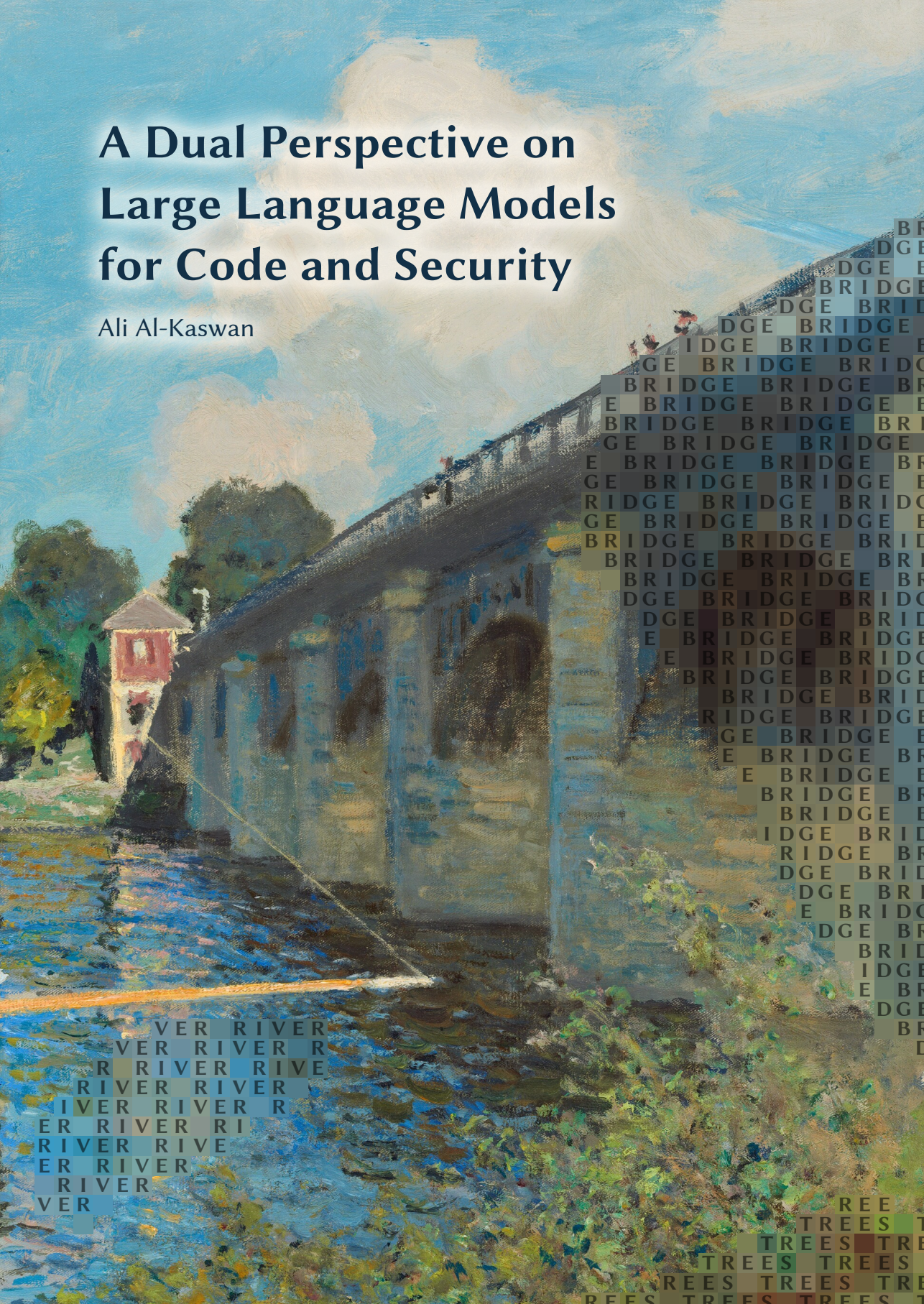
A Dual Perspective on Large Language Models for Code and Security

Ali Al-Kaswan



Propositions

accompanying the dissertation

A Dual Perspective on Large Language Models for Code and Security

by

Ali Al-Kaswan

1. The growing use of AI weakens the link between a software artefact and the individuals who can explain its origin and rationale. [This thesis]
2. In the age of AI, the balance between attackers and defenders will be determined less by raw model capability than by the capacity to govern its use.
3. AI ethics are a vehicle for the imposition of cultural and political values under the appearance of technical neutrality.
4. When we increasingly delegate cognition to AI, we must decide which forms of understanding we are willing to stop cultivating.
5. The central educational risk of AI is not plagiarism, but the accumulation of cognitive debt through habitual outsourcing of thought.
6. Because speculative investment is flooding the AI field, the true cost of developing and deploying AI models is not yet passed on to consumers.
7. Digital autonomy is a prerequisite for sovereignty. Countries that outsource critical IT infrastructure to foreign parties are trading that independence for convenience.
8. The true social value of open-source AI lies in resisting a future in which meaningful access to AI is reserved for those who can afford it.
9. As public resources and trust in science come under increasing pressure, scientists must become better at demonstrating their societal relevance.
10. The environmental and social costs of AI are disproportionately borne by marginalised communities.

These propositions are regarded as opposable and defendable, and have been approved as such by the promotors prof. dr. A. van Deursen and dr. M. Izadi

A Dual Perspective on Large Language Models for Code and Security

A Dual Perspective on Large Language Models for Code and Security

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus Prof. dr. ir. H. Bijl,
chair of the Board for Doctorates
to be defended publicly on
Wednesday, 16 December 2026 at 17:30

by

Ali AL-KASWAN

This dissertation has been approved by the (co)promotors.

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof. dr. A. van Deursen	Delft University of Technology, promotor
Dr. M. Izadi	Delft University of Technology, copromotor
Independent members:	
Prof. dr. G. Smaragdakis	Delft University of Technology
Prof. dr. P. Devanbu	University of California Davis
Dr. O. Gadyatskaya	Leiden University
Prof. dr. M. Pradel	University of Stuttgart
Prof. dr. J. Cito	Vienna University of Technology
Prof. dr. D. Spinellis	Delft University of Technology, reserve member

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).





Keywords: Software Engineering, Artificial Intelligence, Cyber Security

Printed by: ProefschriftMaken | www.proefschriftmaken.nl

Cover: The Bridge at Argenteuil, Claude Monet (1874), National Gallery of Art, Accession Number 1983.1.24, with modifications by the author
https://github.com/aalkaswan/thesis_cover

Style: TU Delft House Style, with modifications by Moritz Beller
https://github.com/Inventitech/phd-thesis-template

ISBN: 978-94-6518-516-3

IPA Series: 2026-20

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

لسانُ الفتى نصفٌ ونصفٌ فؤادُهُ
فلم يبقَ إلا صورةُ اللحمِ والدمِ

A man's tongue is half of him, and the other half his heart;
what remains is but a form of flesh and blood.

