Vision Paper: Proof-Carrying Code Completions

Parnian Kamran
pkamran@ucdavis.edu
University of California, Davis
USA

Premkumar Devanbu
ptdevanbu@ucdavis.edu
University of California, Davis
USA

Caleb Stanford
cdstanford@ucdavis.edu
University of California, Davis
USA

Abstract

Code completions produced by today's large language models (LLMs) offer no formal guarantees. We propose *proof-carrying code completions* (PC^3). In this paradigm, a high-resourced entity (the LLM provided by the server) must provide a code completion together with a proof of a chosen safety property which can be independently checked by a low-resourced entity (the user). In order to provide safety proofs *without* requiring the user to write specifications in formal logic, we statically generate preconditions for all dangerous function calls (i.e., functions that may violate the safety property) which must be proved by the LLM.

To demonstrate the main ideas, we provide a prototype implementation in the program verification language Dafny, and a case study focusing on file system vulnerabilities. Unlike Python code generated by GPT-4, Dafny code generated by PC^3 provably avoids a common weakness related to path traversal (CWE-35), using a single generation attempt ($k = 1$) and a modest number of tokens (3, 350). Our tool is available as an open source repository at https://github.com/DavisPL/PCCC.

CCS Concepts

• **Software and its engineering → Formal software verification**; **Software development techniques**; **Formal methods**.

Keywords

Proof-carrying code, Formal verification, Large language models, Program synthesis, Dafny

ACM Reference Format:
Parnian Kamran, Premkumar Devanbu, and Caleb Stanford. 2024. Vision Paper: Proof-Carrying Code Completions. In *39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3691621.3694932

1 Introduction

A growing majority of software developers (currently, 63% [35]) integrate software tools based on large-language models (LLMs) into the development workflow. Yet, the code produced by LLMs offers no formal guarantee of its correctness or safety, which has raised concerns about its deployment, especially in safety- and security-critical settings. According to existing studies, not only do LLMs sometimes suggest defective code [53, 59] (for example, see Figure 1), but also, users sometimes accept these suggestions [57]. Users who are unsure of how to complete a task may be more likely to accept large blocks of code without careful vetting [9]. To make matters worse, LLMs are not run locally, but over the cloud by a service provider – which is another point of failure, as the provider must be trusted to run a well-trained and properly-aligned LLM faithfully and not to contaminate its output. These considerations raise the basic question: how can we ensure that code produced by LLMs is safe to run?

```
import os
# Function to retrieve public RSA key from the filesystem
def load_rsa_key(home_dir="~", key="id_rsa"):
    key_path = os.path.join(os.path.expanduser(home_dir), ".ssh",
        ↪ key)
    with open(key_path, "r") as f:
        return f.read()
```

Figure 1: Code completion by GitHub CoPilot (Sept. 2023) given the first 3 lines as prompt. The function erroneously returns the *private* key rather than the public key as desired.

The problem can also be posed at a more abstract level. Suppose we view the LLM provider (*e.g.*, OpenAI or Google) as a *computationally well-resourced entity* H, which runs a generative LLM that produces some code C_H, and a user of the code completion tool as a *computationally low-resourced* entity L who wants to use C_H. Assume in addition that L wants to ensure that *all* the code it actually executes satisfies some safety property Π_s (this might be relating to data privacy, filesystem access, service protection, *etc.*). The problem becomes: how can L gain confidence that C_H is safe to run or deploy, i.e., satisfies Π_s?

To solve this problem, we propose *proof-carrying code completions* (PC^3). Inspired by the classic idea of *proof carrying code* [51] and its extensions [5–7, 10, 28], as well as related work in proof generation and synthesis (including but not limited to [16, 22, 25, 50, 54]), in the PC^3 framework, each generated completion comes packaged with a *proof of safety* that can be checked, even by a computationally bounded end user. PC^3 could solve both trust issues raised above: first, if LLMs can be used successfully to equip code with proofs, the code need not be trusted without being checked; and second, if LLM service providers modify the code, they would also have to modify the proof of safety; otherwise such attempts would be detected at the validation step.

We use Dafny [44, 45], an industrial-standard Hoare-logic based program verifier, as a concrete testbed for these ideas. We target users who are not experts in Dafny, as such users are more likely to rely on (and benefit from) safe code completions. We identify (at least) the following three challenges for producing safe code completions in this context:

- First, *the intended user lacks expertise in formal proofs or program verification logics*, so proofs should ideally be generated behind the scenes with minimal user input. In particular, the user may not wish to provide task-specific preconditions, postconditions, theorems, or lemmas for the LLM to prove (*c.f.* recent work using LLMs to automate proofs, e.g., [16, 25, 49, 50]).
- Second, much like users, *LLMs are themselves unfamiliar with the idioms used in many proof languages,* and may struggle with generating logical formulas and proof primitives. In particular, Dafny syntax is out-of-distribution for large text corpora used in LLM training.
- Third, *code completions are partial (incomplete) programs*, so each code completion should be verifiable in isolation – rather than as a whole program fully annotated with proofs.

