. Moving forward, the experiments conclude with a set of recommendations of required resources and technologies.

1.0 Summary

ARES was in part motivated by the determination of President's Council of Advisors on Science and Technology (PCAST) on May 13th, 2023 that published a set of inquiries:

In an era in which convincing images, audio, and text can be generated with ease on a massive scale, how can we ensure reliable access to verifiable, trustworthy information? How can we be certain that a particular piece of media is genuinely from the claimed source? What technologies, policies, and infrastructure can be developed to detect and counter AI-generated disinformation?

To assist a code analysis engine in situations where source code is deficient or completely lacking, as in cases of binary-library code patching, reoptimization or machine retargeting, ARES lifts code from object code to C. Given a function in object-code, the 'Abstraction Lifter (AL) will translate code to C by applying ML decompilation inferencing. The goal is to gradually reduce classical decompilation techniques that heavily rely on human intervention. Ideally, AL can discern and separate business logic from scheduling logic; hereby relaxing subsequent recompilation that is unencumbered by existing optimization constraints. In the best case scenario, AL would be able to directly apply patches/reoptimization or retargeting without any ancillary support.

The following studies explore neural decompilation strategies publicly available at time of writing and indicate a path forward for future work. Two Large Language Model (LLM) variants (Code LLama 2, LLama 3.1 and Starcoder1, Starcoder 2) where finetuned with 'before/after' code pairs on the OpenBLAS library. LLama trained on the lowering process, Starcoder trained on the lifting process with National Security Agency's (NSA) open-source Ghidra decompiler assist. The inferencing test results indicate correctness for only very short sequences for Starcoder 2.

The ML landscape is changing fast. In fact, during the course of the one-year project, pretrained models and pretrained datasets sourced externally changed multiple times. Nevertheless, the performed experiments and literature search inform some recommendations going forward, including the extent of ample staff and resource requirements to achieve the ML decompilation goal. Additionally, current research in the community would indicate the possibility of a principled approach toward a successful LLM decompiler by leveraging automated correctness provers.

2.0 Acknowledgments

This research was supported by the **MARS Initiative**, under the Laboratory Directed Research and Development (LDRD) Program at Pacific Northwest National Laboratory (PNNL). PNNL is a multi-program national laboratory operated for the U.S. Department of Energy (DOE) by Battelle Memorial Institute under Contract No. DE-AC05-76RL01830.

Acronyms and Abbreviations

AMAIS	Advanced Memory to Support AI for Science
ARES	Assurance of Reasoning Enabled Systems
AST	Abstract Syntax Tree
AI	Artificial Intelligence
AL	Abstraction Lifter
BLAS	Basic Linear Algebra Subprograms
CFG	Control Flow Graph
CVF	Computer Vision and Pattern Recognition Conference (CVPR)
CPU	Central Processing Unit
DFG	Dataflow Graph
IEEE	Institute of Electrical and Electronics Engineers
IR	Internal Representation
LAPACK	Linear Algebra Package
LLM	Large Language Model
LLVM	Low Level Virtual Machine
NLP	Natural Language Processing
NSA	National Security Agency
RSA	Ron Rivest, Adi Shamir and Leonard Adleman conference
ML	Machine Learning
PCAST	President's Council of Advisors on Science and Technology
PDG	Program Dependence Graph
PNNL	Pacific Northwest National Laboratory
SLED	Specification Language for Encoding and Decoding

Contents

Figures

3.0 Introduction

3.1 Retrospective

ARES was in part motivated by the determination of President's Council of Advisors on Science and Technology (PCAST) on May 13th, 2023 that published a set of inquiries:

In an era in which convincing images, audio, and text can be generated with ease on a massive scale, how can we ensure reliable access to verifiable, trustworthy information? How can we be certain that a particular piece of media is genuinely from the claimed source? What technologies, policies, and infrastructure can be developed to detect and counter AI-generated disinformation?

3.1.1 Original Proposal Aims

Artificial Intelligence (AI) enables new data analysis capabilities that accelerate society's transition in all its forms – including science – from a first principles, rules-based base to a data driven approach. In particular, the transition driven by Machine Learning (ML) harbors great opportunities, as well as risks. An inherent risk of data driven outcomes results from weakness explaining or deriving ML models and input data with sufficient rigor such to warrant a certain level of trust. While active research is under way to shed some light into black box AI modeling, the work proposed here seeks to attack the 'lack-of-trust' problem by a systems approach via an analysis engine.

3.1.1.1 Base Goal: Data Analysis for Identification of Forgeries

- signature generation & watermarking [5]
- integrity & security checking [4, 2]

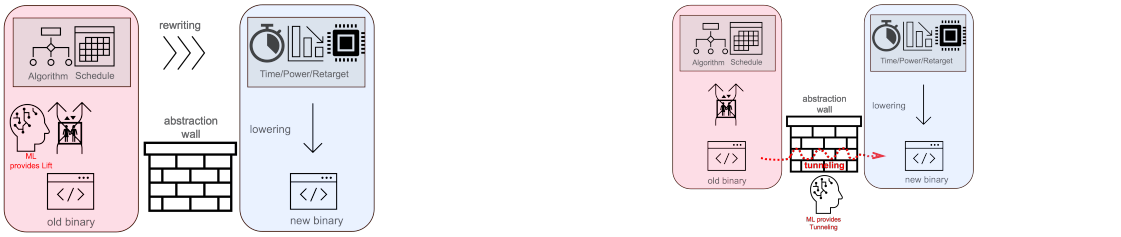
3.1.1.2 Stretch Goal: Code Analysis

The analysis engine can be expanded to capture execution runs at multiple scales. At the dataflow scale, a large set of potential optimizations can be entertained – constant folding, common subexpression elimination, control simplification, superfluous ops removal, algebraic simplifications [14]. At ISA scale, a formulation that separates the concerns of algorithm and schedule would provide a basis for automatic optimization [6].