– Zuhayr ibn Abī Sulmā, *al-Muʿallaqa*

Contents

Summary xiii

Samenvatting xv

Acknowledgments xvii

1 Introduction 1
- 1.1 Background . . . 3
  - 1.1.1 Large Language Models for Code . . . 3
  - 1.1.2 Software Security as an Application Domain . . . 3
  - 1.1.3 LLM Agents . . . 4
  - 1.1.4 Evaluation of LLM Capabilities . . . 4
  - 1.1.5 Security Risks of Code-Oriented LLMs . . . 4
  - 1.1.6 Dual Use, Harmfulness, and Alignment . . . 5
- 1.2 Scope and Core Argument . . . 5
- 1.3 Goals and Research Questions . . . 6
- 1.4 Overarching Methods . . . 7
  - 1.4.1 Domain-Specific Dataset and Benchmark Construction . . . 7
  - 1.4.2 Model Adaptation and Task-Specific Evaluation . . . 7
  - 1.4.3 Comparative Analysis of Models and Techniques . . . 7
- 1.5 Chapters . . . 8
- 1.6 Data Sharing and Research Artefacts . . . 9
- 1.7 Origin of Chapters . . . 10

2 Pre-Trained Language Models to Summarise Decompiled Binaries 13
- 2.1 Introduction . . . 14
- 2.2 Background . . . 15
  - 2.2.1 Compilers and Optimisation Levels . . . 15
  - 2.2.2 Ghidra . . . 17
  - 2.2.3 Stripping . . . 17
  - 2.2.4 Code Summarisation Task . . . 18
  - 2.2.5 Transformer-based Models . . . 18
  - 2.2.6 Transfer Learning . . . 18
- 2.3 CAPYBARA Dataset . . . 18
  - 2.3.1 Data Collection . . . 19
  - 2.3.2 Dataset Preparation . . . 21
  - 2.3.3 Dataset Properties . . . 22

2.4 BinT5 . . . 24
2.5 Experimental Setup . . . 25
2.5.1 Research Questions . . . 25
2.5.2 Baselines . . . 26
2.5.3 Evaluation Metrics . . . 26
2.5.4 Data deduplication . . . 26
2.5.5 Configuration . . . 27
2.6 Results . . . 27
2.6.1 RQ1: Model Effectiveness . . . 27
2.6.2 RQ2: Input Properties . . . 28
2.6.3 RQ3: Dataset Properties . . . 29
2.7 Discussion . . . 31
2.7.1 Exploration of Results . . . 31
2.7.2 Identifiers and Extreme Summarisation . . . 31
2.7.3 Implications . . . 32
2.7.4 Threats to Validity . . . 32
2.8 Related Work . . . 33
2.9 Conclusion . . . 34

3 Do Agents Dream of Root Shells? 35
3.1 Introduction . . . 35
3.2 Background and Related Work . . . 38
3.3 Framework Design . . . 38
3.3.1 Benchmark Harness . . . 39
3.3.2 Dataset Curation . . . 40
3.3.3 Model Curation . . . 41
3.4 Partial Credit and Automated Labelling . . . 42
3.4.1 Automatic Labelling and Validation . . . 43
3.5 Results . . . 44
3.6 Discussion . . . 46
3.6.1 Agent Improvement Opportunities . . . 47
3.6.2 Threats to Validity . . . 47
3.6.3 Future work . . . 48
3.6.4 Reproducibility . . . 49
3.7 Conclusion . . . 49

4 Code Red! Harmful Large Language Models in Programming Tasks 51
4.1 Introduction . . . 51
4.2 Background and Related Work . . . 53
4.2.1 Red-Teaming in AI Safety . . . 53
4.2.2 LLM Alignment . . . 54
4.2.3 Harm Benchmarking . . . 54

4.3 Approach . . . 55
4.3.1 Taxonomy . . . 55
4.3.2 Prompt Creation . . . 57
4.3.3 Response Labels . . . 58
4.3.4 Evaluator Training . . . 58
4.4 Experimental Setup . . . 59
4.4.1 Research Questions . . . 59
4.4.2 Model Configuration . . . 59
4.4.3 Model Selection . . . 60
4.4.4 Manual Labelling . . . 61
4.4.5 Automatic Evaluation . . . 61
4.4.6 Automatic Evaluator Training . . . 62
4.5 Results . . . 62
4.5.1 RQ1: Models . . . 62
4.5.2 RQ2: Harmfulness Across Prompt Categories . . . 65
4.5.3 RQ3: Model Size versus Harmfulness . . . 68
4.6 Discussion . . . 69
4.6.1 Fine-Tuned Models . . . 69
4.6.2 Categories . . . 69
4.6.3 Model Defences . . . 69
4.6.4 Model Size . . . 70
4.6.5 Decoding Parameters . . . 70
4.6.6 Alternatives to Embedding-based Labelling Methods . . . 72
4.6.7 Warnings . . . 72
4.6.8 Imposition of Values . . . 73
4.6.9 Threats to Validity . . . 73
4.6.10 Future Work . . . 74
4.7 Conclusion . . . 75
4.7.1 Data Availability . . . 75
4.7.2 Acknowledgements . . . 76

5 Traces of Memorisation in Large Language Models for Code 77
5.1 Introduction . . . 77
5.2 Background and Related Work . . . 79
5.2.1 Memorisation . . . 79
5.2.2 Membership Inference Attacks . . . 80
5.2.3 Data Extraction Attacks . . . 80
5.2.4 Natural Language Dataset . . . 81
5.3 Approach . . . 82
5.3.1 Data Extraction Security Game . . . 82
5.3.2 Code Dataset Construction . . . 83
5.4 Methodology . . . 84
5.4.1 Research Questions . . . 84
5.4.2 Models . . . 84
5.4.3 Categorisation . . . 85
5.4.4 Extraction . . . 85

5.4.5 Evaluation Metrics . . . 85
5.4.6 Configuration . . . 86
5.5 Results . . . 86
5.5.1 Natural Language vs Code . . . 86
5.5.2 Type of Memorised Samples . . . 91
5.5.3 Which Model Memorises What . . . 91
5.5.4 Pre-Training Data Leakage . . . 96
5.6 Discussion . . . 97
5.6.1 Interpretation . . . 97
5.6.2 Implications . . . 99
5.6.3 Limitations and Threats to Validity . . . 100
5.7 Conclusion . . . 101

6 Model See, Model Do? 103
6.1 Introduction . . . 103
6.2 Background and Related Work . . . 105
6.3 Approach . . . 106
6.3.1 Research Questions . . . 107
6.3.2 Datasets . . . 107
6.3.3 Models . . . 107
6.3.4 Data Portraits . . . 108
6.3.5 Model Preference Metrics . . . 109
6.3.6 Model Generations . . . 110
6.3.7 Environment and Data Availability . . . 111
6.4 Exposure Characterization of Dataset . . . 111
6.5 Results . . . 113
6.5.1 RQ1: Exposure Influence on Metrics . . . 113
6.5.2 RQ2: Exposure Effects by Bug Category . . . 115
6.5.3 RQ3: Exposure Impact on Generations . . . 119
6.6 Discussion . . . 122
6.6.1 Threats to Validity . . . 124
6.6.2 Future Work . . . 125
6.7 Conclusion . . . 125

7 Conclusion 127
7.1 Revisiting the Research Questions . . . 127
7.1.1 RQ1: How can LLMs be adapted and rigorously evaluated for realistic software-security tasks? . . . 127
7.1.2 RQ2: What security, privacy, and safety risks arise from deploying code-oriented LLMs? . . . 128
7.2 Cross-Cutting Findings . . . 129
7.3 Data Sharing and Cumulative Research . . . 129
7.4 Broader Implications . . . 130
7.5 Societal Impact . . . 130
7.6 Limitations and Future Work . . . 131
7.7 Closing Remarks . . . 132

Bibliography **133**

Curriculum Vitæ **155**

List of Publications **157**

Summary

Artificial Intelligence (AI) systems that can read and write computer code, so-called Large Language Models (LLMs), are becoming a routine part of how software is built. Because these systems are good at reasoning about both code and natural language, they are not just useful for everyday programming, they are also relevant to security work, where understanding how a piece of software behaves, or how it might be attacked, is central. At the same time, the very properties that make these AI systems useful also create risks of their own: a model trained on huge amounts of code can accidentally leak private information from that training data, produce code with security flaws, or even help someone carry out a harmful task.