To address the first challenge, we follow the original proof-carrying code work [51] by focusing on global *safety properties*, rather than task-specific formal specifications. The user selects the safety property only once, and can reuse it for any number of completions. Alternatively, the safety property could be set by policy at the team or project level.

To address the second challenge, we build on recent work on synthesizing proofs in Dafny [49] to propose a prompting and agent framework for generating Dafny programs with safety proofs. Our intuition is that generating safety proofs will be easier than generating arbitrary correctness proofs, as the syntax is simpler, and this syntax is often similar or identical across different code examples for the same safety property.

Finally, to address the third challenge, our main idea is to abstract all dangerous behavior in a dedicated *effectful interface* (e.g., an API for filesystem access or unchecked array bounds access). We assume that any program *side effects* (i.e, behavior of a program beyond its input and output) must occur through this interface; this assumption is relatively easy to enforce in Dafny. Pre- and post-conditions to the interface are then statically generated based on the user's chosen safety property. This means that each call to the effectful interface comes with its own verification query (i.e., precondition to be verified), and all of these calls can be checked independently by the Dafny verifier. In particular, some calls can be verified even if others fail; and these can be checked even if the function or method body is incomplete.

Our primary contribution is a combination of these ideas in our architecture proposal for *PC*^{3}, laid out in Section 2. The *PC*^{3} components are: an LLM proof generation loop for generating verified Dafny code; a feedback loop for taking the Dafny compiler output and feeding it back to the LLM; an effectful interface for side effects which interacts with the LLM-produced code; and a verification condition generator which takes as input the desired safety properties and inserts the necessary pre- and post-conditions on the effectful interface.

We have implemented this architecture in a prototype tool focusing on code completions that interact with the file system. In Section 3, we provide a preliminary evaluation of our prototype through a case study. We conclude that it is feasible to use the *PC*^{3} architecture to generate filesystem code in Dafny which provably avoids a path traversal weakness (Mitre CWE-35 [19]). *PC*^{3} is available as an open source repository on GitHub (MIT License) and we welcome issues and feedback.[1]

2 Design and Architecture

In this section, we first describe each component of the *PC*^{3} design and architecture. We present these components as a general framework that could be implemented in any programming language which supports verification. In the final subsection (Section 2.6), we describe the status of each component in our current implementation in Dafny.

2.1 Overall Workflow

The *PC*^{3} workflow, as illustrated in Figure 2, begins with a user submitting a code generation request, which we refer to as a *task*. For example, the task could be: "implement a function which takes the file path and the content as the parameters, which will then be used to open the file and write the content into the given file." Before forwarding the user's query to an LLM, *PC*^{3} instruments the request with additional information including similar few-shot examples (Section 2.2).

Together with the user's task, *PC*^{3} presents the user with a list of predefined safety properties. For instance, these could include: no file path contains the pattern `..`; for each file, all bytes written to the file are ASCII characters; or for each file, the user must have write permissions to that file. Safety properties can be represented in a textual format or in natural language to be understandable for users, but in order to enable verification they must be converted to formal verification conditions (VCs). This is done by the *verification condition generator (VCGen)*, which inserts the VCs as preconditions on the *effectful interface* (Section 2.3). The key contract is that no matter how generated code interacts with the effectful interface, if it satisfies the VCs on code entry, then the safety properties are upheld (Section 2.4).

After querying the LLM and extracting generated code, if verification fails, the prompt is refined with compiler errors and extra diagnostic information for subsequent LLM queries. This iterative process repeats for a threshold number of steps T (Section 2.5).

2.2 Prompting

As shown in Figure 2 (top left), the prompt has three components:

(1) A textual description of the task, provided by the user.
(2) A list of available methods from the effectful interface, annotated with all pre- and post-conditions (see Section 2.3).
(3) A limited number of examples for few-shot learning (or retrieval-augmented generation (RAG) [46]), taken from a database of known examples.

Of the three components, only the first is provided by the user, while the second and third are constructed by the *PC*^{3} architecture (using the safety property as input) and added to form the prompt. A prompt template along these lines is shown in Figure 3.

