

Pairing Human and Artificial Intelligence: Enforcing Access Control Policies with LLMs and Formal Specifications

Carlos E. Rubio-Medrano
carlos.rubiomedrano@tamucc.edu
Texas A&M University - Corpus Christi
Corpus Christi, Texas, USA

Wenlu Wang
wenlu.wang@tamucc.edu
Texas A&M University - Corpus Christi
Corpus Christi, Texas, USA

Akash Kotak
akotak@islander.tamucc.edu
Texas A&M University - Corpus Christi
Corpus Christi, Texas, USA

Karsten Sohr
sohr@tzi.de
University of Bremen
Bremen, Germany

ABSTRACT

Large Language Models (LLMs), such as ChatGPT and Google Bard, have performed interestingly well when assisting developers on computer programming tasks, a.k.a., *coding*, thus potentially resulting in convenient and faster software constructions. This new approach significantly enhances efficiency but also presents challenges in unsupervised code construction with limited security guarantees. LLMs excel in producing code with accurate grammar, yet they are not specifically trained to guarantee the security of the code. In this paper, we provide an initial exploration into using formal software specifications as a starting point for software construction, allowing developers to translate descriptions of security-related behavior into natural language instructions for LLMs, a.k.a., *prompts*. In addition, we leveraged automated verification tools to evaluate the code produced against the aforementioned specifications, following a modular, step-by-step software construction process. For our study, we leveraged Role-based Access Control (RBAC), a mature security model, and the Java Modeling Language (JML), a behavioral specification language for Java. We test our approach on different publicly-available LLMs, namely, OpenAI ChatGPT 4.0, Google Bard, and Microsoft CoPilot. We provide a description of two applications—a security-sensitive Banking application employing RBAC and an RBAC API module itself—, the corresponding JML specifications, as well as a description of the prompts, the generated code, the verification results, as well as a series of interesting insights for practitioners interested in further exploring the use of LLMs for securely constructing applications.

CCS CONCEPTS

• **Security and privacy** → **Access control**; **Software security engineering**; • **Computing methodologies** → **Artificial intelligence**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SACMAT 2024, May 15–17, 2024, San Antonio, TX, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0491-8/24/05

<https://doi.org/10.1145/3649158.3657032>

KEYWORDS

Large Language Models, ChatGPT, Prompt Engineering, Formal Specifications, Software Construction, Java Modeling Language.

ACM Reference Format:

Carlos E. Rubio-Medrano, Akash Kotak, Wenlu Wang, and Karsten Sohr. 2024. Pairing Human and Artificial Intelligence: Enforcing Access Control Policies with LLMs and Formal Specifications. In *Proceedings of the 29th ACM Symposium on Access Control Models and Technologies (SACMAT 2024)*, May 15–17, 2024, San Antonio, TX, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3649158.3657032>

1 INTRODUCTION

Released in late 2022, ChatGPT took the world by surprise, and rapidly became a technological and media sensation in early 2023. Based on Large Language Models (LLMs) [3], a popular technique under the vast spectrum of *Generative Artificial Intelligence* (AI), it can eloquently summarize vast amounts of knowledge as a response to user-issued *prompts*, and has reportedly performed impressively well in domains such as financing, journalism, and fiction writing.

Despite the hype, some pitfalls concerning ChatGPT have been actively discussed by experts, as it may lack the fundamental features devised for a broader, *General Purpose AI* [39]. As an example, when asked for research papers, it makes them up completely (e.g., hallucination) [22]. Also, ChatGPT has been criticized as it may lack enough *safety* and *security* controls. For instance, it may refuse to serve controversial content when asked directly, but workarounds, e.g., asking for the same content without explicitly mentioning the controversial parts, are possible [30].

LLMs in general, have been prophesied as the next evolution of computing programming, a.k.a., *coding*, as conversations with LLMs may result in code being produced, in a variety of different programming languages [16]. Such a feature has become really attractive, as it may significantly alleviate the burden of coding, e.g., design, implementation, and debugging, resulting in faster, more efficient developments and improved maintenance over time [1].

Lately, there has been an interest in augmenting Generative AI approaches such as LLMs with human expert intelligence for enhanced results [37, 17, 24]. In our particular context, there is a need for human-in-the-loop *pipelines* for developing applications from scratch securely and efficiently using LLMs (e.g., ChatGPT, Google Bard / Gemini, Microsoft Copilot or CodeX). To this aim, we propose a pipeline for developing security-sensitive applications using

LLMs by leveraging the concept of *chain of thought* [40], which refers to a series of connected cognitive reasoning steps in a person’s thinking. Our method is to decompose a conceptual complex task (described and defined in natural language) into a series of structured and coherent sub-tasks, facilitating a logical progressive process throughout the code development. Specifically, we decompose the pipeline for secure constructing applications into three stages: composing specifications in the Java Modeling Language (JML) [4], generating code from JML specifications, and refining and validating code using JML-based validation and verification tools (V&V). Each stage is facilitated by LLMs to reduce the amount of human effort required. The proposed pipeline aligns with the logic progression followed by software programmers when developing applications. Decomposing the complex task into manageable sub-steps also enables human-in-the-loop correction, where software programmers can refine and validate the application’s security features at different levels.

With that in mind, this paper reports on our experiences when performing two case studies in which we leveraged our specification-based pipeline approach to construct two applications, a Banking Application as well as a role-based authorization API. The Banking Application implements banking functionality, such as depositing, withdrawing, or transferring money, as well as security-related functionality such as restricting the execution of such operations to certain users only (authorization). The authorization API implements common concepts for Role-Based Access Control (RBAC) [32], a well-known access control model.

Overall, this paper makes the following contributions:

- (1) In Sec. 2 and Sec. 3, we introduce a discussion on the functionality that LLM must provide to effectively serve as assistants to human developers while constructing software applications, which may include, but may not be limited to: (i) the acquisition of previously-unknown knowledge; (ii) the reduction of the overall workload; and, (iii) the support on effectively removing functional bugs and security vulnerabilities at the source code level.
- (2) In Sec. 4, we present a systematic, pipeline-inspired approach combining LLMs and human intelligence, supporting task of constructing secure code by using LLMs and formal specifications, such that not only LLMs can be leveraged to obtain more knowledge on the implementation domain of such applications, but can be also leveraged to automatically generate and correct code as directed by the results of verification and validation tools.
- (3) Finally, in Sec. 5, we present the results and the lessons learned while conducting two case studies in which LLMs were leveraged as assistants using our proposed approach. We provide a description of the JML specifications, the LLM prompts, the generated code, the verification results, as well as a series of interesting insights for future practitioners. Overall, our results show that LLMs can: (i) generate non-trivial functional and security requirements; (ii) understand, generate, infer, and explain JML specifications; (iii) generate code from JML specifications; (iv) understand, generate, explain, and correct RBAC policies; (v) correct code based on feedback received from verification tools;

```

1  /*@ requires array.length > 0;
2  @ ensures (\forallall int i;
3  @         0 <= i < array.length-1;
4  @         array[i] <= array[i+1] );
5  @*/
6  public void sortAscending (int[] array){
7      //bubbleSort(array);
8      quickSort(array); }

```

Figure 1: A Sample DBC Contract.

and, (vi) translate JML specifications to other programming and annotation-based specification languages.