In this dissertation we study this two-sided relationship between AI language models and software security. The first part looks at how these AI systems can be adapted and properly tested for realistic security tasks, going beyond simple, artificial test problems toward settings that better resemble how security work is actually done in practice. This includes teaching an AI model to explain compiled, "black box" software without access to the original code, testing an AI system's ability to work through a simulated hacking challenge on its own, and checking whether AI coding assistants can be led into helping with harmful requests.

The second part turns the question around, examining the risks that come with using these AI systems themselves. This includes showing that AI models can memorise and later leak fragments of the code they were trained on, including sensitive details that should never have been exposed, and demonstrating that these models can also "remember" and repeat buggy code they encountered during training, even when they would otherwise know better.

Throughout this work, the emphasis lies on testing these AI systems in ways that reflect real-world use, and on openly and responsibly sharing the datasets, tools, and testing methods that were developed, so that other researchers can verify and build on these findings.

With these results, we offer a dual perspective: Large Language Models as tools that can genuinely support security work, and as systems that must themselves be carefully evaluated and secured. With these findings, this dissertation contributes to a more grounded, realistic understanding of both the opportunities and the risks that these AI systems bring to software security.

Samenvatting

Artificiële Intelligentie (AI) die computercode kan lezen en schrijven, de zogeheten Large Language Models (LLMs), worden een vast onderdeel van de manier waarop software wordt gemaakt. Doordat deze systemen goed kunnen redeneren over zowel code als natuurlijke taal, zijn ze niet alleen nuttig voor het dagelijkse programmeerwerk, maar ook relevant voor digitaal beveiligingswerk, waar het draait om begrijpen hoe software zich gedraagt of hoe ze kan worden aangevallen. Tegelijkertijd brengen juist de eigenschappen die deze AI-systemen nuttig maken voor beveiligingswerk hun eigen risico's met zich mee: een model dat is getraind op enorme hoeveelheden code kan per ongeluk vertrouwelijke informatie uit die trainingsdata prijsgeven, code met beveiligingslekken voortbrengen, of zelfs iemand helpen bij het uitvoeren van een hack.

In dit proefschrift onderzoeken we deze tweezijdige relatie tussen AI-taalmodellen en softwarebeveiliging. Het eerste deel kijkt naar hoe deze AI-systemen kunnen worden afgesteld en op een gedegen manier getest voor realistische taken. We vervangen kunstmatige testproblemen met situaties die beter aansluiten bij hoe softwarebeveiliging in de praktijk daadwerkelijk verloopt. Zo leren we een AI-model om gecompileerde, "black box-software uit te leggen zonder toegang tot de oorspronkelijke code, testen we of een AI-systeem zelfstandig een gesimuleerde hackuitdaging kan doorlopen, en gaan we na of AI-programmeerassistenten kunnen worden verleid tot het meewerken aan schadelijke verzoeken.

Het tweede deel draait de vraag om en onderzoekt de risico's die het gebruik van deze AI-systemen zelf met zich meebrengt. Zo laten we zien dat AI-modellen fragmenten van de code waarop ze zijn getraind kunnen onthouden en later prijsgeven, inclusief gevoelige details die nooit naar buiten hadden mogen komen, en tonen we aan dat deze modellen zich ook foutieve code kunnen "herinneren"die ze tijdens de training zijn tegengekomen en die vervolgens herhalen, zelfs wanneer ze het eigenlijk beter zouden moeten weten.

In dit hele werk ligt de nadruk op het testen van deze AI-systemen op manieren die het gebruik in de echte wereld weerspiegelen, en op het open en verantwoord delen van de data, tools en testmethoden die zijn ontwikkeld, zodat andere onderzoekers deze bevindingen kunnen controleren en erop kunnen voortbouwen.

Met deze resultaten bieden we een tweeledig perspectief: LLMs als hulpmiddelen die beveiligingswerk werkelijk kunnen ondersteunen, en als systemen die zelf zorgvuldig moeten worden geëvalueerd en beveiligd. Met deze bevindingen draagt dit proefschrift bij aan een realistisch begrip van zowel de kansen als de risico's die deze AI-systemen met zich meebrengen voor softwarebeveiliging.

Acknowledgments

The work presented in this thesis would not have been possible without the support of many people over the last four years. I can't begin to thank everyone, so I'll try to keep this brief, a skill I have yet to demonstrate anywhere else in this thesis. If you came here looking for your name and didn't find it, I'm sorry. It was definitely in the previous draft.

First, I want to thank my supervisors, Arie and Mali, who supported me unequivocally throughout this journey, even when my plans were more ambitious than the deadlines allowed. I also want to thank the members of SERG, who made every day in the group enjoyable, including the days when my experiments did not.

A special thanks goes to the members of AISE Lab, Jonathan, Roham, Razvan, Ziyou and Daniele, for the many fun discussions we shared, some of which were even about research. Thanks also to Prem and the members of his lab, in particular Toufique, Anand and Claudio. It was always a pleasure to work together, even when one side of the call was clearly still waking up.

Much of the work in this thesis was done in collaboration with talented BSc and MSc students, who, for reasons that remain unclear to me, often volunteered their free time to work on these topics. Many thanks to Fabio, Sebastian, Begüm, Maksim, Maxim, Roland, Artur, Venelina, Remco and Aral, among others. From this point onward, I will refer to their code as 'our implementation'.

I would also like to thank the committee members, who took the time to read my thesis and participate in the defence ceremony. You may be the only people to read it all, and I am grateful for your sacrifice. Thanks as well to the broader Software Engineering community, countless members of which I had the pleasure of meeting at conferences, and whose papers I read, cited and, on my better days, understood.

Outside of work, I want to thank my friends and family, who supported me throughout these years and bore my constant complaining about work, nodding along as though they knew what a reviewer was and why I was so upset with the second one. In particular, I want to thank my parents, to whom I owe everything, as well as Kauther, Muhanned, Huda and Zayn.

Finally, thanks to you. Yes, you, the reader (you know who you are). Thank you to everyone who read and appreciated (or criticised) the borderline incomprehensible ideas and opinions I put on paper every day. That includes you, Reviewer 2.