Our template uses Chain-of-Thought (CoT) prompting, following Misu et al. [49], to guide the LLM through the task and verification conditions using step-by-step instructions [65]. Our CoT prompt

[1]https://github.com/DavisPL/PCCC

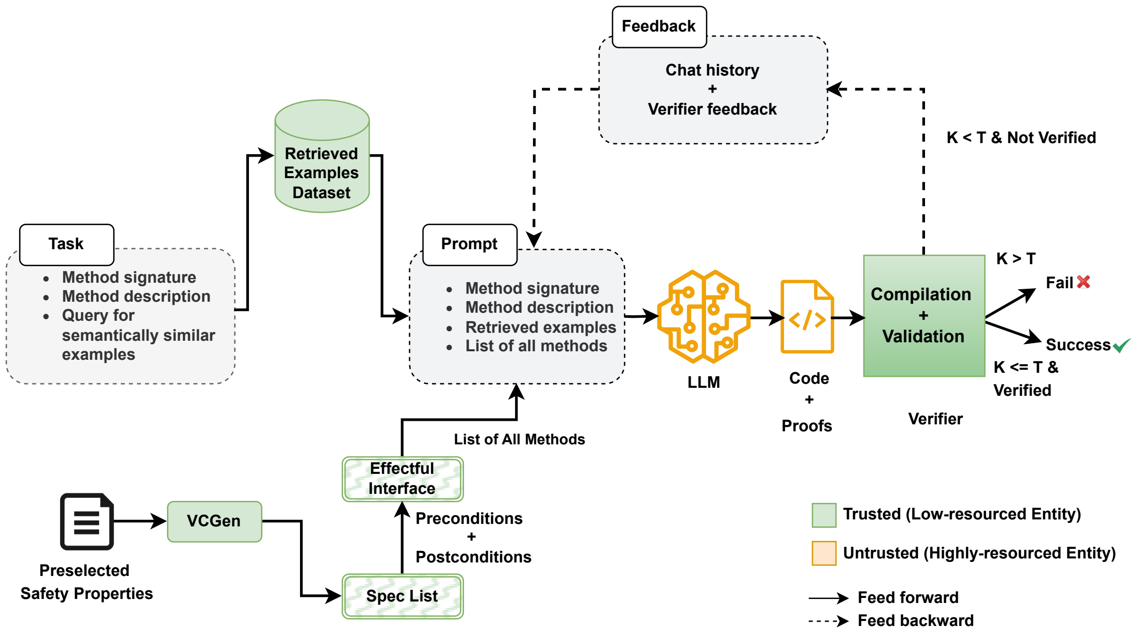


Figure 2: Overview of PC^3 workflow. Trusted computation is shown in green, untrusted computation is shown in yellow, and user input is on the left. The solid arrows indicate feed forward and the dashed arrows show feed backward flows.

contains five few-shot examples, together with the expected response in each step. To choose the few-shot examples, one could also use Retrieval Augmented Generation (RAG) [46]. RAG performs a semantic similarity search between the user query, the method signature, the specifications corresponding to the desired safety properties, and the external dataset to retrieve relevant examples.

The prompt contains a SYSTEM message which can be used to tell the LLM that it should refer to the list of APIs in the effectful interface (with VCs added) when external interaction is required. It is helpful for the SYSTEM message to explicitly emphasize that the generated code must satisfy the desired verification conditions of each chosen API.

2.3 Effectful Interface

The *effectful interface* comprises a list of methods that can have unsafe side effects – for example, opening a file or writing to a file.

Our assumption is that all interactions with the operating system or other potentially unsafe code must pass through this interface. This assumption is possible to enforce in many programming languages (for instance, in Dafny and in any language where side effects such as system calls can only be accessed through the standard library). In Section 2.4 we give an example of how the methods in the effectful interface avoid unsafe behavior, after VCs are added.

2.4 Safety Properties and VC Generator

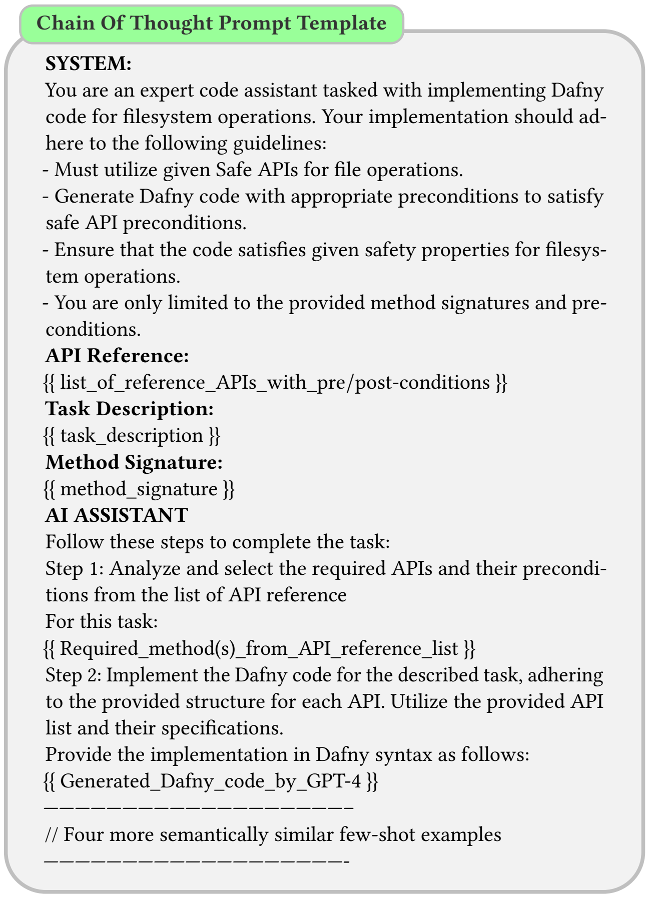
As Lamport [40] described, a safety property specifies that no "bad thing" occurs during any execution of a program. The PC^3 architecture asks the user to choose as input one or more safety properties from a predefined list. For example, one available safety property could be that all files written to should have the extension `.txt` or `.md`. For readability, each safety property may be associated with a natural language description.