3.1.1.3 Goal Evolution

At inception of the one-year project, the 'Base Goal' was outsourced to another project – 'Advanced Memory to Support AI for Science - AMAIS' – with an exemplar of image integrity checking, called 'Disharmony: Data Forensics using Reverse Lighting Harmonization', spearheaded by Philip Wootaek Shin, Vijaykrishnan Narayanan, Jack Sampson from Pennsylvania State University and mentored by Mahantesh Halappanavar and Andrés Márquez. The work was submitted to 'IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) 2025', and is under review as of the date of this report's submission.

The reminder of this report discusses research performed on 'Code Analysis'.



(a) Binary to C Lifting

(b) Binary to C Lifting and Optimization Tunneling

Figure 1: Abstraction Lifter (AL)

4.0 Code Analysis

4.1 Approach

To assist the analysis engine in situations where source code is deficient or completely lacking, as in cases of binary-library code patching, reoptimization or machine retargeting, ARES lifts code from object code to C (Figure 1a). Given a function in object-code, the 'Abstraction Lifter (AL) will translate code to C by applying ML decompilation inferencing [26, 17]. The goal is to gradually reduce classical decompilation techniques that heavily rely on human intervention. Ideally, AL can discern and separate business logic from scheduling logic; hereby relaxing subsequent recompilation that is unencumbered by existing optimization constraints [6]. In the best case scenario, AL would be able to directly apply patches/reoptimization or retargeting (Figure 1b) without any ancillary support. The following studies explore neural decompilation strategies publicly available at time of writing and indicate a path forward for future work.

4.2 Datasets

4.2.1 OpenBLAS

OpenBLAS [29] is a portable, open-source (BSD) implementation of the Basic Linear Algebra Subprograms (BLAS) and Linear Algebra Package (LAPACK). It is a fork of GotoBLAS2 1.13 BSD. It is written in Modern Fortran, C and Macro Assembler, the latter specific to a variety of target microprocessor architectures, including X86, POWER, MIPS, SPARC, ARM, RISC-V and IBM-Z variants. The total corpus entails 7111 object files for a specific target after compilation.

4.2.2 BLIS

Similar to OpenBLAS, BLIS [35] is a more recent refactoring of GotoBLAS2. It is written in Modern Fortran, C++, C and Macro Assembler and X86, POWER, ARM, RISC-V variants. A typical corpus for a target comprises 371 object files. It received the prestigious J. H. Wilkinson Prize for Numerical Software in 2023.

4.3 Direct Abstraction Lift - Code LLama - Object Code to C

(T. J Stavenger)

4.3.1 Fine tuning LLM models to optimize assembly code

In an effort coordinated with other parts of the project, this task looked at fine tuning generative AI models for optimizing assembly code. Broadly, the concept was to build a tool that would perform an automated abstraction lift and optimization of code. Ideally such a tool could identify parallelization strategies and memory allocation improvements. Other tasks focused on the abstraction lift, while this researched the feasibility of training an LLM to recognize assembly and offer suggestions for performance improvements. The approach taken was to use existing decompilers to produce assembly, use this dissembled code as training data, fine tune a model, and then test how well the model generalized the performance optimizations using test data.

4.3.1.1 Approach Detail

The OpenBLAS library includes the ability to compile its software to optimized binaries for selected CPU architectures. By using this build feature, and decompiling each into assembly, we produced a training dataset of unoptimized and optimized assembly code pairs. Our data set was split into 29k function pairs for training, 3k for test, and 3k for validation. The training data set pairs were placed into 'Code Llama' and 'Llama 3.1' prompt templates for fine tuning and then tuned on Nvidia A100 80GB GPU.

4.3.1.2 Implementation

The team in initially did not have extensive experience in fine tuning LLMs, so early implementation steps involved becoming familiar with the typical LLM fine tuning workflow. The team chose to use the autotrain-advanced package to limit the coding requirements for fine tuning, allowing the typical coding approach to be applied as completed within this package. Autotrain-advanced provides a "no-code" solution to fine tuning LLMs, giving researchers the ability to provide the train/test/validate data, the base model, and tuning parameters without having to write the code to perform the fine tuning. As the team worked on the task, the available LLMs for code progressed and evolved dramatically. The team began fine tuning 'Code Llama 2' and finished with a fine tuned 'Llama 3.1' model.

4.3.1.3 Results

Initial fine tuning jobs against Code Llama produced poor results. Runs with 'Code Llama' were hampered by poor choice of learning rates and epochs, and possible less flexibility for 'Code Llama' to work with assembly code. The result was an over-trained model that often produced no assembly code at all to optimization prompts.

Subsequent attempts at fine tuning 'Llama 3.1' were more successful, producing a loss rate curve that did not indicate as much overtraining, see Figure 2. This model still struggled with responses to test and validation data. While it did provide assembly code, the provided assembly code was not functionally equivalent to that provided.