Ali
Delft, November 2026

1

Introduction

Large language models are increasingly embedded in the software engineering workflow, and their ability to reason about code and natural language makes them equally relevant to software security, where they can support tasks such as program understanding, reverse engineering, and vulnerability analysis. At the same time, the properties that make these models useful for such work also introduce risks of their own: models trained on large code corpora may leak sensitive training data, generate insecure code, or provide harmful technical assistance.

In this thesis we study this dual relationship between LLMs and software security. First, we examine how LLMs can be adapted and rigorously evaluated for realistic security-relevant tasks, moving beyond simplified benchmarks toward settings that reflect practical security work. Second, we examine the security, privacy, and safety risks that arise when code-oriented LLMs are themselves deployed, including memorisation, harmful outputs, and the propagation of flawed coding patterns. Across both directions, we place emphasis on realistic evaluation and the responsible release of datasets, benchmarks, and evaluation pipelines to support reproducible research.

The resulting body of work offers a dual perspective: LLMs as tools that can meaningfully support software-security practice, and LLMs as systems that must themselves be evaluated and secured. With these findings we contribute to a more grounded understanding of both the opportunities and the risks that large language models bring to software security.

1

Large language models (LLMs) are increasingly becoming part of the software engineering workflow [1, 2]. They are used to generate code [3, 4], explain unfamiliar APIs [2], assist with debugging [5], summarise programs [6], and support software maintenance tasks [2]. Their usefulness is not limited to conventional development settings: the same capabilities that allow LLMs to reason about code and natural language also make them relevant to cybersecurity [7]. In this domain, LLMs can assist with program understanding [8], reverse engineering [9], vulnerability analysis [10], and the automation of complex cyber workflows [7].

Security is a particularly high-impact setting for these models because software security work often requires analysts to connect low-level technical evidence with higher-level understanding. A reverse engineer, for example, may need to infer the purpose of a binary function from decompiled code whose identifiers, comments, and original structure have been removed [5]. A security analyst may need to inspect large amounts of source code, trace execution behaviour, or reason about the consequences of a vulnerability [11]. An autonomous cyber agent may need to combine planning, tool use and command-line interaction [12]. These are precisely the kinds of settings in which LLMs appear promising: they can process heterogeneous textual artefacts, generate explanations, and support iterative reasoning over code and system behaviour.

At the same time, the use of LLMs in software engineering introduces security relevant risks of its own. Models trained on large corpora of source code may memorise and reveal sensitive, proprietary, or otherwise unintended training data [13]. Models used as programming assistants may provide harmful technical guidance [14], generate vulnerable code [15], or assist with malicious software engineering tasks [16]. Even when used for benign development, code models may reproduce buggy patterns encountered during training [17], causing exposure to flawed code to influence their later behaviour. These risks are not peripheral to the use of LLMs in software engineering; they arise from the same properties that make the models useful, including their scale, their exposure to large code corpora, and their ability to generate plausible technical artefacts.

This thesis is motivated by this dual role. LLMs are promising tools for security, but they are also systems that must themselves be examined from a security perspective. Treating these two perspectives separately gives an incomplete picture. Capability without risk analysis can lead to unsafe deployment, while risk analysis without understanding capability can miss the practical contexts in which harms emerge. The central argument of this thesis is therefore that LLMs for code and security must be studied through both lenses at once: as tools that can support security-relevant work, and as technical systems that introduce new privacy, safety, security, and reliability concerns.

Thesis hypothesis: Large language models can be adapted to provide useful capabilities in realistic software-security tasks, but these same capabilities are inseparable from security, privacy, safety, and reliability risks; therefore, progress in applying LLMs to software security requires joint evaluation of both capability and risk under realistic, domain-specific conditions.

1.1 Background

In this section we introduce the concepts needed to situate the thesis. We first discuss large language models for code and their role in software engineering, before turning to their use in software-security settings. We then outline the rise of LLM agents and the evaluation challenges they introduce, followed by the risks that arise when code-oriented LLMs are deployed in practice.

1.1.1 Large Language Models for Code

Large language models (LLMs) are neural models trained to model sequences of tokens in large corpora of natural text. Modern LLMs are commonly based on the Transformer architecture, whose self-attention mechanism enables models to learn dependencies across long token sequences [18]. Since code is also highly structured and natural, perhaps even more so than natural language [19], LLMs are also commonly applied to code [20, 21]. When applied to source code, this modelling objective allows LLMs to learn regularities in programming languages, software idioms, API usage, documentation patterns, and the relationship between code and natural language [4, 19, 20, 22].

This has made LLMs increasingly relevant to software engineering. They are now used for code completion, code generation, code summarisation, program repair, debugging assistance, API explanation, and software maintenance [1–4, 6]. These applications rely on the fact that software is not only executable behaviour, but also a human-facing artefact. Source code contains identifiers, comments, documentation, formatting conventions, and architectural cues that help developers understand intent [23]. Code-oriented LLMs exploit these signals to generate plausible completions, explanations, and transformations.

However, software artefacts vary substantially in how much human-readable information they preserve. Source code often contains rich semantic cues, while lower-level representations such as assembly or decompiled code may lose identifiers, comments, type information, and source-level structure [24].

1.1.2 Software Security as an Application Domain

Software security is a high-impact setting for LLMs because many security tasks involve reasoning across multiple levels of abstraction. Analysts often need to interpret partial or noisy technical evidence, relate it to expected program or system behaviour, and use specialised tools to refine their understanding [11]. Activities such as reverse engineering, vulnerability analysis, malware analysis, and incident response therefore depend not only on low-level inspection, but also on the ability to form and revise higher-level explanations of what a system does and how it may fail [5, 11]. LLMs are potentially useful in these settings because they can process heterogeneous textual inputs, generate natural-language explanations, and support reasoning over code, logs, command outputs, and documentation [7, 8, 10].

Binary reverse engineering illustrates this challenge. When source code is unavailable, analysts may rely on disassemblers and decompilers to recover a representation that is closer to source code. Yet decompiled code remains difficult to read: compilation and stripping can remove much of the natural-language channel of software, including comments, variable names, and type information [5, 9]. This motivates the use of LLMs for decompiled program understanding.

1

A second security-relevant setting is interactive cyber problem solving. Penetration-testing exercises, and security-auditing workflows require more than static code understanding. They involve planning, tool use, command-line interaction, observation of intermediate results, and adaptation when a hypothesis fails [7, 12]. In these contexts, LLMs are increasingly embedded in agentic systems that can invoke tools, inspect outputs and act over multiple steps. This shifts the evaluation target from a single model prediction to the behaviour of a model-driven system operating in an environment.

1.1.3 LLM Agents

Large language models are increasingly used as components of agentic systems rather than only as single-turn text or code generators. In an LLM agent, the model is placed in a control loop: it observes the current state, decides on an action, invokes tools, inspects the result, and iterates toward a goal [25–27]. This shifts the model from passive prediction to interactive task execution. Agentic systems typically add scaffolding around the base model, such as tool access, memory, planning prompts, reflection mechanisms, and interfaces to external environments [28].