The selected safety property is then provided as input to the VC Generator (VCGen). At this stage, the property needs to be converted to formal preconditions that are understandable to the verification tool. These preconditions are the inserted directly into the effectful interface (for example, as preconditions to the `read` or `write` methods).

Note that it is up to the VC Generator to do this correctly, i.e. to actually ensure that the safety properties must hold, given the preconditions. More formally: if the safety property is Π_s, it should be true that for any function f in the effectful interface, and for any input x to that function, *if* x satisfies the precondition generated by VCGen, *then* for any execution of f on input x, and for any point during that execution, the program state at that point satisfies Π_s.

2.5 Validation

Lastly, the validation process uses a program verification tool (the *verifier*, e.g., Dafny, Coq, Idris, Agda, or Lean) to validate that the code compiles and that verification passes. Any syntax errors generated by the compiler (for example, an unidentified function or method name) are fed back to the LLM for correction. The feedback can also be amended to convey additional diagnostic information about, for example, wrong effectful interface API usage, method signature hints, or linking the error message with specific lines in the code.



Chain Of Thought Prompt Template

SYSTEM:
You are an expert code assistant tasked with implementing Dafny code for filesystem operations. Your implementation should adhere to the following guidelines:
- Must utilize given Safe APIs for file operations.
- Generate Dafny code with appropriate preconditions to satisfy safe API preconditions.
- Ensure that the code satisfies given safety properties for filesystem operations.
- You are only limited to the provided method signatures and preconditions.

API Reference:
{{ list_of_reference_APIs_with_pre/post-conditions }}
Task Description:
{{ task_description }}
Method Signature:
{{ method_signature }}
AI ASSISTANT
Follow these steps to complete the task:
Step 1: Analyze and select the required APIs and their preconditions from the list of API reference
For this task:
{{ Required_method(s)_from_API_reference_list }}
Step 2: Implement the Dafny code for the described task, adhering to the provided structure for each API. Utilize the provided API list and their specifications.
Provide the implementation in Dafny syntax as follows:
{{ Generated_Dafny_code_by_GPT-4 }}
——————————————————
// Four more semantically similar few-shot examples
——————————————————-

Figure 3: Chain of thought prompt with few-shot examples and API reference list from the effectful interface.

Similar to other work in this space, our proposed architecture runs the verification loop up to a fixed number of iterations (the *verify@K* metric [49]); we use K to denote the current iteration number, and T to denote the threshold (maximum) number of iterations.

2.6 Current implementation

In this section we explain the current implementation of each component in the PC^3 framework in Dafny. Currently, our implementation focuses on filesystem operations. Thus, our prompt template contains a task description, method signature, and 5 few-shot Dafny examples along with a list of APIs for primary filesystem operations.

For the effectful interface, we adapted and expanded the list of filesystem-related functions available in the Dafny standard library [41], as well as in an existing open-source file IO interface [66]. The specific methods we allow are: `Open`, `Join`, `Read`, `Write`, `Copy`, `Flush`, and `Seek` that may have unsafe side effects during execution.

The safety properties we consider (again focusing on filesystem operations [62]) are inspired by CWEs reported by Mitre [18] We support safety properties disallowing path traversal, sensitive files access, home directory access, invalid characters in filename and pathname, reserved names in a path, and invalid file extensions.

Currently, VC generation is a manual process based on mapping the desired safety properties to corresponding preconditions.

Lastly, for the verifier, the current implementation uses $T = 5$ as the maximum number of attempts before giving up on a task. We use the Dafny compiler to generate error messages, and we link each error to the line of code that leads to that error.

3 Case study

To study the feasibility of the PC^3 framework, we conducted a preliminary case study. This section presents a comparative analysis of code generated by GPT-4 under two scenarios: first, without any safety properties as a baseline in Python, and second, using the PC^3 framework in Dafny with preselected safety properties.