2 BACKGROUND AND RELATED WORK

2.1 Assertion-based Contract Specifications

Design by Contract (DBC) [19] is a well-known software development methodology in which the *original* developers of a software module M establish a *contract* with other *future* developers, a.k.a., *clients*, for M to be used correctly when building software. Commonly, such a contract is defined in terms of formal descriptions (specifications), a.k.a., *assertions* [31] in the form of *pre-* and *post-*conditions: before using M , clients must make sure that M ’s pre-conditions hold. Similarly, the original developers must guarantee that M ’s post-conditions hold once it has finished execution. Fig. 1 shows a DBC contract written in the *Java Modeling Language* (JML) [4], a DBC-inspired specification language for Java modules, i.e., the class method `sortAscending`. Pre-conditions (line 1), expressed by means of the `requires` keyword, require the input *array* to have a length greater than zero. Post-conditions (lines 2-4), expressed by means of the `ensures` keyword, state that, if the pre-condition just mentioned is met, the input *array* will have all elements in ascending order. Finally, as shown in lines 7-8, the original developers may choose among different options to implement the `sortAscending` method.

2.2 Verification and Validation Tools

Over the years, a suite of tools has been developed to verify the correctness of software modules by checking their source code implementations against their DBC contracts. As an example, the contract of a module M can be used as a *test oracle* by first translating it into *Runtime Assertion Checking* (RAC) code [7]. Then, values are *randomly* created for each of M ’s formal parameters, and compared against the RAC code created for M ’s pre-condition. If such a pre-condition is satisfied, a *valid* test case is said to be created. Otherwise, the test case is discarded. If the test case is *valid*, M ’s body and the RAC code for M ’s post-condition are both executed. If the post-condition is satisfied, the test is regarded as a *success*, and as a *failure* otherwise. Going back to Fig. 1, a set of *valid* test cases would include producing instances of the input parameter *array* whose length is greater than zero. Once the test cases are executed, if the resulting *array* always has all of its elements correctly in ascending order, the `sortAscending` method is said to have been verified successfully.

Beyond the dynamic validation approach with RAC, also static techniques and supporting tools exist. One well-established technique is *Extended Static Checking* (ESC). Extended Static Checkers

```

1  RBAC3 :
2  - a set of Roles R,
3  - a set of Users U,
4  - a set of Permissions P,
5  - a Role Hierarchy RH,
6  - a role-user assignment  $UA \subseteq R \times U$ ,
7  - a role-permission assignment  $PA \subseteq R \times P$ ,
8  - a set Card of constraints on RH, UA, PA.

```

Figure 2: The RBAC₃ Security Model.

attempt to statically establish DBC assertions (contracts) without any need for running the application. To be effective, ESC is based upon the principle of *modular reasoning*, which implies that it attempts to verify one method at a time [14]. For verifying a method’s assertions, it uses the contracts of the methods called within rather than the code itself. The verification is then carried out by translating a method’s code and specifications into verification conditions, which in turn are checked by a built-in automated theorem prover. As the problem of Extended Static Checking is not decidable in general, false positives and negatives may be produced. The OpenJML tool-suite offers both RAC and ESC capabilities, which makes it an attractive validation platform for Java-based contracts [10]. Recently, a German security authority has mentioned OpenJML as one of the tools supporting the formal verification of secure software [2] demonstrating its relevance. Also, Bertrand Meyer, who first developed the DBC methodology described in Sec. 2.1, points out the problems of AI-based automated code generation by giving a detailed example. To overcome such problems, he proposes: “For the past few years, in the competition with remarkable new subjects such as (surprise) machine learning, these disciplines of requirements analysis, precise specification, and software verification (both dynamic tests and static analyses including proofs) have taken a second seat.” [20] Our work follows this line of thinking.

2.3 Role-Based Access Control

Role-Based Access Control (RBAC) has is the leading security model for defining and enforcing authorization properties in software systems, mainly due to its flexibility and manageability [32]. Over the years, RBAC has developed to the de-facto standard for access management in industry due to its economic impact [23]. Fig. 2 shows a formal description of RBAC₃, one of the different sub-models of RBAC. The basic components, roles, users, and permissions, are defined in the context of sets (lines 1-3). Roles are in turn organized into a partial order called the *role hierarchy* (line 5). For instance, a role $r \in R$ is said to be *senior* to another role $r' \in R$, and, conversely, role r' is said to be *junior* to role r if and only if $(r, r') \in RH$. Users are assigned to roles by means of the UA relation (line 6). Conversely, permissions are assigned to roles by means of the PA relation (line 7). Constraints on the RH, UA, and PA relations are shown in line 8. For instance, the constraint $(u, UA, 1)^* \in \text{Card}$ restrict every user $u \in U$ to be assigned to a single role only. Finally, a user $u \in U$ is granted a permission $p \in P$ if and only if there exists (u, r) for some role $r \in R$, and either $(r, p) \in PA$ or $(r', p) \in PA$ for some role $r' \in R$ and $(r, r') \in RH$.

2.4 Using LLMs for Coding

LLM (Large Language Models) is a deep model that is trained on large-scale datasets with the aim of generating human-level text [3].

One notable and widely discussed LLM is the GPT (Generative Pre-trained Transformer) family developed by OpenAI, including GPT [28] and GPT2 [29]. These models are trained to predict the next token based on the previously generated tokens. LLM has also gained attention for its potential in generating near-human-level code. It can generate API calls [26], aid in self-debugging [5, 27], and automate data analysis [6].

Despite the impressive performance of LLM in coding tasks, it still needs to be improved to reach human-level proficiency. Various techniques are being employed to enhance the accuracy of code generation. One popular approach is to incorporate the concept of a *chain of thoughts* (COT) into the prompting [40]. COT refers to a series of connected reasoning steps or ideas in the thinking process of humans. Drawing inspiration from the success of COT techniques, a line of work seeks to promote coherent and structured reasoning for coding, such as turning one complex coding task into several rounds with self-debugging and few-shot demonstrations [5], decomposing a complex coding task into smaller sub-tasks [27, 25], and dissecting intricate tasks into more digestible sub-problems tackled in a sequential manner (e.g., “least-to-most prompting” [41]). Another line of work seeks to understand code generation via explanations. One study demonstrates the LLM’s promise in reducing the steep learning curve associated with understanding code semantics [18]. Meanwhile, another study evaluates on GitHub Copilot, a tool powered by LLMs. While initial indications hint at the tool’s efficacy in assisting with code inception, the study elucidates the challenges of interpreting and debugging code snippets generated by Copilot [36]. People have also started to realize the importance of LLM’s security issues. For example, Subhash et al. explore Gradient-based universal adversarial attacks on large language models to ensure the models’ responsible and secure deployment for high-risk sectors [35]. However, we focus on the security issues of LLM-generated codes instead of adversarial attacks on LLM itself, which has not been much observed in the literature. A recent work in this area comes from Sandoval et al. [33]. They carried out a study with Computer Science students, which points out that only about 10% more typical low-level security bugs have been introduced in C code while using an LLM. Although this rate is surprisingly moderate, one can, however, conclude that the productivity gained by employing an LLM did not lead to more secure code.