These abilities are particularly relevant for software engineering and security because many tasks in these domains require sustained interaction rather than a single answer. Debugging, vulnerability analysis, and penetration-testing workflows involve inspecting artefacts, running tools, interpreting outputs, and repeating this process until progress is made [1, 2, 7].

1.1.4 Evaluation of LLM Capabilities

Evaluating LLMs in software security is difficult because many standard benchmarks abstract away the conditions that make security work challenging. Conventional code benchmarks often use clean source-code inputs, isolated tasks and single-turn evaluation [4, 15]. Such benchmarks are useful for measuring general coding capability, but they may not capture the demands of interactive workflows.

Realistic evaluation therefore requires task settings that preserve the structure of the security problem. For software-engineering safety, it means evaluating not only whether a model produces syntactically valid code, but also whether its outputs are secure, reliable, and contextually appropriate [7, 10, 12].

1.1.5 Security Risks of Code-Oriented LLMs

The use of LLMs in software engineering also introduces security risks. Code-oriented models are commonly trained on large corpora of software collected from public repositories and other internet sources [29, 30]. These corpora may contain duplicated code, vulnerable implementations, secrets, licence-restricted material, proprietary fragments, and buggy program versions [13, 17]. Because model training data are often opaque, it can be difficult to determine what a model has seen and how that exposure affects later behaviour [29].

One such risk is memorisation [29, 31, 32]. LLMs may reproduce parts of their training data verbatim or near-verbatim, and prior work has shown that such memorised content can sometimes be extracted through targeted attacks [13, 30]. In the context of code, memorisation is especially concerning because software corpora may contain sensitive

strings, private implementation details or licence-restricted code [29, 33]. Data leakage from code models can therefore create privacy and intellectual-property concerns.

A related reliability risk is that models may reproduce buggy or insecure code patterns encountered during training [33, 34]. If a model has been exposed to both buggy and fixed versions of a program, its later behaviour may be shaped by familiarity rather than correctness. This creates a problem for evaluation as well as deployment: benchmark performance may be influenced by training-data exposure, and generated code may reflect memorised errors rather than robust program understanding [17, 35].

1.1.6 Dual Use, Harmfulness, and Alignment

Software-security knowledge is inherently dual-use. A model that can explain vulnerabilities, automate reverse-engineering steps, or support exploit development may assist defenders, but the same capabilities may also lower the barrier to misuse [14, 16]. A response is *misaligned* when it is not aligned with human *values*, which is different from human preference. In this thesis, we call a response *harmful* when it gives a user actionable support for software-related harm, such as generating malware, bypassing protections, violating software licences, or assisting unfair and dangerous software-engineering practices.

General-purpose alignment criteria, such as helpfulness, honesty, and harmlessness (HHH) [36], therefore require domain-specific interpretation in software engineering [14]. Harmfulness in this domain is not limited to natural-language abuse or general unsafe advice.

1.2 Scope and Core Argument

The thesis is organised around two complementary perspectives. The first perspective, *LLMs for Security*, examines how large language models can support realistic software-security tasks. This includes settings in which models are used to assist analysts, extend existing program-analysis workflows, or operate as components of more autonomous cyber systems. From this perspective, the central question is not merely whether LLMs can solve isolated benchmark problems, but how they can be adapted, evaluated, and understood in settings that resemble practical security work.

The second perspective, *Security of LLMs*, shifts the focus from the use of LLMs as security tools to the security implications of the models themselves. Code-oriented LLMs are trained on large and often opaque corpora of software artefacts, and their behaviour can create privacy, safety, security, and reliability risks. They may memorise and reveal training data, provide harmful assistance, or reproduce buggy and insecure code patterns. These risks are especially important in software engineering, where generated outputs may be incorporated directly into systems and where errors can have significant downstream consequences.

These two perspectives are closely connected. The same capabilities that make LLMs useful for security tasks can also make them risky: a model that can reason about vulnerabilities may assist defenders, but it may also lower the barrier to misuse; a model that has learned from large code corpora may provide useful programming support, but it may also leak or reproduce problematic training data. Meaningful progress therefore requires

1

studying capability and risk together. This thesis treats them not as separate topics, but as mutually informing aspects of the same technical development: the growing role of LLMs in software security.

1.3 Goals and Research Questions

These perspectives lead to several goals that structure the thesis. The first goal is concerned with *capability*: determining how LLMs can be adapted and evaluated for realistic software-security tasks. This includes moving beyond abstract code benchmarks toward settings that better reflect the conditions under which security work is performed, such as decompiled-code analysis, interactive cyber environments, and multi-step agentic workflows. This goal is captured by the first research question.

> **RQ1:** How can LLMs be adapted and rigorously evaluated for realistic software-security tasks?

This question asks how models that are primarily trained on source code and natural language can be extended to security-relevant settings, and how their performance should be measured once they are placed in those settings. The emphasis is on realistic evaluation: rather than assessing models only on simplified or isolated benchmarks, the thesis studies how they behave under conditions that more closely resemble practical security work. This question also supports the trajectory goal by examining how model behaviour changes across task settings, evaluation designs, and model properties.

The second goal is concerned with *risk*: characterising the security, privacy, safety, and reliability concerns introduced by code-oriented LLMs. These risks include the possibility that models provide harmful technical assistance, memorise and reveal training data, or reproduce buggy code patterns. Studying these risks is essential because LLM outputs are increasingly used in practical software-engineering contexts, where unsafe or unreliable behaviour may propagate into real systems.

The second research question focuses on the risks introduced by code-oriented LLMs:

> **RQ2:** What security, privacy, and safety risks arise from deploying code-oriented LLMs?

This question shifts attention from LLMs as instruments for security work to LLMs as systems that themselves require security analysis. Code-oriented LLMs may reveal memorised training data, provide harmful assistance, or reproduce insecure and buggy code patterns. These risks are particularly consequential because generated code and technical advice can be incorporated directly into software systems. The thesis therefore examines not only whether LLMs are useful in software-security contexts, but also what forms of misuse, leakage, and unreliable behaviour must be accounted for when such models are deployed.

The third goal is concerned with *data sharing and cumulative research*. Across the thesis, the empirical studies produce datasets, benchmarks, evaluation scripts, and analysis pipelines that are intended to support reproducibility and follow-on work. This goal is particularly important in LLM and security research, where opaque datasets and inconsistent evaluation practices can make results difficult to interpret or compare [37]. At the same

time, sharing artefacts in this domain requires care: several artefacts involve dual-use security tasks, harmful prompts, extraction methods, or sensitive code data. The thesis therefore treats responsible artefact release as part of the research contribution itself.

Together, these research questions address the two central perspectives of the thesis: RQ1 studies LLMs as tools for software-security work, while RQ2 studies code-oriented LLMs as systems that introduce security, privacy, safety, and reliability risks. The data-sharing goal cuts across both questions by making the resulting datasets, benchmarks, and evaluation pipelines available for reproducible and cumulative research.

1.4 Overarching Methods

In this thesis we use a common empirical methodology across the chapters.

1.4.1 Domain-Specific Dataset and Benchmark Construction

Each chapter begins by constructing or curating a dataset or benchmark environment that reflects the structure of the target security task. We treat dataset construction as a research contribution in its own right, rather than a prerequisite for the experiments.