4.3.1.4 Future Work

As the team was approaching the end of the work completed, Meta released its LLM Compiler that has the ability to suggest Opt parameters to optimize binary size of provided intermediate representation code. The paper for the model tuned on 546B tokens of LLVM-IR and assembly code using over 1.4M A100 GPU hours. PNNLs internal resources likely would not be able to match this scale of data or compute time, as such future work in this area would likely be best applied to starting with models like Meta's LLM Compiler as its base and working from there on targeted areas related to sponsor mission needs. Broad scope like that researched on this project may continue to struggle with finding significant results without first focusing the scope or scale of the work.

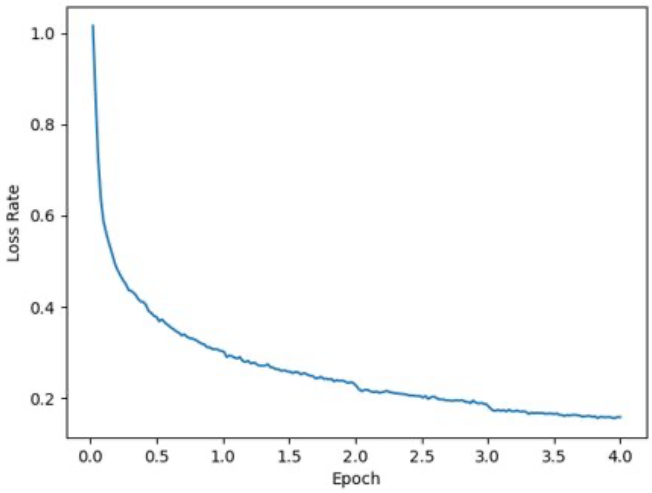


Figure 2: Code Lama LossRate: Object Code to C Lifting

.

4.4 Assisted Abstraction Lift - Starcoder1/2 - P-Code to P-Code
(A. Márquez)

4.4.1 LLM Choice

As another choice for LLM, we chose the gamut of Starcoder code transformers from Huggingface [17], applying the versions that became available during execution of the project. 'Starcoder 1' came out right before ARES started, Starcoder 2, mid-year during the project. Starcoder's largest LLM model is 15B, trained on 1T+ tokens. 'Starcode 2' is also a 15B model, trained on 4+T tokens. The context window is 2^{14} tokens and the sliding window is 2^{12} tokens. The Starcoder family offers an interesting comparison point to the Code LLama option: Starcoder is open and the pretraining datasets ('The Stack 1/2') are permissively-licensed source code files covering 30 programming languages, publicly available and open to introspection. This contrasts with LLama and other commercial code LLM options that might allow for model introspection to some extent but whose training sets are kept opaque. The 'Stack 2' comprises 67.5TB of raw data, after deduplication and curation 32.1TB remain. Introspection allows for analysis of the make-up of the various codes used in the training. The share of assembly code is unfortunately low [3]. We assume this to be the case for all LLM publicly available that target code transformations.

4.4.2 Decompilation Assist: Ghidra + SLEIGH

Starting from the understanding that code LLM are trained to compile – not decompile – and that the corpus of assembler codes is commonly scant, we decided to boost the LLM with traditional decompilation techniques, following established approaches in the field.

Ghidra is a reverse engineering tool developed by the National Security Agency (NSA). The binaries were released at the RSA Conference in March 2019 [1]. It encompasses SLEIGH, an internal language representation (IR) to describe various architectures uniformly to facilitate generalized disassembly and decompilation. It's derived from SLED (Specification Language for Encoding and Decoding). SLEIGH Instructions are described in P-code across the various architectures. To support uniformity, no ISA side-effects, with the exception of memory operations, are allowed. We therefore chose Ghidra to assist the LLM by mapping object code to P-code and creating various decompilation pairs to train the LLM. This offers various benefits creating training data sets, see Figure 3.