3 PROBLEM STATEMENT

The tool-based, automated/semi-automated generation of code based on a series of pre-defined requirements has been largely studied in the literature in the past, applying a plethora of different approaches and techniques, with varying results [15]. Unfortunately, a constant among these approaches is the lack of adoption in practice [34]. However, given the emerging popularity of LLMs, and the promising results obtained when asked to perform coding tasks as described in Sec. 2.4, we believe there is an enormous potential for the widespread adoption of such a technique. Therefore, in this paper we are interested in exploring the capabilities of publicly-available LLMs to answer the following research question:

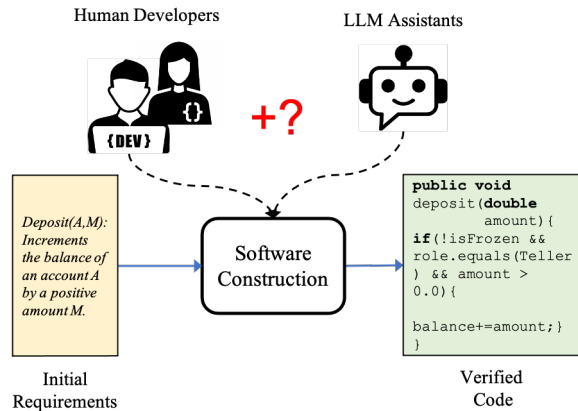


Figure 3: The research question considered in this paper: Are LLMs effective assistants for building secure applications?

RQ₁: Can publicly-available LLMs be effectively and efficiently leveraged for securely constructing software applications?

Concretely, an approach combining human developers, LLMs, and other relevant tools like the ones described in Sec. 2.2 should not only make the development process faster, but should also significantly contribute to producing a better and more secure application. That includes successfully meeting the following goals:

- **G₁ [Knowledge]:** An LLM-based approach must assist human developers on improving the domain-specific functional as well as the security-related *knowledge* needed to successfully build a secure application. Knowledge can be improved in a *quantitative* way, e.g., obtaining a larger number of relevant functional and security requirements; or in a *qualitative* way, e.g., improving the target rules of access control policies to be enforced by the application under construction.
- **G₂ [Workload]:** An LLM-based approach must assist human developers with reducing the overall *workload* required to successfully build a secured application. In this context, workload can be defined in terms of the efforts required to carry out the design, implementation, and testing of the target application, and can be measured by the overall time required for such tasks, the number of coding hours, the number of hours spent on performing tests, etc.
- **G₃ [Verification]:** An LLM-based approach must assist human developers with detecting and removing any software inconsistencies and/or errors, a.k.a., *bugs* in both functional and/or security-related implementation code. That may include classical functional bugs such as arithmetic miscalculations as well as security-related ones such as vulnerabilities introduced in the enforcement code of access control policies.
- **G₄ [Validation]:** Finally, an LLM-based approach must assist human developers with constructing a software application that meets both its functional and security requirements. That includes making sure that the implementation code is correctly verified and validated with respect to the requirements enunciated in G₁ [Knowledge], and has removed any bugs/vulnerabilities as stated in G₃ [Verification].

In Sec. 5, we present the lessons we learned while conducting two case studies based on the approach we present in Sec. 4, which provide empirical evidence towards answering the research question considered in this paper.

4 OUR APPROACH: COMBINING HUMAN AND ARTIFICIAL INTELLIGENCE

We now present our initial exploratory work to answer the research question presented in Sec. 3. Concretely, we are interested in leveraging LLMs to assist human developers on transforming a series of initial requirements, possibly contained within a software design document written in English, into a securely tested software application by means of a systematic construction approach.

4.1 Preliminary Assumptions.

For the purposes of our approach, we assume that human developers have accumulated experience on software construction/development matters above an entry, a.k.a., *junior*, level. That may include pro-efficient coding skills in Object Oriented Programming (OOP), e.g., encapsulation, inheritance, and polymorphism, as well as testing and debugging. Also, we assume developers are knowledgeable in the topics discussed in Sec. 2, i.e., formal specifications (Sec. 2.1), specification-based V&V tools (Sec. 2.2), the RBAC security model (Sec. 2.3), and prompt engineering techniques for LLMs (Sec. 2.4). Finally, the development of an approach at the *beginner* and junior experience levels, e.g., programming students and/or recent graduates, or *senior* developers not familiar with the topics discussed in Sec. 2, is left as a very interesting line of future work.

4.2 A Pipeline for Secure Software Construction.

As shown in Fig. 4, our approach consists of a pipeline that leverages LLMs, formal specifications, as well as verification and validation tools, all combined as follows:

- (1) *Requirements Gathering.* We start with a pre-defined *initial* list of functional and security software requirements, written in the natural language, which is provided by the human developers. Our goal is to collect minimum human knowledge as seed. Next, we ask the LLM assistants to generate an additional list of natural language requirements, which are later combined with the initial one to produce a *final* list that is consumed in subsequent steps of our pipeline.
- (2) *Preliminary Specifications.* Next, we asked our LLM assistant to generate a series of preliminary, a.k.a., *rough*, JML specifications based on such natural language requirements.
- (3) *Specification Refinement.* Our next step consists on *manually* customizing, a.k.a., *refining*, the JML specifications obtained in the previous step, in an effort to better describe the intended behavior of the software, e.g., handle special and/or corner cases, exceptions, security violations, etc. This step may require proficiency in the formulation of JML specifications, since even when the specifications produced by the LLM in the first place may be of decent quality, still the input of an expert is desired to correctly describe security and functional behavior. We argue that our approach with minimal supervision ensures superior

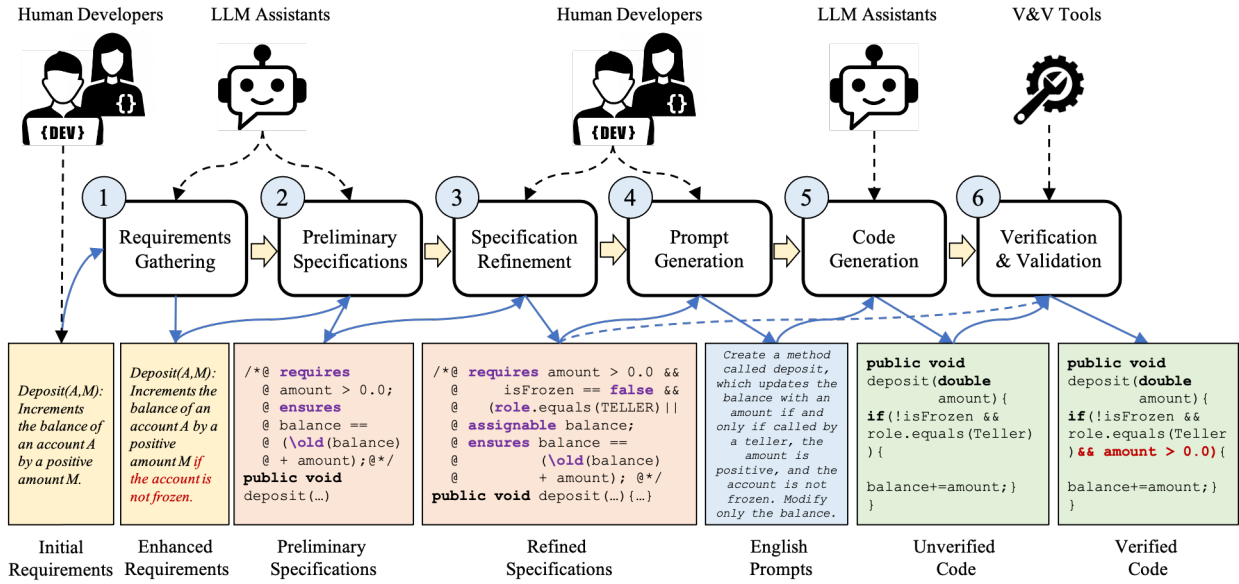


Figure 4: A Pipeline for Securely Constructing Applications Using Formal Specifications and LLMs.

security quality, in contrast to designing from scratch or without non-supervision.