Where no suitable dataset exists, we build one from scratch. In Chapter 2, we mine and compile CAPYBARA, because no existing benchmark covers binary-level code summarisation at the required scale [4]. In Chapter 5, we construct a code-extraction benchmark, because the existing extraction challenge of Carlini et al. targets natural language and no equivalent code benchmark was available [30, 38].

Where a well-known dataset exists but lacks the evaluation machinery the task demands, we extend it. In Chapter 6, we build an exposure-aware evaluation pipeline around the ManySStuBs4J benchmark [39], augmenting it with membership inference against the Stack-v2 training corpus [40]. In Chapter 3, we curate a set of Capture The Flag challenges and wrap them in a virtualised benchmark harness with checkpoint-based partial-credit scoring.

1.4.2 Model Adaptation and Task-Specific Evaluation

Having constructed a suitable evaluation setting, each chapter adapts one or more LLMs to the task and evaluates them under a set of structured research questions.

In Chapter 2, adaptation is achieved through fine-tuning. We apply the transfer-learning paradigm [20]. The evaluation follows standard code-summarisation practice [41].

In Chapters 3 to 6, adaptation is achieved primarily through prompting rather than weight updates. In Chapter 3, LLMs are placed inside agentic scaffolding with tool access and a reflection-prompting mechanism [25, 27], evaluated inside full virtual-machine environments that isolate each agent run. In Chapter 4, models are evaluated on a dataset of handwritten prompts. In Chapters 5 and 6, models are prompted with prefixes of varying length, and completions are compared against held-out suffixes using exact-match and fuzzy-match metrics.

1.4.3 Comparative Analysis of Models and Techniques

Each chapter evaluates multiple models, configurations, or techniques in order to characterise not just whether a capability or risk exists, but how it varies across model families,

sizes, and adaptation choices. Model sets span open-weight and proprietary models, code-specific and general-purpose models, and different parameter scales, allowing the studies to situate individual results within a broader landscape.

Where comparisons require human judgement, we establish inter-rater reliability before drawing conclusions. In both Chapter 4 and Chapter 3 we use multiple human raters to label model responses. Both chapters use the human-agreement baseline to validate automatic evaluators before scaling up. The human-derived labels were used to train and validate an automatic classifier in Chapter 4 and to validate an LLM judge in Chapter 3. In both cases this was to make large-scale comparison feasible.

1.5 Chapters

In the first part of the thesis we examine *LLMs for Security*. We study how LLMs can be adapted and evaluated for security-relevant tasks that resemble practical workflows. In the second part of the thesis we examine the *Security of LLMs*.

Chapter 2 focuses on binary reverse engineering, where analysts often need to understand programs without access to source code. We show that source-code-pretrained models can be extended to summarise decompiled functions. We contribute the CAPYBARA dataset, the BinT5 model, and an empirical evaluation of binary-function summarisation. In this chapter we demonstrate that LLMs can support program understanding in low-level security workflows.

Chapter 3 moves from static code understanding to interactive problem solving. We use Capture The Flag challenges as a controlled but realistic setting for studying LLM-based cyber agents in virtualised environments. In this chapter we create DeepRed, a benchmark framework with checkpoint-based partial-credit scoring, and evaluate current LLM agents on realistic CTF tasks. The broader contribution is to show that realistic evaluation is necessary for understanding the actual capability boundaries of cyber agents.

Chapter 4 connects capability with risk by studying harmfulness in software engineering. LLMs used for programming can assist legitimate development and security work, but the same technical competence can also support malicious or unsafe tasks. In this chapter we develop a taxonomy of harmful software-engineering scenarios, a prompt dataset, and an automatic evaluator for assessing model behaviour. In this chapter we therefore position harmfulness as the link between the practical utility of LLMs and the risks created by unsafe or misaligned behaviour.

Chapter 5 studies memorisation in code LLMs, focusing on the risk that models reveal sensitive, proprietary, or otherwise unintended training data. We show that code LLMs are vulnerable to extraction attacks and that memorisation increases with model scale. This raises privacy, security, and intellectual-property concerns for models trained on large code corpora.

Chapter 6 extends the memorisation argument from privacy to reliability and software quality. We study whether prior exposure to buggy or fixed code influences model behaviour, and whether models can reproduce memorised bugs. In this chapter we show that exposure-aware evaluation is necessary when assessing code LLMs, because training-data exposure can affect both benchmark results and generated outputs.

Table 1.1: Mapping between the thesis research questions and research chapters.

RQ	Research question	Chapters
RQ1	How can LLMs be adapted and rigorously evaluated for realistic software-security tasks?	Chapters 2 to 4
RQ2	What security, privacy, and safety risks arise from deploying code-oriented LLMs?	Chapters 4 to 6

1.6 Data Sharing and Research Artefacts

Across chapters, we also contribute reusable research artefacts. These include datasets, benchmarks, evaluation scripts, model pipelines, and scoring procedures. Throughout the thesis, we chose to share these artefacts to help other researchers reproduce, compare, and extend the methods presented in this work. This is especially important in LLM research, where opaque training data, closed evaluation pipelines, and inconsistent benchmark practices often limit interpretability and cumulative progress. At the same time, artefact release in this domain must be handled responsibly. Several artefacts involve dual-use security tasks or sensitive data.

To support both accessibility and long-term availability, the artefacts are shared in two complementary forms. First, we provide easily accessible versions through commonly used platforms such as GitHub and Hugging Face, making the resources straightforward to inspect, reuse, and integrate into existing research workflows. Second, we preserve the artefacts through permanent archival services, such as Zenodo or Software Heritage, to ensure that the exact research materials remain available beyond the lifetime of any individual repository or platform. In Table 1.2 we list the artefacts per chapter.

Table 1.2: Research artefacts released for each chapter.

Chapter	Accessible release	Permanent archive
Chapter 2	CAPYBARA; BinT5	CAPYBARA; BinT5
Chapter 3	DeepRed	DeepRed; VMs
Chapter 4	CodeRed; Hammurabi's Code	CodeRed
Chapter 5	Replication package	Replication package
Chapter 6		Replication package

1

1.7 Origin of Chapters

This thesis is presented as a collection of research articles. Chapter 2 to Chapter 6 are self-contained and can be read independently. Each chapter contains its own introduction, background, related work, and conclusion sections. As a result, some overlap between chapters is unavoidable, particularly in the discussion of background concepts, related work, and methodological choices.

The origin of each research chapter is summarised in Table 1.3, the year column denotes when the experiments were conducted, which may differ from the year of publication. Because the models and techniques studied here evolve rapidly, the results reflect the state of the art at the time of the experiments and may be outdated. The author of this thesis is the first author of each article. Chapter 4 and Chapter 3 were written in collaboration with BSc students. For these chapters, the student authors are listed alphabetically and contributed equally.

The chapters have been edited only minimally compared to their published or accepted versions. The main changes concern formatting and presentation: figures and tables have been rerendered to fit the size and style of this thesis, references have been harmonised, and links to online resources have been replaced with permanent snapshots where possible.

Table 1.3: Origin of the research chapters included in this thesis.