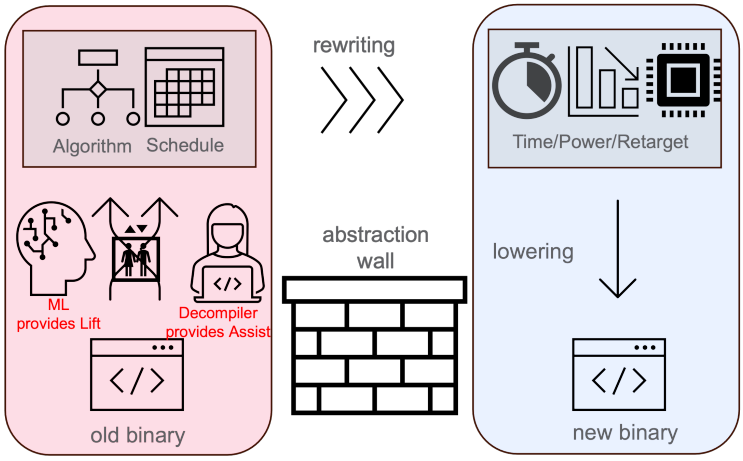
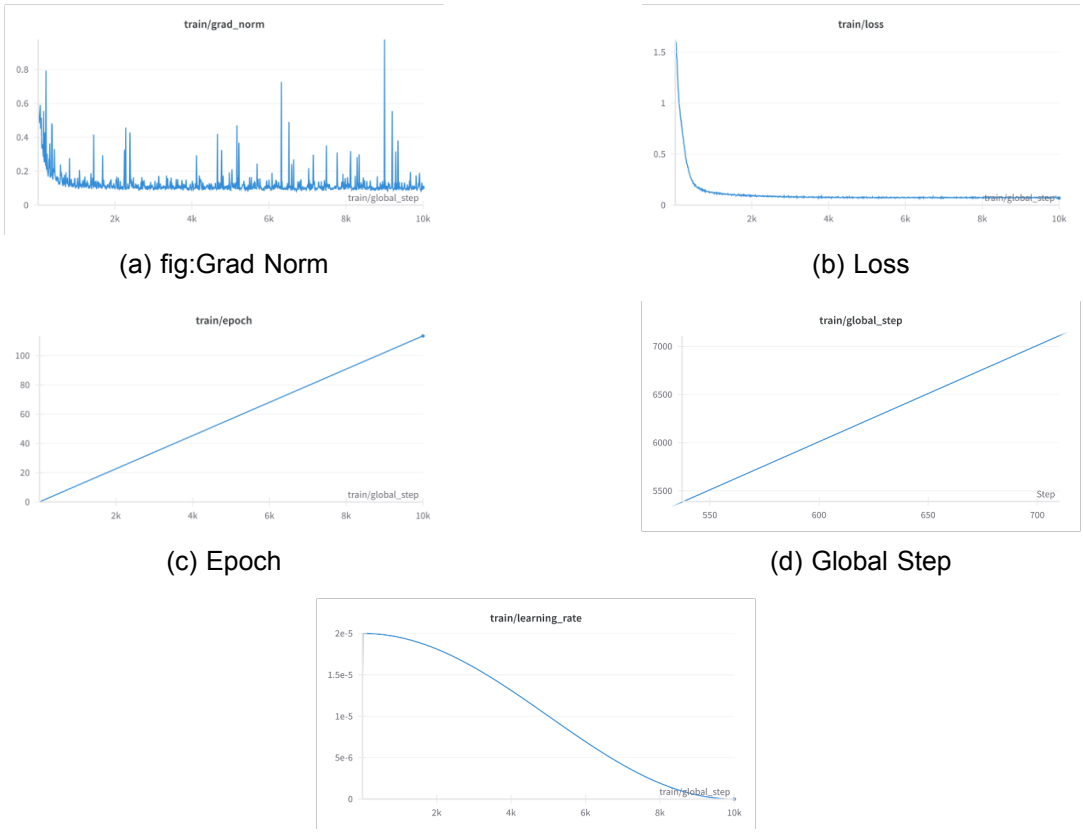


Figure 3: Starcoder 2 assisted by Ghidra

.

- The training occurs on object code/decompilation pairs, as opposed to source/compiled object code



(a) fig:Grad Norm

(b) Loss

(c) Epoch

(d) Global Step

(e) Learning Rate

Figure 4: Finetuning runs Starcoder2-3B: Typical Run

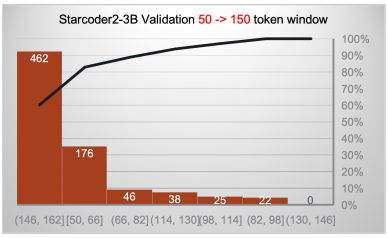
- A unifying object code representation (P-Code) offers a larger corpus, as opposed to object code binned for multiple architectures

4.4.2.1 Approach Detail and Implementation

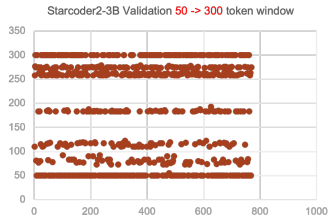
Similar to 4.3.1.1 and 4.3.1.2, we use OpenBLAS with the same training, testing and validation split to finetune 'Starcoder 1/2' 3B models on a Nvidia A100 80GB GPU – the minimum amount of memory. The training pairs for the transformer comprise P-code instantiations at different levels of optimization. Ghidra/Sleigh is used to produce these pairs. The highest level of P-code optimization can be transformed to 'C' and hence could be construed as equivalent to the experiments on 'Code LLama 2' and 'LLama 3.1' described in 4.3. Our factorial design experiment sweeps over output token window sizes of (150, 300, 500, 750) tokens with 50 input token and 'lift levels' of ($\perp$, $\top$). As metric we chose precise matching since code, as opposed to natural languages, is brittle to semantic outcomes as a function of syntax and keywords.

4.4.2.2 Results

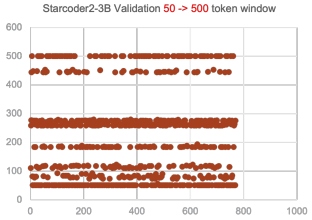
Figures 4 show a typical finetuning run on a Nvidia A100 80GB GPU. The loss function (Figure 4b), after less than 2k epochs, quickly plateaus to zero, with the gradient norm



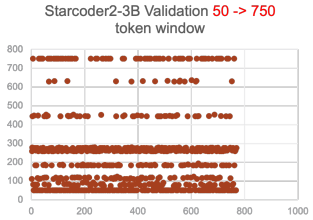
(a) Lift Level $\perp$ -> $\top$, Token Window 50 -> 150



(b) Lift Level $\perp$ -> $\top$, Token Window 50 -> 300



(c) Lift Level $\perp$ -> $\top$, Token Window 50 -> 500



(d) Lift Level $\perp$ -> $\top$, Token Window 50 -> 750

Figure 5: Inference Results: Lift Levels $\perp$ -> $\top$, Output Token Window Sweep from 150 to 750

(Figure 4a) exhibiting some noise. Overall, the learning rate, approaches zero at $\sim$10k epochs (Figure4e).