Our case study aimed to evaluate the PC^3 framework for a simple filesystem operation: creating a full path in the user's system by appending a given file to a predetermined path. The main objectives were to answer the following research questions:

RQ1 Is the generated code safe, i.e., does it satisfy the selected safety properties?
RQ2 How efficient are the chain of thought prompt and few-shot learning examples in obtaining code that verifies?
RQ3 Is the generated code correct, i.e., does it implement the operation in question?

Experimental setup. Our experiment used the GPT-4 API (released in March 14, 2023), a model with 175 billion parameters and and an 8192 token context length. The experiments were run on August 10, 2024 using the GPT-4 API, which at that time was referencing the gpt-4-0613 model. The model temperature is 0.75. We used LangChain `0.2.7` and Dafny `4.6.0`, executing the PC^3 framework on a local M1 Pro machine with 10 cores, 16 GB memory, running macOS `14.5`.

Task description. Our case study was inspired by *Mitre CWE-35* [19], which relates to path traversal vulnerabilities [17]. Path traversal occurs when a program uses external input to create a path that should be within a restricted directory, but can resolve to a location outside of that directory using doubled or tripled dot slash patterns. We designed a simple scenario shown in Figure 4. This prompt asks the model to generate code for path construction by appending an external input file to a predetermined path on the user's system (`/home/user/documents/`).

Baseline. Figure 5 shows GPT-4's response in Python: a code snippet using `os.path.join()`. The resulting code, while functionally correct, lacks insurance against dangerous patterns that could lead to path traversal attacks (such as `../`, `..`, or any representation of these patterns). Consider a scenario that the function receives an external file with doubled dot patterns, e.g., `../../etc/passwd`. This input bypasses the restricted directory (`/home/user/documents/`), potentially exposing a sensitive file outside that directory (e.g., `../../etc/passwd`).

Results. We applied the PC^3 framework in Dafny to the same scenario, adding a method signature to the prompt (Section 3). For this task, we selected the following safety properties: prevention of path traversal, no empty strings, and no invalid characters. We implement the VC generation step manually by annotating `Join` in the effectful interface with preconditions corresponding to the mentioned safety properties (Figure 4). The translation in this case is a straightforward one-to-one mapping of safety properties to preconditions, and thus would be possible to automate; but a more general treatment of VCGen is an important direction for future work (Section 5).

Figure 6 shows GPT-4's response in its first attempt ($K = 1$). As shown, GPT-4 correctly generates preconditions which avoid the path traversal vulnerability, i.e., *CWE-35*. The token usage for this task is roughly 3,350 with an average cost of \$0.109 per query (for the COT prompt template shown in Figure 3). We manually inspect the output to ensure it is correct. We also ensure that the preconditions generated are not inconsistent by checking whether the method body is unreachable (this can be done in Dafny by adding `assert false` to the method body).

Conclusions: **(RQ1)** Yes: the code generated does satisfy the required safety properties for the given task in Figure 4. **(RQ2)** Our PC^3 implementation is able to generate code for this task using a modest number of tokens and a single verification attempt. **(RQ3)** Yes: by manual inspection, we observe that the generated output is correct and the generated preconditions are consistent.

4 Related Work

Proof-carrying code (PCC). PCC was introduced by Necula in 1997 [51]. PCC was originally used as a model for *mobile* code security [43], where it would be passed between mobile agents (today's edge devices). PCC was originally used to provide assurance for basic safety properties, in particular type safety, memory safety, and data access restrictions [52]. Appel and Felty introduced *foundational* PCC, which generalizes PCC to allow custom types and custom semantics [8], and these foundations inspired later efforts to build large-scale program verification frameworks based on PCC [10, 28]. PCC is also closely related to packet filtering mechanisms used by operating systems (notably, BPF [48] and its successor eBPF in the Linux kernel).

Appel and Felten extended PCC with a sequence of logged API calls to enforce certain properties [6, 7], encoded using security automata [58, 64], which can be seen as analogous to the effectful interface used in our work. Vanegue showed that both PCC and foundational PCC can be vulnerable to *weird machines*, where despite the program being safe to run, it has undesirable side effects that are invisible to the specification [63].

Language models for proofs. Large language models have recently been shown to be moderately effective at generating formal (machine-checked) proofs [16, 25, 30, 54]. These proofs can be generated in proof assistants like Coq, Isabelle, or HOL, and may have the potential to replace earlier dedicated tools such as CoqHammer [21] and other machine-learning based techniques [24, 56, 67]. There is also some work in generating formally verified code in other languages, such as C [22] and F* [16]. Despite promising successes, language models still exhibit failure cases when asked to generate proofs (see [68]).