- (4) *Prompt Generation.* Once the JML specifications have been refined, we propose to use them as *scripts* for generating prompts, which will be then later fed to the LLM. For this process, we identify three different techniques: (i) *Syntax-Guided Approach:* the prompts are created by *wrapping* the specifications as follows: we start with the post-conditions, and then continue with pre-conditions, processing one requires/ensures clause at a time. Inside each clause, we unwrap each complex statement, and produce a prompt separately, which is added to the result pool. (ii) *Specs-Assisted Prompts:* The prompts are created by asking the LLM to explain a subset of the JML specifications, which are fed one at a time as explained in the previous technique. Then, from each explanation, an individual prompt is generated separately and added to the result pool. (iii) *LLM-Assisted Prompts:* This approach proceeds as the previous one by submitting the JML specifications to the LLM. However, this time, the LLM is asked to produce the prompts directly, without any extra content or explanation. From there, a manual customization of each prompt may take place if desired. As before, each generated prompt is finally added to the result pool.
- (5) *Code Generation.* Each individual prompt in the result pool is then fed to the LLM to produce code. Related prompts may also be combined together if needed, e.g., the prompts related to a single Java method. The resulting code is then merged into a single Java module and compiled locally for correctness.
- (6) *Code Verification & Validation.* Finally, the code generated by the LLM is forwarded to the JML-based V&V tools provided by the OpenJML framework, i.e., RAC and ESC, as described in Sec. 2.2. If errors are found by those tools, these are fed back as prompts to the LLM and it is asked what went wrong. If the LLM cannot provide a convincing solution to the issues, then

the code is manually modified and fed again to the V&V tools. Alternatively, the JML specifications can also be modified manually, as illustrated in Step 3, and the pipeline is restarted from that point onward: prompt generation (Step 4), code generation (Step 5) and verification (Step 6).

5 CASE STUDIES

In this section, we present a pair of case studies we conducted following the pipeline approach discussed in Sec. 4, which leverages a series of publicly-available LLM assistants for securely constructing software. We start in Sec. 5.1 by describing the methodology we followed for both studies, which includes a summary of the sessions conducted with our LLM assistants as well as a description of the criteria used to evaluate the effectiveness of the information retrieved from them. Next, in Sec. 5.2, we describe the construction of a banking application that implements RBAC as a security model to restrict access to critical functionality, e.g., withdrawing money from an account. Later, in Sec. 5.3, we describe the development of an Application Programming Interface (API) for the RBAC security model. For each case study, we present the natural language (English) requirements, the preliminary and *refined* JML specifications, the code obtained from our LLM assistants, as well as the final code that was obtained after using JML-based tools. Finally, in Sec. 5.4, we present a summary of the lessons we learned during our case studies, which we hope are useful for future practitioners interested in leveraging our approach for secure software construction.

5.1 Methodology

LLM Sessions. We carried out a series of interactive sessions with the LLMs considered in our study by leveraging the COT techniques described in Sec. 2.4. We started our sessions by asking our LLM

<pre> 1 Balance(A): Returns the balance of the bank 2 account A. Such balance must never 3 be negative. 4 Deposit(A,M): Increments the balance of A by 5 a positive amount M. 6 Withdraw(A,M): Decrements the balance of A by 7 a positive amount M if and only 8 if the balance of A is greater or 9 equal to M. 10 Transfer(A,B,M): Increments the balance of A 11 by a positive amount M, which is 12 then decremented from the balance 13 of B if and only if the balance 14 of B is greater or equal to M.</pre>	<pre> 1 PayInterest(A,R): Increments the balance of A 2 by a percentage amount determined 3 by interest rate R, for instance, 4 A = A + (A * (R/100)). 5 The value of R must be a value 6 greater than 0 but less than 100. 7 Freeze(A): Denies the subsequent execution of 8 the Balance, Deposit, Withdraw, 9 PayInterest, and Transfer 10 operations for the account A. 11 Unfreeze(A): Allows the subsequent execution 12 of the Balance, Deposit, Withdraw, 13 PayInterest, and Transfer 14 operations for the account A.</pre>
---	--

(a) Initial Requirements Obtained from Human Developer.

(b) New Requirements Obtained from LLM Assistants.

Figure 5: Sample Requirements for a Banking Application.

```

1 /*@ requires amount > 0.0 &&
2   @ amount <= acc.balance;
3   @ ensures ((acc.balance ==
4   @ \old(acc.balance) - amount) >= 0);
5   @*/
6 public boolean withdraw(double amount){...}
```

Figure 6: Preliminary Specifications for `withdraw` generated by ChatGPT.

```

1 /*@ requires amount > 0;
2   @ requires balance >= amount;
3   @ ensures \old(balance) - amount == balance; */
4 public void withdraw(int amount) {...}
```

Figure 7: Preliminary Specifications for `withdraw` generated by Google Bard.

```

1 //@ requires amount > 0 && balance >= amount;
2 //@ ensures balance == \old(balance) - amount;
3 public boolean withdraw(double amount){...}
```

Figure 8: Preliminary Specifications for `withdraw` generated by CoPilot.

assistants about any specific real-life deployments of the technologies considered in our case studies, in order to provide a suitable starting context. Next, we asked our LLM assistants about specific information related to each of the steps of our proposed pipeline approach, e.g., for Step 1, we inquired about a series of specific functional and security requirements, e.g., operations and security policy rules, that a secure software *back-end* should implement, as well as a proper explanation for each of them. Later, we asked the LLM assistants to provide more contextual information on each of the responses provided in the previous step, e.g., a description of potential use-case scenarios, in an attempt to highlight their importance for real-life deployments. Finally, we also asked our LLM assistants to provide any *sources*, e.g., documents, websites, articles,

etc., for each provided response, as well as any additional relevant information, e.g., the date when the source was first published.

V&V Tool Sessions. In the validation and verification step, we use OpenJML’s RAC and ESC (see Section 2.2). The former tool checks JML specs, as depicted in Fig. 12, against the LLM-generated code at runtime, whereas the latter statically checks the generated code against the specifications, i.e., without running the code.

5.2 Case Study I: A Banking Application

We conducted our experiments in this case study with three LLM assistants, ChatGPT, Bard, and CoPilot.

Step 1. Requirements Gathering. The initial list of functional requirements for the Banking Application, as defined by a human developer with experience as a regular *user* of banking operations, are listed in Fig. 5(a), and contained basic operations such as `Deposit`, `Withdraw`, and `Transfer`. Later, we asked our LLM assistants for information on additional banking operations that could be implemented in the context of our application. As a result, we obtained several different options besides the ones listed in Fig. 5(a). Fig. 5(b) shows a brief summary of additional operations that were finally included as requirements in the context of our banking application.

In addition, the Banking Application required a security policy restricting access to the aforementioned operations as follows:

- Three roles were identified: *Manager*, *Supervisor*, and *Teller*. Following Sec. 2.3, a role hierarchy is defined as: role *Manager* is senior to *Supervisor*, and role *Supervisor* is senior to *Teller*.
- Highly-sensitive operations must be reserved only for users assigned the role *Manager*, e.g., the `Freeze` and `Unfreeze` operations.
- Mildly-sensitive operations are reserved only for users assigned the role *Supervisor*, and roles seniors to *Supervisor* (*Manager*), e.g.: `Withdraw`, `Transfer`, and `PayInterest`.
- All other operations are accessible to all employees, despite their role assigned, e.g., `Balance` and `Deposit`.