Figures 5 depict the precise matching results of the factorial design experiment with inputs drawn from the test set source binaries converted to P-code ($\perp$), to the lifted output in P-code with maximum decompilation mustered by Ghidra by applying $\sim$500 term-rewriting rules ($\top$). The output is equivalent to rudimentary 'C'. Each input/output pair is plotted or binned as a function of the maximum matching string index. Figure 5a is plotting a binned histogram, showcasing close to 90% input/output pair maximum matching for lengths up to 150 token. Figures 5b, 5c, 5d plot each input/output pair separately. The figures give a good visual cue on how more outputs eventually fail to reach maximum matching with increasing output length.

4.4.2.3 Future Work

While the results are somewhat encouraging, some major problems remain that would indicate some revised course of action:

- The results indicate lengths of 500-750 token, corresponding to short 'C' functions. The conclusion is that larger functions or code segments spanning multiple functions will not decompile correctly.
- Publicly available models are not pretrained towards decompilation. Using Starcoder, which is pretrained for 'in-the-middle' code completion on high-level source codes – with a large emphasis on Python – is not an ideal pretrained model for decompilation finetuning.
- While Transformers hold promise, ML that captures long range token dependencies, beyond of what is normally found in natural language processing (NLP), are necessary. An obvious choice is to exploit the inherent structure of codes, usually represented as graphs in compiler analysis, such as: Abstract Syntax Trees (AST), Control Flow Graphs (CFG), Dataflow Graphs (DFG) or Program Dependence Graphs (PDG). This would indicate the need for some Graph

ML or very large Transformer windows.

- Tokenization and embedding for decompilation will require dedicated engineering.
- Presumably, further training data homogenization and curation, specifically for storage locations, would greatly improve training results.
- There is no guarantee of correctness in the neural decompilation process. Using a decompiler like Ghidra with its large set of transformation rules should yield, with a sufficiently large training corpus, an ML that implicitly learned these rules. Yet, this is not sufficient. Automated correctness provers would need to flank the decompilation and optimization process as described in the next section 4.5

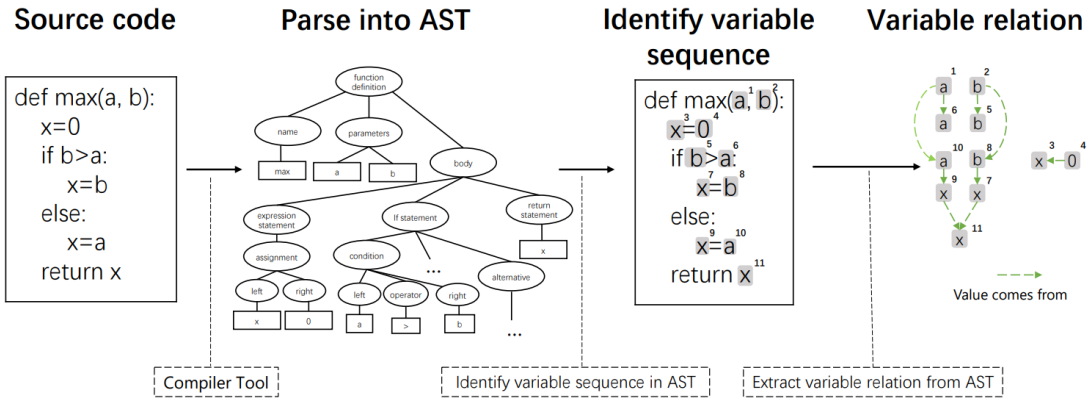


Figure 6: The procedure of extracting data flow given a source code. The graph in the rightmost is data flow that represents the relation of ”where-the-value-comes-from” between variables [13].

4.5 Proposal for a More Principled LLM Compiler/Decompiler

(T. C Fujimoto)

4.5.1 Capturing Code Structure

Ideally, we want an interpretable tool for compilation and/or reverse engineering. For example, if we want to decompile low-level code to a high-level language, we want the output to preserve the semantic intent of the low-level input. Current LLM compiler research seems to take the typical route of training large transformer neural networks as if code text is not significantly different from natural language text. Adding code structure to the finetuning of LLM code generators might improve performance. Some ideas include adding graph-based structure to the training data to help improve performance. As a simpler alternative to abstract syntax trees, GraphCodeBert [13] ignores syntactic structure and uses graphs to represent dependency relations between variables (see Figure 6). This method has been shown to improve performance in transformer code generators.

4.5.2 Augmenting LLM Compilers with Automated Theorem Provers

LLM proof assistants have become popular and can guarantee formal correctness [34, 16]. LLM compilers, inspired by proof assistants, would suggest several operations that make it easy to convert code in a high-level language to a lower-level language (and vice versa). Like a proof, this LLM would take an expression and translate it to a new expression using rules that guarantee the preservation of the expression’s meaning. LLMs have also been applied to automated theorem provers [31, 32], which is more appropriate for LLM compilers. Unlike mathematical proofs, there is less need for human intervention during the intermediate steps. Following this rule-based methodology ensures correctness of the code, which is more important for critical systems.