Model Input Without PC^3 Framework

TASK: Your task is to implement a function which takes the filename as a parameter, which will then be used to create a full file path in a predetermined directory (e.g., `/home/user/documents/`).

Given Task in PC^3 Framework

Task Description:
Your task is to implement a function which takes the filename as a parameter, which will then be used to create a full file path in a predetermined directory (e.g., `/home/user/documents/`).
Method Signature:

```
method CreateFullPath(filename: seq<char>)
```

Filesystem API + pre/post-conditions

```
method Join(p: path, f: file) returns(result:
     ↪ path)
 requires non_empty_string(f)
 requires non_empty_string(p)
 requires !has_dangerous_pattern(f)
 requires !has_dangerous_pattern(p)
 requires has_absoloute_path(f)
 requires is_valid_file_name(f)
 requires is_valid_path_name(p)
 requires has_valid_file_length(f)
 requires has_valid_path_length(p)
 requires has_valid_path_length(p+f)
 requires append_file_to_path(p, f) == p + f
```

Figure 4: Top: prompt for the case study given to GPT-4 without PC^3. Middle: prompt for the case study with PC^3. Bottom: effectful interface API with pre- and post-conditions that is made available to the PC^3 version.

```python
def create_full_path(filename):
    base_directory = "/home/user/documents/"
    return os.path.join(base_directory, filename)

# Example usage:
filename = "example.txt"
full_path = create_full_path(filename)
print("Full path to the file:", full_path)
```

Figure 5: Response generated by GPT-4 to the prompt in Figure 4 (top) (without using PC^3).

The most relevant related works are Misu et. al. [49] and Mungier et. al. [50]. The first of these [49] uses LLMs to synthesize Dafny code together with Dafny proofs. Compared to our work, they consider tasks given by a description together with pre- and post-conditions, and unit tests, and they do not consider safety properties. Their work on prompt engineering (such as chain of thought and

```
method CreateFullPath(filename: seq<char>) returns (↩
    ↪ fullPath: seq<char>)
  requires non_empty_string(filename)
  requires is_valid_file_name(filename)
  requires !has_dangerous_pattern(filename)
  requires has_valid_file_length(filename)
{
  var directory: seq<char> := "/home/user/documents/";
  fullPath := Join(directory, filename);
}
```

Figure 6: Response generated by GPT-4 to the prompt in Figure 3 (using PC^3) with the task in Figure 4 (middle and bottom).

retrieval-augmented generation) to synthesize Dafny code was very helpful to our own work and implementation. The second [50] uses LLMs to synthesize helper assertions, which are an important part of proofs in Dafny and often difficult to get right, especially for new users.

Language models for software. Predating LLMs, several lines of work aim to leverage natural language and natural language modeling for software. This *naturalness* of software [31] can be exploited for various tasks [4], such as code completion [13, 29, 34, 55], variable naming [12, 32, 33, 42], type annotation [2, 37, 38], code summarization [1, 3, 14, 60, 61], *etc.* Some recent work has shown that large language models can repeat human-created errors. For example, LLMs (despite being trained on code *after* all the bugs in our dataset of defects [39] were fixed) tend to reproduce buggy code more often than the corresponding fix code [36]. There have also been reports of using LLMs to create malicious code [27]. Other researchers report that developers using programming assistants sometimes don't review the code generated by these assistants very carefully [9, 47]. We do not attempt to provide a full survey of work on AI safety for LLMs, but for some examples of recent work, see [26, 57].

Tool support in Dafny. Finally, there is recent work on improving the developer experience in Dafny, using traditional automation and tool support rather than LLMs. For example, this work includes counterexample generation [15], automated testing [23], and better support for type soundness proofs [20].

5 Discussion and Outlook

In an era where LLMs are used to freely generate code, both by users and as a part of larger systems, formal guarantees will become increasingly more important [11]. In this paper, we considered the problem of generating *code completions* together with proofs of a safety property, and instantiated our solution, PC^3, in the context of the Dafny program verification language. Our initial work and case study focused on filesystem vulnerabilities demonstrates that the idea is promising, and opens several avenues for future work.

In our design, side effects (any operations which may violate the safety property) are abstracted behind a safe effectful interface, with automatically generated pre- and post-conditions. Providing additional Dafny interfaces (e.g., for network access, database access, raw memory reads and writes, and other system calls) is a traditional systems problem that we believe is especially relevant for LLM safety. A full implementation of the (currently manual) VCGen component is another important direction; as part of this effort, it would be useful to incorporate configuration languages for describing safety properties in ways that are accessible to end users, for example based on access control policies. Finally, while using LLMs to synthesize Dafny code is feasible using existing prompting techniques, in our experience it is still far more difficult than generating code in mainstream languages. Developing prompting techniques and fine-tuning models, or developing tools which implement PC^3 in mainstream languages (e.g., Python, C, or Rust) could help close the gap and reduce the effort required in prompt engineering.