Again, we asked our LLM assistants for information regarding this access control policy. For instance, we asked what operations should be carried out only for upper-level employees, e.g., *Managers* and *Supervisors*, and what operations should be available to all

```

1 /*@ requires amount > 0.0 &&
2 @ amount <= acc.balance &&
3 @ isFrozen == false &&
4 @ (role.equals(Role.SUPERVISOR) ||
5 @ role.equals(Role.MANAGER));
6 @ assignable balance, acc.balance;
7 @ ensures ((acc.balance ==
8 @ \old(acc.balance) - amount) >= 0)
9 @ ==> (\result == true) ||
10 @ (acc.balance ==
11 @ \old(acc.balance) - amount) < 0)
12 @ ==> (\result == false); */
13 public boolean withdraw(double amount){...}

```

(a) Specifications for the withdraw Method.

```

1 /*@ requires amount > 0.0 &&
2 @ amount <= acc.balance &&
3 @ isFrozen == false &&
4 @ (role.equals(Role.SUPERVISOR) ||
5 @ role.equals(Role.MANAGER));
6 @ assignable balance, acc.balance;
7 @ ensures balance ==
8 @ (\old(balance) + amount) &&
9 @ acc.balance ==
10 @ (\old(acc.balance) - amount);
11 @*/
12 public void transfer (double amount,
13 BankAccount acc){...}

```

(b) Specifications for the transfer Method.

Figure 9: Refined JML Specifications for a Banking Application.

employees. For our specific policy, we were to confirm most of the policy rules described above. Only minor changes were suggested and ultimately accepted. For instance, Google Bard suggested that the Freeze/Unfreeze operations should only be available to Managers. Initially, it was available to both Managers and Supervisors.

Overall, the information received from our LLM assistants with respect to requirements gathering provided *significant* evidence towards meeting goals G_1 [Knowledge], G_2 [Workload], and G_4 [Validation].

Step 2. Preliminary Specifications. Following our proposed approach, we asked our LLM assistants to retrieve a series of JML specifications for the banking operations shown in Fig. 5(a) and Fig. 5(b). As shown in Fig. 6 and Fig. 8, LLM assistants are able to provide syntactically correct specifications. However, they make limited use of JML syntax, and they may not be able to fully cover different specification cases as needed by Human Developers. As an example, in Fig. 6, JML specifications restricting the execution of the `withdraw` method to roles Supervisor and Manager, as required by the policy described in a previous step, were missing. Also, the JML assignable clause, which limits the variables and parameters that can be potentially modified by method `withdraw` is also missing. Such a specification is crucial to prevent the method from introducing unwanted *side effects* [8].

Step 3. Refined Specifications. To express the requirements in a more formal and precise way, we leveraged the preliminary specifications depicted in formulated them as JML specifications as shown in Figs. 9(a) and 9(b). For example, the JML specification for the `withdraw` method (see Fig. 9(a)) includes preconditions stating that (1) the roles Supervisor and Manager are required, (2) the amount must be greater than or equal to 0.0, (3) the amount must be less than or equal to the account’s balance and (4) finally the account must not be frozen. The specifications for the other methods were defined in a similar way, and required a moderate-to-considerable effort from the human developers before the *refinement* could be considered as optimal for further steps of our case study.

Overall, the information obtained from our LLM assistants with respect to providing useful JML specifications via Steps 2 and 3 may be considered as *moderate* evidence towards meeting goals G_1 [Knowledge], G_2 [Workload], and G_3 [Verification].

Step 4. Prompt Generation. As described in Sec. 4, we conceived three different strategies to generate prompts based on JML specifications. Fig. 10 shows the prompts generated while using the *Syntax-Guided* approach on the JML post-condition specifications for the transfer method, shown in Fig. 9(b) (lines 7-10). Conversely, Fig. 11 shows the prompts generated for the JML pre-condition specifications of the same transfer method in Fig. 9(b). This time, the RBAC policy is disclosed as a single prompt (lines 1-3), and it is interleaved with other sample prompts for functionality behavior (lines 5-13). We performed the translation from JML specification to prompts in a *manual* way, e.g., without the use of any supporting implementation software, which is left as an interesting feature to explore as a part of future work.

Step 5. Code Generation. We provided all the generated prompts to our LLM assistants within the same session, starting with the prompts for JML post-conditions and continuing with prompts for JML pre-conditions. An illustrative excerpt of the obtained code, as generated by ChatGPT, is shown in Fig. 12. Lines 3-9 show the code generated for the `withdraw` method, which was obtained from prompts generated for JML post-conditions. Interestingly, ChatGPT generated code from JML pre-conditions separately, as depicted in lines 12-18, which shows code to allow for role Manager to not only execute the `withdraw` method, but to also implement the pre-condition checks required by our refined JML specifications.

As a comparison, Fig. 13 shows the corresponding code produced by GitHub’s CoPilot. It provided a simpler implementation, not introducing a Manager class, but with similar functionality.

Overall, the evidence provided by LLM assistant on coding generation, comprising Steps 4 and 5 of our approach, was *significant* toward meeting goals G_1 [Knowledge] and G_2 [Workload], as the workload of coding was significantly reduced once prompts for our JML specification were provided by human developers. This is consistent with other recent studies featuring the use of LLMs for code generation, as it is described in Sec. 2.4.

Step 6. Code Verification & Validation. We employed OpenJML’s RAC and ESC tools to validate the Java code generated in the previous step. Both tools reported problems with this code. For example, missing `spec_public` statements for private member fields were

Table 1: Assessing Goals for LLM Assistants via Case Study I.

Goal	Step 1	Steps 2-3	Steps 4-5	Step 6
G ₁ [Knowledge]	Significant	Moderate	Significant	Moderate
G ₂ [Workload]	Significant	Moderate	Significant	Moderate
G ₃ [Verification]	N/A	Moderate	N/A	Significant
G ₄ [Validation]	Significant	N/A	N/A	Moderate

```

1 Create Java code for a bank account
2
3 Create a method called transfer, which
4 deducts a certain amount of money from
5 an external bank account and pass it to
6 the current bank account

```

Figure 10: Prompts from a JML Ensures Clause.

reported by both RAC and ESC.¹ In another case, ESC reported a violation of a pre-condition of `Account.withdraw`. To understand this problem, consider lines 15 and 16 in Fig. 12. One can see that the method `withdraw` of the (inner) class `Manager` does not check the condition `!isFrozen` before calling `Account.withdraw` and hence the pre-condition of `Account.withdraw` cannot be satisfied (see line 7, where it is required that `isFrozen` must be false). Consequently, ESC correctly reported this violation.

We gave this feedback to ChatGPT and formulated a prompt stating that the pre-condition `!isFrozen` could not be established by ESC. Interestingly, it first tried to modify the JML specification accordingly. This, however, was not the fix we expected. So we requested ChatGPT to correct the code rather than the JML specification. It then actually came up with the corrected code by adding `!isFrozen` to the if-statement before the `account.isFrozen` call. It also explained its fix correctly:

- In this updated code, the `Manager.withdraw` method includes the additional precondition `!account.isFrozen` to establish the necessary condition for calling `account.withdraw`. This change ensures that the `!isFrozen` precondition of the `withdraw` method in the `Account` class can be proven when invoked by the `Manager` class. By incorporating this modification, ESC should be able to successfully verify the code with the desired preconditions and ensure that the account is not frozen before performing a withdrawal.

Overall, with respect to Step 6, the information retrieved by LLM assistants can be considered as *moderate* evidence with respect to meeting goals G₁ [Knowledge], G₂ [Workload], and G₄ [Validation]. Conversely, the assistance of the JML tools can be considered to be *significant* evidence with respect to G₂ [Workload], G₃ [Verification], and G₄ [Validation]. Finally, a summary of the assessment of the research goals addressed in this paper, with respect to our Case Study I, can be found in Table 1.