4.5.2.1 LLM Automated Theorem Provers

While much of LLM research focuses on LLM proof assistants [16, 34], automated theorem provers are more relevant to building an LLM compiler. Currently, the most capable models for

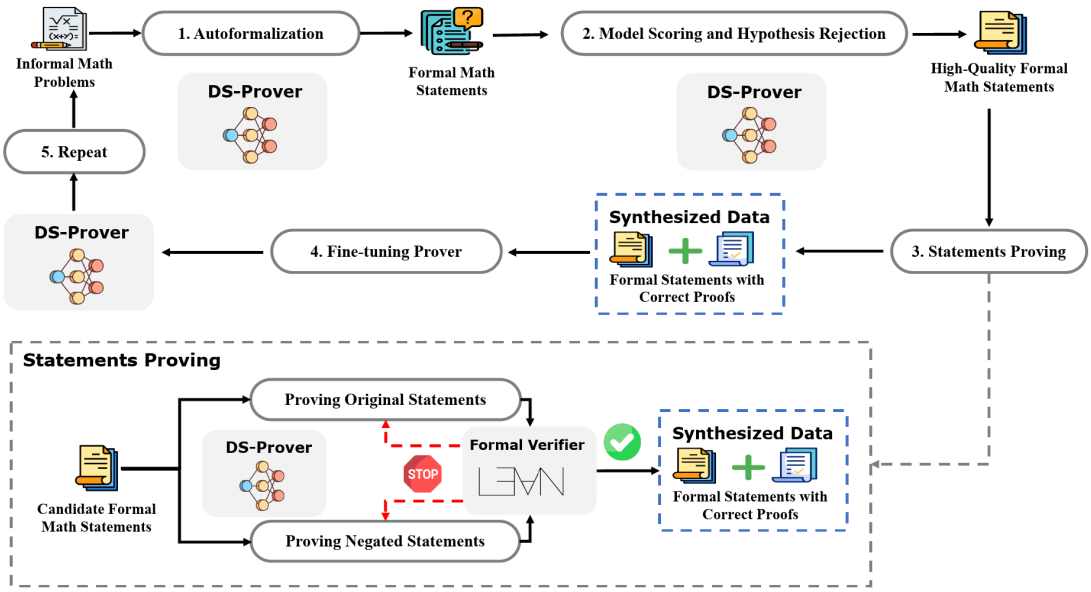


Figure 7: An overview of DeepSeek-Prover. The steps in this process are (1) autoformalization, (2) model scoring and hypothesis rejection (quality filtering), (3) statements proving, and (4) fine-tuning (refinement) [31].

automated theorem proving (ATP) are the DeepSeek-Prover models. The first model, called DeepSeek-Prover [31], has four key steps (see Figure 7):

1. Autoformalization. Using a large dataset of Lean 4 code back-translated into natural language problem descriptions with GPT-4, the authors finetune DeepSeek-Prover using a mathematical problem solving LLM [20] to translate mathematical descriptions in natural language to formal expressions.

2. Quality Filtering. First, the authors develop a scoring criteria for statement quality using chain-of-thought. Second, the authors discard false statements by using the DeepSeek-Prover to prove the statement leads to a ”False” conclusion in Lean 4.

3. Statement Proving. The model searches for proofs from the assumptions. The authors use concurrent proof searches between a statement and its negation to accelerate proof synthesis.

4. Iterative Refinement. Up to a point, finetuning on the model on newly generated data seems to improve the model’s theorem proving performance.

The same group also introduced an improved model, called DeepSeek-Prover 1.5 [32], which is augmented with reinforcement learning (RL) and Monte Carlo tree search (MCTS) (Figure 8) inspired by AlphaGo [21]. These additions, along with balancing multi-pass proof-step generation and single-pass whole-proof generation, led to DeepSeek-Prover 1.5 achieving a success rate of around 60% in the MiniF2F benchmark [36] (compared to 50% by the previous DeepSeek-Prover) .