Acknowledgments

This research was supported in part by the US National Science Foundation under award CCF #2403762. The authors would like to thank Earl Barr and others for early discussions of the idea and the anonymous reviewers for their feedback.

References

[1] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[2] Toufique Ahmed, Premkumar Devanbu, and Vincent J Hellendoorn. 2021. Learning lenient parsing & typing via indirect supervision. *Empirical Software Engineering* 26 (2021), 1–31.

[3] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. 2023. Extending Source Code Pre-Trained Language Models to Summarise Decompiled Binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 260–271.

[4] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[5] Andrew W Appel. 2001. Foundational proof-carrying code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 247–256.

[6] Andrew W Appel and Edward W Felten. 2001. Models for security policies in proof-carrying code. *Technical Report TR-636-01* (2001).

[7] Andrew W Appel, Edward W Felten, and Zhong Shao. 2005. Scaling proof-carrying code to production compilers and security policies (Final technical report). *Princeton University and Yale University, January* (2005).

[8] Andrew W Appel and Amy P Felty. 2000. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 243–253.

[9] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[10] Gilles Barthe, Pierre Crégut, Benjamin Grégoire, Thomas Jensen, and David Pichardie. 2007. The MOBIUS Proof Carrying Code Infrastructure: (An Overview). In *International Symposium on Formal Methods for Components and Objects*. Springer, 1–24.

[11] Emery Berger and Ben Zorn. 2024. AI Software Should be More Like Plain Old Software — blog.sigplan.org. https://blog.sigplan.org/2024/04/23/ai-software-should-be-more-like-plain-old-software/. [Accessed 10-08-2024].

[12] Dave Binkley, Matthew Hearn, and Dawn Lawrie. 2011. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 203–206.

[13] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*. 213–222.

[14] Raymond PL Buse and Westley R Weimer. 2010. Automatically documenting program changes. In *Proceedings of the 25th IEEE/ACM international conference on automated software engineering*. 33–42.

[15] Aleksandar Chakarov, Aleksandr Fedchin, Zvonimir Rakamarić, and Neha Rungta. 2022. Better counterexamples for Dafny. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 404–411.

[16] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. 2024. Towards Neural Synthesis for

SMT-Assisted Proof-Oriented Programming. *arXiv preprint arXiv:2405.01787* (2024).

[17] The OWASP Community. July 2024. Path Traversal | OWASP Foundation — owasp.org. https://owasp.org/www-community/attacks/Path_Traversal. [Accessed 10-08-2024].

[18] The MITRE Corporation. 2020. CWE - CWE-1219: File Handling Issues (4.15) — cwe.mitre.org. https://cwe.mitre.org/data/definitions/1219.html. [Accessed 10-08-2024].

[19] The MITRE Corporation. July 16, 2024. CWE - CWE-35: Path Traversal: &apos;.../...//&apos; (4.15) — cwe.mitre.org. https://cwe.mitre.org/data/definitions/35.html. [Accessed 10-08-2024].

[20] Joseph W Cutler, Emina Torlak, and Michael Hicks. 2024. Improving the Stability of Type Soundness Proofs in Dafny. In *Proceedings of the First Workshop on Dafny*.

[21] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning* 61 (2018), 423–453.

[22] Sarah Fakhoury, Markus Kuppe, Shuvendu K Lahiri, Tahina Ramananandro, and Nikhil Swamy. 2024. 3DGen: AI-Assisted Generation of Provably Correct Binary Format Parsers. *arXiv preprint arXiv:2404.10362* (2024).

[23] Aleksandr Fedchin, Tyler Dean, Jeffrey S Foster, Eric Mercer, Zvonimir Rakamarić, Giles Reger, Neha Rungta, Robin Salkeld, Lucas Wagner, and Cassidy Waldrip. 2023. A toolkit for automated testing of Dafny. In *NASA Formal Methods Symposium*. Springer, 397–413.

[24] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: semantics-aware proof synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31.

[25] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. arXiv:2303.04910 [cs.LG]

[26] Ryan Greenblatt, Buck Shlegeris, Kshitij Sachan, and Fabien Roger. 2023. AI control: Improving safety despite intentional subversion. *arXiv preprint arXiv:2312.06942* (2023).

[27] Maanak Gupta, CharanKumar Akiri, Kshitiz Aryal, Eli Parker, and Lopamudra Praharaj. 2023. From ChatGPT to ThreatGPT: Impact of Generative AI in Cybersecurity and Privacy. *IEEE Access* (2023).

[28] Nadeem A Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. 2003. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning* 31 (2003), 191–229.