Chapter	Year	Full reference
Chapter 2	2022	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)*. 260–271.
Chapter 3	2026	Ali Al-Kaswan, Maksim Plotnikov, Maxim Hájek, Roland Vízner, Arie van Deursen, and Maliheh Izadi. 2026. Do agents dream of root shells? Partial-credit evaluation of LLM agents in Capture The Flag challenges. In *AIWare 2026*.
Chapter 4	2024	Ali Al-Kaswan, Sebastian Deatc, Begüm Koç, Arie van Deursen, and Maliheh Izadi. 2025. Code Red! On the harmfulness of applying off-the-shelf large language models to programming tasks. *Proceedings of the ACM on Software Engineering* 2, (FSE), 2477–2499.
Chapter 5	2023	Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. 2024. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–12.
Chapter 6	2025	Ali Al-Kaswan, Claudio Spiess, Prem Devanbu, Arie van Deursen, and Maliheh Izadi. 2026. Model See, Model Do? Exposure-aware evaluation of bug-vs-fix preference in code LLMs. In *2026 IEEE/ACM International Conference on Mining Software Repositories (MSR)*.

2

Extending Source Code Pre-Trained Language Models to Summarise Decompiled Binaries

Binary reverse engineering *is used to understand and analyse programs for which the source code is unavailable. Decompilers can help, transforming opaque binaries into a more readable source code-like representation. Still, reverse engineering is difficult and costly, involving considerate effort in labelling code with helpful summaries. While the automated summarisation of decompiled code can help reverse engineers understand and analyse binaries, current work mainly focuses on summarising source code, and no suitable dataset exists for this task. In this work, we extend large pre-trained language models of source code to summarise decompiled binary functions. Furthermore, we investigate the impact of input and data properties on the performance of such models. Our approach consists of two main components; the data and the model. We first build CAPYBARA, a dataset of 214K decompiled function-documentation pairs across various compiler optimisations. We extend CAPYBARA further by removing identifiers, and deduplicating the data. Next, we fine-tune the CodeT5 base model with CAPYBARA to create BinT5. BinT5 achieves the state-of-the-art BLEU-4 score of 60.83, 58.82 and, 44.21 for summarising source, decompiled, and obfuscated decompiled code, respectively. This indicates that these models can be extended to decompiled binaries successfully. Finally, we found that the performance of BinT5 is not heavily dependent on the dataset size and compiler optimisation level. We recommend future research to further investigate transferring knowledge when working with less expressive input formats such as stripped binaries.*

2.1 Introduction

Reverse engineering binary programs has many applications, in particular, software security [11]. Binary reverse engineering is a hard task, requiring highly skilled reverse engineers [11, 42]. Disassemblers and decompilers can help in this process. Disassemblers transform the binary into a low-level intermediate representation, and decompilers lift the representation to a high-level programming language-like representation. But the output of decompilers is still difficult to read and understand [11, 43]. Much of the work that goes into reverse engineering a binary is spent labelling functions with semantic descriptions [11]. Current approaches [5, 44–49] mainly focus on recovering aspects lost in the compilation and decompilation process, such as names and types. Existing works fail to address the inherent difficulties in binary code comprehensibility, namely, the need for a high-level overview of the code.

For source code, methods exist to automatically generate summaries from code [6, 50]. Source code summarisation is used to automatically generate short natural language descriptions of code, which support program comprehension and aid maintenance [6, 41]. While these methods have been successfully applied to programming languages such as Python, Java and PHP [20, 22, 51], using pre-trained language models [20, 22, 51], none of these methods has been applied to the relatively syntactically-poor output of decompilers (see Figures 2.1a and 2.1b). Being able to quickly determine the context and application of a function, can save valuable analysis time, and greatly benefit reverse engineers. Function and variable names alone, are inadequate representations of the source code [6], which is why having descriptive summaries of binaries is desirable.

Following [52], source code can be described as having two information channels: the algorithmic channel and the natural language channel. The algorithmic channel specifies the execution of a program (semantics), while the natural language channel explains the purpose and context of the program to humans [52]. The natural channel includes function and variable names, code comments and the specific human-readable structure of programs. Processors only consider the algorithmic channel to execute a program, while humans use both the algorithmic channel and the natural channel to understand a piece of code [52]. Furthermore, code is very regular and predictable, even more so than natural languages [19]. The compilation process, which transforms readable code into executable binaries, removes much of the information contained in the natural channel. Especially stripped binaries — binaries of which the symbol table is removed — are challenging, since they have almost no identifiers at all as can be observed in Figure 2.1c.

The goal of this chapter is to advance the field of binary reverse engineering by exploring the application of code summarisation to decompiled binaries by taking advantage of source code pre-trained language models.

However, there exists no dataset of aligned binaries and source code summaries since this is a new and unexplored task. As pointed out by LeClair and McMillan, the lack of standardised datasets is a major barrier to ongoing research, which we will address for this task [53]. In this chapter, we create a dataset containing pairs of decompiled and stripped-decompiled functions and summaries of these functions. During the creation of this dataset, we conform to the current best practices for dataset construction [53, 54].

We apply this dataset to an existing pre-trained language model using transfer learning, by fine-tuning this pre-trained model on our dataset. For this task, we selected a

pre-trained CodeT5 model, which was only trained on source code [20].

We perform experiments on this model to explore the impact of decompilation, and the importance of identifiers. Furthermore, we explore the impact of compiler optimisation levels, the dataset size and the level of duplication.

Our findings are that the decompilation and alignment of stripped functions has a very high failure rate; and the resulting stripped model has low performance. But, we found that the model shows state-of-the-art performance with both decompiled code as well as demi-stripped code, code of which the identifiers were removed after decompilation. Our experiments on data duplication and dataset size further show that these models can be trained with few data, and that while duplicates have a high impact on performance, their presence is not paramount to model performance.

Our key result: *language models pre-trained on source code can be fine-tuned on binaries, opening up a range of new possibilities for the automated analysis of binaries.*

To summarise, the main contributions of this chapter are:

- CAPYBARA[1], a dataset of **C***ombined* **A***ligned decom***P***iled* **B***inar***Y** *code* **A***nd* **R***elated* **A***nnotations*. A novel dataset of aligned, C, decompiled, stripped-decompiled and demi-stripped summary pairs[2] (Section 2.3);
- BinT5[3], a **Bin**ary summarisation Code**T5** model, a simple and straightforward adaptation of a source code trained code summarisation model to decompiled code using CAPYBARA (Section 2.4);
- An empirical investigation on the impact of the properties of decompiled code and the properties of CAPYBARA (Sections 2.5 and 2.6);

The materials, including the processed and raw data, the trained model checkpoints and steps to replicate our experiments, are openly available in our replication package[4].

2.2 Background

In this section, we introduce the background of compilers, binary reverse engineering, transfer learning and the code summarisation task.

2.2.1 Compilers and Optimisation Levels

Compilers are programs that convert source code from one programming language to another, but generally, and in the context of this work, the term is used to refer to programs that translate high-level code, like C, to a lower-level language such as machine code or bytecode. For our work, we focus on the GNU Compiler Collection (GCC)[5] and Clang/LLVM (Clang)[6].