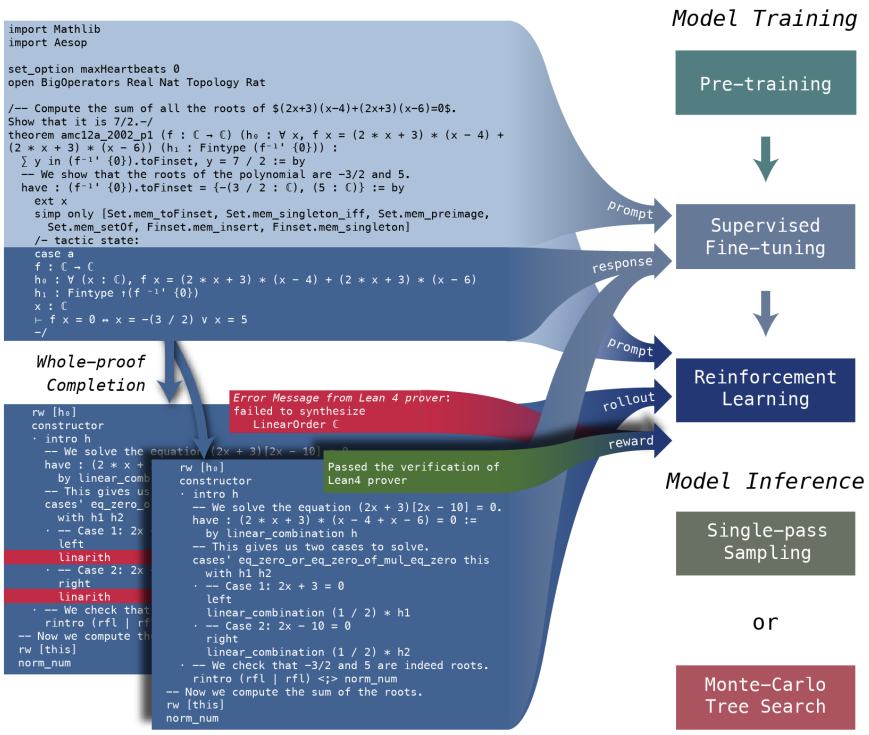


Figure 8: DeepSeek-Prover-V1.5 is trained through pre-training, supervised fine-tuning, and reinforcement learning [32].

4.5.3 LLM-Augmented Compiler Outline

The following is a rule-based LLM-based compiler application that automates the compilation steps. We split the process into the traditional steps (LLMs are not necessary at every step):

1. Lexical Analysis
2. Syntactic Analysis
3. Semantic Analysis (can benefit from a rule-based LLM)
4. Intermediate Representation (can benefit from a rule-based LLM)
5. Code Generation (can benefit from a rule-based LLM)
6. Optimization (can be fine-tuned using reinforcement learning on an objective measure of efficiency)

While LLM code generators have made impressive advancements, these models are still far from perfect. For example, Meta AI's LLM compiler is not dependable and efficient enough to be a practical replacement for current compilers [9]. The main assumption behind this outline is that the levels of abstraction indicated above compound the risk of inaccurate LLM code generation. In particular, we reduce the complexity of the task by having the LLM compiler only on a subset of steps. In particular, steps 1 and 2 can be satisfied by established algorithms. Hence, the LLM compiler only needs to be used for steps 3 through 6. Verified LLM code lifting and transpilation have been done before [7, 33], but reliable LLM compilation/decompilation with low-level programming languages is still an open problem [24]. The difficulty on the LLM is diminished if there are less levels of abstraction in the LLM compilation process.

4.5.4 Advantages of the Principled LLM Compiler

1. Except for syntactic and semantic correctness, the form of the low-level language expression does not matter. Mathematical proofs require deriving the exact form of the conclusion. Compilation to low-level languages only requires that the derived expression is well-formed in the intended language. Unlike in mathematical proofs, the exact form of the conclusion is not important.
2. We can finetune on openly available LLM compilers on Huggingface to accomplish steps 3 through 6 above. This will be more efficient than finetuning on other LLM code generators not trained on an adequate amount of assembly code.
3. Even if unsuccessful in translating an expression from one language to another, using the techniques from automated theorem proving will provide more interpretability by showing the steps the LLM is taking. This information can be useful in improving the LLM compiler because the typical LLM approach hides how the code is translated in the forward pass of the neural network.

4.5.5 Anticipated Difficulties

1. Translating the success from mathematical theorems to code compilation/decompilation. This would involve subtle engineering to apply the strategies from the automated theorem proving research to programming language compilation. This involves converting one expression to another equivalent expression. We also need to build the right framework that accepts code as input instead of mathematical expressions.

2. Deciding on the theorem proving language. Most work on LLM proof assistants and automated theorem provers are implemented in the Lean 4 interactive theorem proving language [18]. However, an open-source formally verified compiler, called CompCert [15], is implemented in a different language called Coq [25]. If CompCert is beneficial for a LLM compiler, it might be more advantageous to use Coq. We will have to weigh the pros and cons between the two languages.

3. For disassembly, we need to develop ways that ensure correctness without strict supervised fine-tuning. Unlike low-level languages, we care about high-level language readability. It is unclear how scalable reinforcement learning from human feedback (RLHF) [8] will be in this application.

4.5.6 Novelty

Current approaches to LLM compiler projects approach the problem through the typical “data + compute = LLM” paradigm. The current trend likely won’t lead to compilation/decompilation frameworks that are as reliable as established tools. LLMs with formal rules of code generation might subvert that trend. LLM-based applications that use a more principled approach to the compilation process is a more promising approach. LLM proof assistants and automated theorem proving will provide more secure LLM code generation. The novelty of using methods of formal verification is (1) ensuring correctness not guaranteed with typical LLM code generation, and (2) avoiding reliance on flawed scores that depend on arbitrary similarities between generated text that are not necessarily relevant to syntactic or semantic correctness [27, 10]. LLMs also have the ability to improve using RLHF [8]. Instead of human preferences, we use reinforcement learning to maximize desirable properties like FLOPs/sec or minimize energy usage.

4.5.7 Proposal for Time Frame and Team Formation

To accomplish this task, here is the proposed team formation:

- Generative AI researchers (1-2 people) to clearly define the problem, map out the software to be built, and propose the training/evaluation framework of the LLM compiler.

- Automated theorem proving researcher/experts (1-2 people) to extend past LLM theorem prover research to the LLM compiler. They must also coordinate with other team members to ensure the theorem proving portion of the software works with the ML portion. Some data scientists in NSD might fit this role.

- Compiler researchers/engineers (1-2 people) to direct the AI/ML researchers/engineers as domain experts, lead the data collection for training, and provide expertise on how to evaluate performance.