5.3 Case Study II: An API for RBAC

We also conducted a second case study, in which an RBAC API is implemented, which, for example, could be used by the Banking

¹Private member fields may only occur in JML specifications if they are defined with this JML statement.

```

1 Create a text policy file to represent
2 the following role hierarchy
3 "Teller:Supervisor:Manager"
4
5 Create Java code for an RBAC role
6
7 Create Java code transfer money from
8 one account to another, use the BankAccount
9 class created before
10
11 Before transferring money check that the
12 amount is greater than zero and the amount
13 is less or equal to the account balance

```

Figure 11: Prompts from a JML Requires Clause.

```

1 public class BankingApp {
2     public static class Account {
3         public void withdraw(
4             double amount) {
5             if(amount > 0 &&
6                 balance >= amount
7                 && !isFrozen){
8                 this.balance -= amount;
9             }
10        }
11    public static class Manager {
12        public void
13            withdraw(Account account,
14                double amount) {
15            if (account != null &&
16                amount > 0) {
17                account.withdraw(amount);
18            }
19        }
20    }
21 }

```

Figure 12: Sample Code Generated by ChatGPT.

```

1 public class BankAccount {
2     private double balance;
3     public void withdraw(double amount) {
4         checkFrozen();
5         balance -= amount;
6     }
7 }

```

Figure 13: Sample Code Generated by CoPilot.

Application for implementing role-based policies. The aim is to gain further experience with our LLM-based approach for constructing secure code and obtain hints how far our results can be generalized. The case study was carried out with ChatGPT and now also with Google Bard (instead of CoPilot) to conduct a broader comparison.

Step 1. Requirements Gathering. The requirement-gathering phase was simplified in this case. In particular, we let the LLM assistants help us find the functional and security requirements required for a basic RBAC-API implementation. Please consider the first two prompts depicted in Fig. 15 for this purpose. The second prompt


```

1 create a bank application in Java
2 with almost all the functionalities
3 that are required by a user to operate
4 a bank account and also for the bank
5 to be able to check for the banking
6 rules based on the United States Banking
7 System and Protocols.

```

Figure 14: Sample Prompts for Requirements.

```

1 Do you know RBAC?
2 Do you know the RBAC standard?
3 Can you produce a small Java module implementing a
  core RBAC API?
4 Can you extend this with role hierarchies?
5 And Static Separation of Duty?

```

Figure 15: Sample Requirements Prompts for an RBAC API.

(concerning the ANSI RBAC standard [12]) was meant to find out whether the LLMs knew the RBAC standard. As a result of these experiments, both LLMs did know core RBAC concepts including users, roles, and permissions as well as user and permission assignment and could explain these concepts to the user in response to the prompts (addressing G_1 [**Knowledge**]). Additionally, Google Bard even seems to know the ANSI RBAC standard.

Step 2. Preliminary Specifications and Step 3. Refined Specifications. From the results of the first two prompts, we concluded that the LLMs knew the principles of RBAC and its components quite well. There was no need (in our first attempt) to devise more detailed specifications. This corresponds to the procedures usual developers would follow as they often do not have a clear understanding of the components of an RBAC system and therefore do not know the functional and security requirements of such an access control system in advance.

Step 4. Prompt Generation and Step 5. Code Generation. Based upon the aforementioned observation, we simply asked the LLMs to implement core RBAC concepts (cf. the third prompt in Fig. 15). Both LLMs were then capable of generating code for a core RBAC API. In particular, both LLMs implemented an interface with the following functionality:

- `addUserToRole(u, r)`: Adds user `u` to role `r`.
- `removeUserFromRole(u, r)`: Removes user `u` from role `r`.
- `addPermissionToRole(r, p)`: Adds permission `p` to role `r`.
- `removePermissionFromRole(r, p)`: Removes permission `p` from role `r`.
- `hasPermission(u, p)`: Returns true if `u` has a role with permission `p`, otherwise false.

Both LLMs were also able to generate code for hierarchical RBAC as well as for Simple Static Separation of Duty (the last two prompts). Google Bard used Java inheritance for its implementation of hierarchical RBAC and Static Separation of Duty.

To give a concrete example, Fig. 16 and Fig. 17 show the code generated by ChatGPT and Google Bard for a method that checks whether a user has a specific permission (a kind of policy decision

```

1 public boolean checkAccess(User user, Permission
  permission) {
2     for (Role role : user.getRoles()) {
3         if (role.getPermissions().contains(
  permission)) {
4             return true;
5         }
6     }
7     return false;
8 }

```

Figure 16: The `checkAccess` Method Generated by ChatGPT.

```

1 public boolean hasPermission(String userId, String
  permissionName) {
2     for (String roleName : userRolesMap.
  getOrDefault(userId, Collections.emptySet
  ())) {
3         if (rolePermissionsMap.containsKey(
  roleName) && rolePermissionsMap.get(
  roleName).contains(permissionName)) {
4             return true;
5         }
6     }
7     return false;
8 }

```

Figure 17: The `hasPermission` Method Generated by Google Bard.

point). One can see that each code snippet looks reasonable and uses data structures for permission and user assignment. As a result, LLMs seem to help developers with the task of implementing core RBAC functionality. Employing LLMs, a developer, who might not be familiar with RBAC concepts, has a basis for implementing this code—this may reduce her workload (G_2 [**Workload**]).

Step 6. Code Verification & Validation. As mentioned above, we decided not to define formal specifications for the RBAC engine in advance as developers often do not have a clear picture of the RBAC system’s components. However, to address G_3 [**Verification**] and G_4 [**Validation**], we had to provide specifications or more precisely rules that static vulnerability checkers usually require for their work [9]. In our approach (see Fig. 4), we employ OpenJML (both RAC and ESC) as the verification & validation tool, implying that we still need JML specifications as rules. As annotating code is usually considered a burden, a more automated approach is desirable. Automated specification inference, however, is also a difficult task [13, 11, 21, 38]. For this reason, we decided to let the LLM help us and generate JML specifications from the code (in essence, doing specification inference for us). For this task, we entered the following prompt:

“Can you provide JML specs for each public method, while the code remains unchanged?”

Table 2: Assessing Goals for LLM Assistants via Case Study II.

Goal	Step 1	Steps 2-3	Steps 4-5	Step 6
G ₁ [Knowledge]	Significant	Moderate	Significant	Moderate
G ₂ [Workload]	Significant	Moderate	Significant	Significant
G ₃ [Verification]	N/A	Moderate	N/A	Significant
G ₄ [Validation]	Significant	N/A	N/A	Moderate

```

1 //@ requires userId != null && permissionName !=
  null;
2 //@ ensures \result == (\exists String roleName;
  userRolesMap.get(userId).contains(
  roleName) && rolePermissionsMap.containsKey(
  roleName) && rolePermissionsMap.get(roleName).
  contains(permissionName));

```

Figure 18: JML specs for hasPermission Generated by ChatGPT.

```

1 //JML specification for hasPermission method
2 @Requires({"userId != null", "permissionName !=
  null"})
3 @Ensures("result elementof {true, false}")

```

Figure 19: JML specs for hasPermission Generated by Bard.