[29] Sangmok Han, David R Wallace, and Robert C Miller. 2009. Code completion from abbreviated input. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 332–343.

[30] Vincent J Hellendoorn, Premkumar T Devanbu, and Mohammad Amin Alipour. 2018. On the naturalness of proofs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 724–728.

[31] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[32] Einar W Høst and Bjarte M Østvold. 2008. The Java programmer's phrase book. In *International Conference on Software Language Engineering*. Springer, 322–341.

[33] Einar W Høst and Bjarte M Østvold. 2009. Debugging method names. In *European Conference on Object-Oriented Programming*. Springer, 294–317.

[34] Daqing Hou and David M Pletcher. 2011. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 233–242.

[35] Stack Exchange Inc. 2024. Stack Overflow Developer Survey 2024. https://survey.stackoverflow.co/2024/ai Accessed July 2024.

[36] Kevin Jesse, Toufique Ahmed, Premkumar T Devanbu, and Emily Morgan. 2023. Large Language Models and Simple, Stupid Bugs. *arXiv preprint arXiv:2303.11455* (2023).

[37] Kevin Jesse, Premkumar T Devanbu, and Toufique Ahmed. 2021. Learning type annotation: is big data enough?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1483–1486.

[38] Kevin Jesse, Premkumar T Devanbu, and Anand Sawant. 2022. Learning to predict user-defined types. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1508–1522.

[39] Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 573–577.

[40] Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* SE-3, 2 (1977), 125–143. https://doi.org/10.1109/TSE.1977.229904

[41] Dafny language team. 2023. libraries/src/FileIO at master · dafny-lang/libraries — github.com. https://github.com/dafny-lang/libraries/tree/master/src/FileIO. [Accessed 10-08-2024].

[42] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *14th IEEE international conference on program comprehension (ICPC'06)*. IEEE, 3–12.

[43] Peter Lee and George Necula. 1997. Research on proof-carrying code for mobile-code security. In *DARPA workshop on foundations for secure mobile code*. Citeseer, 26–28.

[44] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

[45] K Rustan M Leino. 2023. *Program Proofs*. MIT Press.

[46] Patrick Lewis, Ethan Perez, Aleksandara Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *ArXiv* abs/2005.11401 (2020). https://api.semanticscholar.org/CorpusID:218869575

[47] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. Understanding the Usability of AI Programming Assistants. *arXiv preprint arXiv:2303.17125* (2023).

[48] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture.. In *USENIX winter*, Vol. 46. 259–270.

[49] Md Rakib Hossain Misu, Cristina V Lopes, Iris Ma, and James Noble. 2024. Towards AI-assisted synthesis of verified dafny methods. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 812–835.

[50] Eric Mugnier, Emmanuel Anaya Gonzalez, Ranjit Jhala, Nadia Polikarpova, and Yuanyuan Zhou. 2024. Laurel: Generating Dafny Assertions Using Large Language Models. *arXiv preprint arXiv:2405.16792* (2024).

[51] George C Necula. 1997. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 106–119. Original paper.

[52] George C Necula and Peter Lee. 1998. Safe, untrusted agents using proof-carrying code. *Mobile agents and security* (1998), 61–91.

[53] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[54] Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393* (2020).

[55] Romain Robbes and Michele Lanza. 2010. Improving code completion with program history. *Automated Software Engineering* 17 (2010), 181–212.

[56] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 1–10.

[57] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A user study on the security implications of large language model code assistants. *USENIX Security* (2023).

[58] Fred B Schneider. 2000. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)* 3, 1 (2000), 30–50.

[59] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 71–82.

[60] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering*. 43–52.

[61] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*. 101–110.

[62] Jinghan Sun, Shaobo Li, Jun Xu, and Jian Huang. 2023. The Security War in File Systems: An Empirical Study from A Vulnerability-centric Perspective. *ACM Trans. Storage* 19, 4, Article 34 (oct 2023), 26 pages. https://doi.org/10.1145/3606020

[63] Julien Vanegue. 2014. The weird machines in proof-carrying code. In *2014 IEEE Security and Privacy Workshops*. IEEE, 209–213.

[64] David Walker. 2000. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 254–267.

[65] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2024. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.

[66] James Wilcox. 2018. notes/fileio.dfy at master · wilcoxjay/notes — github.com. https://github.com/wilcoxjay/notes/blob/master/fileio.dfy. [Accessed 10-08-2024].

[67] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. 2021. TacticZero: Learning to Prove Theorems from Scratch with Deep Reinforcement Learning. arXiv:2102.09756 [cs.LG]

[68] Shizhuo Dylan Zhang, Talia Ringer, and Emily First. 2023. Getting More out of Large Language Models for Proofs. *arXiv preprint arXiv:2305.04369* (2023).