- Software engineers (2-3 people) who can either (i) train and evaluate LLMs, (ii) integrate theorem proving (in Lean 4 or Coq) with ML training/inference, or (iii) gather and clean programming language data for LLM finetuning.

During the first year, we can leverage the openly available LLM compilers and automated theorem provers. These pretrained models should make steps 3 through 5 easier. We will also decide on some preliminary benchmarks for compilation and disassembly. The hardest part would be to successfully integrate the theorem proving and ML training parts into a principled LLM compiler. During the second year or later, we can choose between multiple research paths:

1. Perfecting compilation. Here, we investigate how to formally integrate CompCert [15] with the LLM compiler. Since CompCert is implemented in Coq, this depends on the LLM compiler being able to interact with a Coq application.

2. Improve decompilation. We look at past methods of disassembly and reverse engineering [19, 28]. We combine this with past LLM decompile methods and models [30, 24]. For the decompiler, we also want desirable properties like interpretability and code readability.

3. Improve Efficiency. We would focus on step 6 of the principled LLM compiler and optimize efficiency metrics. Past work on RL and energy efficiency will guide this path [11].

4.6 Discussion

This principled approach shows more promise toward a successful LLM compiler than past efforts. By breaking the process into the typical steps of compilation, we can build an application that correctly and efficiently compiles code from one language to another. Other research organizations are pursuing LLM compilers. US National labs (LLNL, Argonne) and large tech companies (Google, Meta, Tencent) are investing in LLM low-level programming language compilation/decompilation [12, 9, 30].

One extra benefit of the automated theorem proving approach is discarding the distinction between compilation and decompilation. The tools of compilation and decompilation have always been split in two. Currently, there are the standard compilers like the GNU Compiler Collection, and the standard decompilation tools for reverse engineering like Ghidra. Both are large software projects that needed years to build and still require funding for future maintenance. The current LLM approach, proposed by groups like Meta AI [9], is an architecture similar to a backbone encoder-decoder transformer neural network. Here, the engineering complexity is transferred to the training instead of maintaining a codebase with large technical debt. We are still required to split the compiler and disassembler to encoder and decoder respectively. In the proposed principled LLM compiler, this distinction would no longer exist. Ideally, we would have only one transformer neural network, with a sufficient set of rules/tactics, that generates both the compilation/decompilation steps needed to reach the desired language representation. That is, one takes the rules for both compilation and decompilation, and combines them into a single list of rules for the LLM prover to use for transforming any expression. This would follow the trend from AlphaGo [21], to AlphaGo Zero [23], and finally AlphaZero [22], where two neural networks trained on human data were initially used, but the framework was improved to elegantly rely on only one neural network with no human data.

References

[1] Ghidra.

[2] Meta AI: Integrity.

[3] The Stack 2.

[4] Threats, Vulnerabilities, and Controls of Machine Learning Based Systems: A Survey and Taxonomy.

[5] A Systematic Review on Model Watermarking for Neural Networks. *Frontiers Big Data*, 4, November 2021.

[6] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.*, 38(4), July 2019. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[7] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit A Seshia, and Alvin Cheung. Verified code transpilation with llms. *arXiv preprint arXiv:2406.03003*, 2024.

[8] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[9] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*, 2024.

[10] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741, 2023.

[11] Qiming Fu, Zhicong Han, Jianping Chen, You Lu, Hongjie Wu, and Yunzhe Wang. Applications of reinforcement learning for building energy efficiency control: A review. *Journal of Building Engineering*, 50:104165, 2022.

[12] Aiden Grossman, Ludger Paehler, Konstantinos Parasyris, Tal Ben-Nun, Jacob Hegna, William Moses, Jose M Monsalve Diaz, Mircea Trofin, and Johannes Doerfert. Compile: A large ir dataset from production sources. *arXiv preprint arXiv:2309.15432*, 2023.

[13] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2021.

[14] Rasmus Munk Larsen and Tatiana Shpeisman. TensorFlow Graph Optimizations, 2019.

[15] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[16] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving. *arXiv preprint arXiv:2404.09939*, 2024.

[17] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder 2 and The Stack v2: The Next Generation, 2024. _eprint: 2402.19173.

[18] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer, 2021.

[19] Miroslav Popovic, Vladimir Kovacevic, and Ivan Velikic. A formal software verification concept based on automated theorem proving and reverse engineering. In *Proceedings Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 59–66. IEEE, 2002.

[20] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Yu Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[21] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[22] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[23] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[24] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286*, 2024.

[25] The Coq Development Team. The Coq reference manual – release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman, 2024.

[26] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, 2023. _eprint: 2302.13971.

[27] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Does bleu score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176. IEEE, 2019.

[28] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound c code decompilation for a subset of x86-64 binaries. In *Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings 18*, pages 247–264. Springer, 2020.

[29] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[30] Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Refining decompiled c code with large language models. *arXiv preprint arXiv:2310.06530*, 2023.

[31] Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024.

[32] Huajian Xin, ZZ Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, et al. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*, 2024.

[33] Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. Vert: Verified equivalent rust transpilation with few-shot learning. *arXiv preprint arXiv:2404.18852*, 2024.

[34] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[35] Field G. Van Zee and Robert A. van de Geijn. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, June 2015.

[36] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

Pacific Northwest National Laboratory

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99354
1-888-375-PNNL (7665)

www.pnnl.gov