Fig. 19 and Fig. 18 depict the JML specifications for the hasPermission method generated by Google Bard and ChatGPT, respectively. The specifications produced by Google Bard are not syntactically correct and furthermore do not seem to be helpful (see Fig. 19, line 3), whereas ChatGPT’s JML specifications are close to what is needed: They state, for example, that the user’s role set must contain a role that has the requested permission, correctly using JML’s exist expression (see Fig. 18). Having in mind that such specifications should be handled with caution and must likely be adjusted, they can be seen as good starting points alleviating the specification burden (G₂ [Workload]). Inference systems like Daikon [11] can only provide likely invariants rather than giving definitive results.

We finally used a slightly modified variant of the JML specifications with OpenJML’s ESC–ESC, but could not prove this specification as it is in general beyond the capabilities of static checkers to prove such complex specifications and hence future work is needed to improve such validation tools in this direction. Beyond checking, we can also feed inferred (and maybe adjusted) specifications back to step 3 of our pipeline.

As with Case Study I, we also give a summary of the assessment of the research goals we pursued (see Table 2). In particular, Step 6 contributes significantly to G₂ [Workload] and G₃ [Verification] due to its specification-inference capabilities.

5.4 Results and Lessons Learned

We now summarize the tasks that we performed with LLM and also report on our experiences.

- (1) *LLMs Can Generate Code from JML specifications for a Software Application.* LLMs were able to understand the JML specifications that we provided, similar to the specifications, for example, depicted in Fig. 9(a). There was some inconsistency in the code initially. However, overtime the code generation got improved as the model got updated. Fig. 12 depicts a code excerpt that has been generated by LLM-based on the given JML specifications, which in turn formulate the functional and security requirements. As an implementation detail, one can see that it generated Java inner classes for implementing the specific functionality for managers, supervisors, and tellers.
- (2) *LLM Can Generate Non-trivial Requirements for a Software Application.* As mentioned in Sec. 4, we also requested ChatGPT to complement and extend the functional and security requirements for our banking application. As shown in Fig. 14, we asked ChatGPT to provide requirements based on the United States Banking System and Protocols. As a response, ChatGPT provided the following:
 - Implement a createAccount method, which takes a Customer object and an initial balance as parameters. It generates a new account number, creates a new Account object, associates it with the customer, and adds it to the accounts map.
 - Implement a withdraw method, which returns a Boolean indicating whether the withdrawal was successful. It checks if the account exists and has sufficient balance before performing the withdrawal.
 - Add methods getAllAccounts and getAccountsByCustomer to retrieve all accounts or accounts associated with a specific customer, respectively.
- (3) *LLMs Can Understand, Generate, Infer, and Explain JML Specifications.* We conducted several experiments to find out if and how far LLM can understand JML and its concepts (see below for an example of how ChatGPT interprets JML specifications). ChatGPT was aware of JML and how to use the specifications to generate some basic related code from it. It could also generate JML specifications itself, i.e., we formulated some requirements of the Banking Application as textual prompts and requested ChatGPT to produce JML specifications from them. In this task, we had problems in finding the right English prompt for the required specifications to be generated at first. After formulating the right prompt, however, we were able to generate the JML specifications we expected. The generated code and the specifications were moderate enough to require some or no changes to them. It also seemed that ChatGPT could learn from the conversations and was aware of the context. Furthermore, ChatGPT was able to infer JML specifications from Java code as demonstrated by the second case study.
- (4) *LLMs Can Understand, Generate, Explain, and Correct RBAC Policies.* We also requested ChatGPT for assistance on explaining and implementing the RBAC policy shown in Sec. 5.2. We

```

1 Can you give me a brief overview
2 of RBAC and how it can be integrated
3 in our existing code ?
4
5 Update the existing code to use
6 RBAC principles and do not limit
7 it's functionality to what we
8 already have achieved instead try
9 to add as many features and
10 functionalities as you can.

```

Figure 20: Sample Prompts for RBAC.

started with general prompts and then ask the ChatGPT to incorporate the explanations given into our application context. As shown in Fig. 20, we first started with an initial request (lines 1-3), which was later extended to a second one (lines 5-10), in which we encouraged ChatGPT to provide more detailed, useful information. As a response, ChatGPT provided some useful information, whose sample is shown next. Besides functionality that could be directly added to the code shown in Fig. 12, ChatGPT also suggested the need to implement a full administrative system to allow for users to be properly assigned roles, thus assisting on better preserving the security properties shown in the RBAC policy of Sec. 5.2. Finally, ChatGPT also suggested that the RBAC policy is implemented by introducing authorization code checks before attempting any banking operation.

- *Maintain User-Role Mapping: Establish a mechanism to map users to their assigned roles. This mapping can be stored in a database or another data structure. You can associate each Customer object with a specific role or introduce a separate User class that holds the role information.*
- *Role Assignment and Management: Provide functionality to assign and manage roles for users. This could include methods to assign roles during user registration, modify roles based on user promotions or changes in responsibilities, and handle revocation of roles when necessary.*
- *Access Control Enforcement: Integrate RBAC into the existing code by verifying the user's role and permissions before executing specific operations. For example, when creating a new account, check if the user's role allows the creation of accounts. If not, deny the operation.*

We incorporated some of this functionality to our case study as relevant, and use the suggestions to generate more code-producing prompts, following the guidelines described in Sec. 4.

- (5) *LLMs Can Correct Code Based on Feedback from Validation Tools and Understand the Identified Problem.*

Step 6 (verification step) in the first case study confirms that the LLMs under study are able to understand JML specifications and possible problems related to that code if given precise feedback. In particular, LLMs can correct the code accordingly.

- (6) *LLMs Can Translate the JML Specifications to other Programming and Annotation Languages.* In another experiment, we requested ChatGPT to translate the generated code plus the

```

1 public void Withdraw(double amount)
2 {
3     Contract.Requires(amount > 0);
4     Contract.Requires(balance >= amount);
5     Contract.Requires(!isFrozen);
6     Contract.Ensures(balance ==
7     Contract.OldValue(balance) - amount);
8
9     if (amount > 0 && balance
10         >= amount && !isFrozen)
11     {
12         balance -= amount;
13     }
14 }

```

Figure 21: Sample C# Code from JML Specs.

JML specifications to other programming languages and corresponding annotation languages. When requesting C, it came up with C code and corresponding ACSL (Frama-C) specifications; similarly, it produced C# code and .NET Code Contracts. In an earlier ChatGPT session, however, it produced C# code annotated with JML specifications, which is not the result we expected. Also, we did not check if the specifications and the code are actually correct. Fig. 21 shows a sample snippet of the generated C# code along with some DBC-like specifications.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a pipeline approach as well as the details and the lessons learned from our case study. We can conclude that, when it comes to building applications using LLMs, generating code from JML specifications seems to be a preferred method compared to generating code directly from natural language descriptions. This is because JML specifications are logical expressions that represent a coherent chain of thought (Sec. 2.4) derived from natural language descriptions or software requirement documents. Using a chain of thought in prompting aligns with the “predicting next token” training objective and offers sequential guidance to LLMs. This guidance provides a consistent context throughout the process of code generation. An accurate code generation that adheres to security specifications will facilitate the correct implementation of security features of the final application. As a result of the two case studies conducted, we can conclude that LLM assistants provide moderate-to-significant assistance to human developers when it comes to producing secure applications leveraging our approach. We have identified the cases when LLMs perform significantly well, e.g., code generation, as well as tasks in which more work is needed, e.g., code verification and validation. Overall, the combination of human and artificial intelligence, facilitated by LLMs, formal specifications, and expert-level human developers, seems promising as automated code generation techniques evolve in the future.

As a part of future work, we plan to extend our case study to consider other security models and techniques, as well as other applications. Also, we plan to experiment with different ways to create prompts from JML specifications, as described in Sec. 4. Finally, we also plan to incorporate feedback obtained throughout the different

phases of our pipeline into a *reinforcement learning* approach, so we can incorporate new knowledge such as effective prompts and verified code into an existing LLM, thus potentially increasing its chances of producing better code results in the future.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation (NSF) under Grants No. 2150351, No. 2131263, and No. 2112631, and by the US Department of Transportation (USDOT) Tier-1 University Transportation Center (UTC) Transportation Cybersecurity Center for Advanced Research and Education (CYBER-CARE). (Grant No. 69A3552348332).

REFERENCES

- [1] Reed Albergotti and Louise Matsakis. 2023. Openai has hired an army of contractors to make basic coding obsolete. Semafor. <https://www.semafor.com/article/01/27/2023/openai-has-hired-an-army-of-contractors-to-make-basic-coding-obsolete>. [Online; accessed June-9-2023]. (2023).
- [2] Bernhard Becker, Oliver Denninger, Jonas Klamroth, Max Scheerer, and Jörg Henß. 2023. Formal Verification of Complex Software Systems—A Study. Agentur für Innovation in der Cybersicherheit GmbH, (Ed.) <https://www.cyberagentur.de/wp-content/uploads/2023/07/OevIT-Vorstudien-Los-1.pdf>. (2023).
- [3] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proc. of the 2007 Joint Conf. on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. Prague, Czech Republic, (June 2007), 858–867.
- [4] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2003. An overview of jml tools and applications1 www.jmlspecs.org. *Electronic Notes in Theoretical Computer Science*, 80, 75–91. Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS’03).
- [5] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.
- [6] Liying Cheng, Xingxuan Li, and Lidong Bing. 2023. Is gpt-4 a good data analyst? *arXiv preprint arXiv:2305.15038*.
- [7] Yoonsik Cheon. 2007. Automated random testing to detect specification-code inconsistencies. In *Proceedings of the 2007 International Conference on Software Engineering Theory and Practice*. Orlando, Florida, U.S.A.
- [8] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. 2005. Model variables: cleanly supporting abstraction in design by contract. *Softw., Pract. Exper.*, 35, (May 2005), 583–599. doi: 10.1002/spe.649.
- [9] Brian Chess and Jacob West. 2007. *Secure Programming with Static Analysis*. (First ed.). Addison-Wesley Professional. ISBN: 9780321424778.
- [10] David R. Cok. 2014. OpenJML: software verification for Java 7 using JML, OpenJDK, and Eclipse. *Electronic Proceedings in Theoretical Computer Science*, 149, (Apr. 2014), 79–92. doi: 10.4204/eptcs.149.8.
- [11] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69, 1, 35–45. Special issue on Experimental Software and Toolkits. doi: <https://doi.org/10.1016/j.scico.2007.01.015>.
- [12] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. 2001. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4, 3, (Aug. 2001), 224–274. doi: 10.1145/501978.501980.
- [13] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an annotation assistant for esc/java. In *FME 2001: Formal Methods for Increasing Software Productivity*. José Nuno Oliveira and Pamela Zave, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 500–517. ISBN: 978-3-540-45251-5.
- [14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended static checking for java. In *Proc. of the ACM SIGPLAN 2002 Conf. on Progr. Language Design and Implementation (PLDI ’02)*. Berlin, Germany, 234–245. ISBN: 1581134630. doi: 10.1145/512529.512558.
- [15] Mahadevan Ganapathi, Charles N Fischer, and John L Hennessy. 1982. Retargetable compiler code generation. *ACM Computing Surveys (CSUR)*, 14, 4, 573–592.
- [16] Gunnet Kaur. 2023. How to improve your coding skills using chatgpt. Coin-Telegraph. <https://cointelegraph.com/news/how-to-improve-your-coding-skills-using-chatgpt>. [Online; accessed June-9-2023]. (2023).
- [17] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. (2023). arXiv: 2305.01210 [cs. SE].
- [18] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. 2022. Generating diverse code explanations using the gpt-3 large language model. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2*, 37–39.
- [19] Bertrand Meyer. 1992. Applying “design by contract”. *Computer*, 25, 10, (Oct. 1992), 40–51.
- [20] Bertrand Meyer. 2022. What do chatgpt and ai-based automatic program generation mean for the future of software. *Blog entry—Commun. ACM*, 65, 12, 5.
- [21] Tanveer Mustafa and Karsten Sohr. 2015. Understanding the implemented access control policy of android system services with slicing and extended static checking. *Int. J. Inf. Secur.*, 14, 4, (Aug. 2015), 347–366. doi: 10.1007/s10207-014-0260-y.
- [22] Daniel Najafali, Justin M Camacho, Erik Reiche, Logan Galbraith, Shane D Morrison, and Amir H Dorafshar. 2023. Truth or lies? the pitfalls and limitations of chatgpt in systematic review creation. *Aesthetic Surgery Journal*, sjad093.
- [23] National Institute of Standards and Technology. 2020. Role-Based Access Control—RBAC. <https://csrc.nist.gov/projects/role-based-access-control/economic-impact>. [Online; accessed June-8-2023]. (2020).
- [24] Ansong Ni, Srinii Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. 2023. LEVER: learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research)*. Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, (Eds.) Vol. 202. PMLR, (July 2023), 26106–26128. <https://proceedings.mlr.press/v202/ni23b.html>.
- [25] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: an open large language model for code with multi-turn program synthesis. (2023). arXiv: 2203.13474 [cs. LG].
- [26] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.
- [27] Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: decomposed in-context learning of text-to-sql with self-correction. *arXiv preprint arXiv:2304.11015*.
- [28] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.
- [29] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1, 8, 9.
- [30] Kevin Roose. 2022. The brilliance and weirdness of chatgpt. The New York Times. <https://www.nytimes.com/2022/12/05/technology/chatgpt-ai-twitter.html>. [Online; accessed June-9-2023]. (2022).
- [31] D. S. Rosenblum. 1995. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21, 1, 19–31.
- [32] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. 1996. Role-based access control models. *Computer*, 29, 2, (Feb. 1996), 38–47.
- [33] 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. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, (Aug. 2023), 2205–2222. ISBN: 978-1-939133-37-3.
- [34] Jiho Shin and Jaechang Nam. 2021. A survey of automatic code generation from natural language. *Journal of Information Processing Systems*, 17, 3, 537–555.
- [35] Varshini Subhash, Anna Bialas, Weiwei Pan, and Finale Doshi-Velez. 2023. Why do universal adversarial attacks work on large language models?: geometry might be the answer. (2023). arXiv: 2309.00254 [cs. LG].
- [36] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, 1–7.
- [37] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. experience: evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA ’22) Article 332*. Association for Computing Machinery, New Orleans, LA, USA, 7 pages. ISBN: 9781450391566. doi: 10.1145/3491101.3519665.
- [38] Julien Vanegue and Shuvendu K. Lahiri. 2013. Towards practical reactive security audit using extended static checkers. In *2013 IEEE Symposium on Security and Privacy*, 33–47. doi: 10.1109/SP.2013.12.
- [39] Alexandros Vassiliades, Nick Bassiliades, and Theodore Patkos. 2021. Argumentation and explainable artificial intelligence: a survey. *The Knowledge Engineering Review*, 36, e5.
- [40] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.
- [41] Denny Zhou et al. 2023. Least-to-most prompting enables complex reasoning in large language models. (2023). arXiv: 2205.10625 [cs. AI].