[1]CAPYBARA: https://doi.org/10.5281/zenodo.7229809
[2]Decompiled code with strip-like obfuscation applied
[3]BinT5: https://doi.org/10.5281/zenodo.7229913
[4]Replication package: https://archive.softwareheritage.org/swh:1:snp:f4f0a12fb7df9dc07ca3e14e7fefad244d3e2683
[5]GCC: https://web.archive.org/web/20220630142950/https://gcc.gnu.org/
[6]Clang: https://web.archive.org/web/20220706000729/https://clang.llvm.org/

Compilers feature optimisation levels. Generally, the goal of optimisations is the improvement of runtime performance or program size at the expense of compilation time and the ability to debug [55].

By default, if GCC is invoked without any optimisation options, the program will be compiled with -O0. -O1, -O2 and -O3 incrementally apply more optimisation to the binary at the expense of a higher compilation time [56]. Optimisations can restructure and transform the program in relation to the source code, by changing the control flow or the data of the program [57]. This obfuscation can complicate the reverse engineering process by reducing the accuracy of tools [57].

```
/**
 * Get the Synchronizing source for an RTP/RTCP Socket
 * @param rs RTP Socket
 * @return Synchronizing source
 */
uint32_t rtp_sess_ssrc(const struct rtp_sock *rs){
    return rs ? rs -> enc.ssrc : 0;}
```

(a) Source rtp_sess_ssrc function

```
ulong rtp_sess_ssrc(long param_1){
    uint local_14 ;
    if (param_1 == 0){
        local_14 = 0;
    } else {
        local_14 = * (uint *) (param_1 + 4);}
    return (ulong) local_14;
}
```

(b) Decompiled rtp_sess_ssrc function

```
ulong FUN_00100d30 ( long param_1 ){
    uint local_14 ;
    if (param_1 == 0) {
        local_14 = 0 ;
    } else {
        local_14 = * (uint *) (param_1 + 4);}
    return ( ulong ) local_14 ;}
```

(c) Stripped decompiled rtp_sess_ssrc function

Figure 2.1: Example source, decompiled and stripped code snippet

2.2.2 Ghidra

Ghidra[7] is a free and open-source reverse engineering toolkit developed by the US National Security Agency. Ghidra contains many separate analysis modules that allow a reverse engineer to analyse binaries. Ghidra features a disassembler, which disassembles binaries back into an intermediate representation. In the case of x86-x64 binaries like the binaries this work focuses on, the intermediate representation will be the Assembly language. The decompiler, on the other hand, is a processor language-agnostic transformation engine that takes the disassembled code and creates a source code representation, namely pseudo-C. Pseudo-C follows the general language conventions of C, but it cannot be compiled.

Observe the relatively simple `rtp_sess_ssrc` function from `creytiv/re`[8] shown in Figure 2.1a. We compile the project using the -O3 compiler level as defined in the project. We decompile the binaries using Ghidra's decompiler using the standard configuration, the resulting pseudo-code is shown in Figure 2.1b. We observe that aside from the function name, almost the entire natural channel has been destroyed by the compilation and decompilation process. The parameter and variable names are gone, any documentation is removed and the relatively simple logic has been unrolled to a much more difficult-to-understand representation. Ghidra also incorrectly labelled many of the variable types and failed to identify the `struct` datatype.

Using our trained BinT5 model we can summarise the decompiled code and generate the following summary: `Get the source for an RTP/RTCP Socket`. This summary gives us an indication of the purpose of the function. Integrating this generated summary into Ghidra increases the readability of the entire binary. Keep in mind that a reverse engineer has to understand not just this function, but hundreds of different functions in a single binary.

2.2.3 Stripping

Aside from compiling with higher optimisation levels, binaries can also be stripped to obfuscate the underlying code and to resist analysis [58]. Commercial off-the-shelf software is often stripped to reduce the memory and storage footprint of the binaries, and to resist analysis to protect the intellectual property of the creator. Many vulnerable and malicious binaries are, unfortunately, also stripped to resist security analysis and hide their faults [5].

Unix and Unix-like operating systems include a strip utility. The strip utility removes any operands that are not necessary for the execution of the binary while ensuring that the execution of the binary remains unchanged. The exact implementation and what constitutes unnecessary operands are left to the implementor.[9] The strip utility as implemented in GNU/Linux removes the symbol table from the binary. The symbol table contains each symbol's location, type and name.

[7] Ghidra: https://web.archive.org/web/20220601075714/https://github.com/NationalSecurityAgency/ghidra

[8] re: https://archive.softwareheritage.org/swh:1:rev:1f928f6399068233de546678da3815c39367cd72

[9] strip: https://web.archive.org/web/20220519124845/https://pubs.opengroup.org/onlinepubs/7908799/xcu/strip.html

Like higher optimisation levels, the use of stripping can greatly complicate the efforts to reverse engineer a binary, as well as reduce the accuracy and effectiveness of reverse engineering tools [58].

For example, we compile, strip and decompile the function in Figure 2.1a, and the resulting stripped decompiled function is shown in Figure 2.1c. In addition to the details lost by the decompilation process, the stripper removed all symbols, like the function names.

2.2.4 Code Summarisation Task

Code summarisation (also referred to as source code summarisation) is the task of writing short descriptions from source code, usually a single-sentence summary of the source code. The main use is for software documentation, like the one-sentence JavaDoc description used in Java [53]. This documentation is important for program comprehension and maintenance. But the process of writing and maintaining these descriptions is a labour-intensive and time-consuming task, which is where the benefits of automating that process arise. Automatic code summarisation is an active and popular research problem in the field of software engineering [53].

2.2.5 Transformer-based Models

Transformers were originally proposed by Vaswani et al. as a sequence-to-sequence architecture [59]. Unlike the Recurrent Neural Networks [60] (RNN), the Long Short-Term Memory [61] (LSTM) variant of RNNs [60] and Convolutional Neural Networks [62] (CNN), Transformers only use a mechanism called self-attention to capture dependencies between the input and output. The current state-of-the-art NLP models for programming languages such as CodeT5 [20], CodeBERT [22] and PolyGlotCodeBERT [51] are all based on the Transformer architecture [59].

2.2.6 Transfer Learning

Pre-trained Transformers-based language models, such as RoBERTa [63], CodeBERT [22] and CodeT5 [20] utilise a pre-train then fine-tune paradigm. The bespoke paradigm was initially introduced by Kenton and Toutanova. In this paradigm, the models are first trained in an unsupervised manner on a large unlabelled dataset. These pre-trained models can then be fine-tuned to perform a more specialised task, such as summarisation. Transfer learning uses the knowledge that is obtained in one task to solve a different task. It allows the creation of general models that are trained once on massive datasets. These general models, which contain general domain knowledge can then be fine-tuned for a specific downstream task. This approach is quicker and requires less training data than training a model on the downstream task from scratch [64].

2.3 CAPYBARA Dataset

We require a dataset of decompiled functions labelled with a descriptive summary to create and assess our solution. This dataset should be relatively large to suit the 'data-hungry' nature of deep-learning models. Furthermore, the dataset needs to feature a diverse set of data representative of our solution's actual real-life use case.

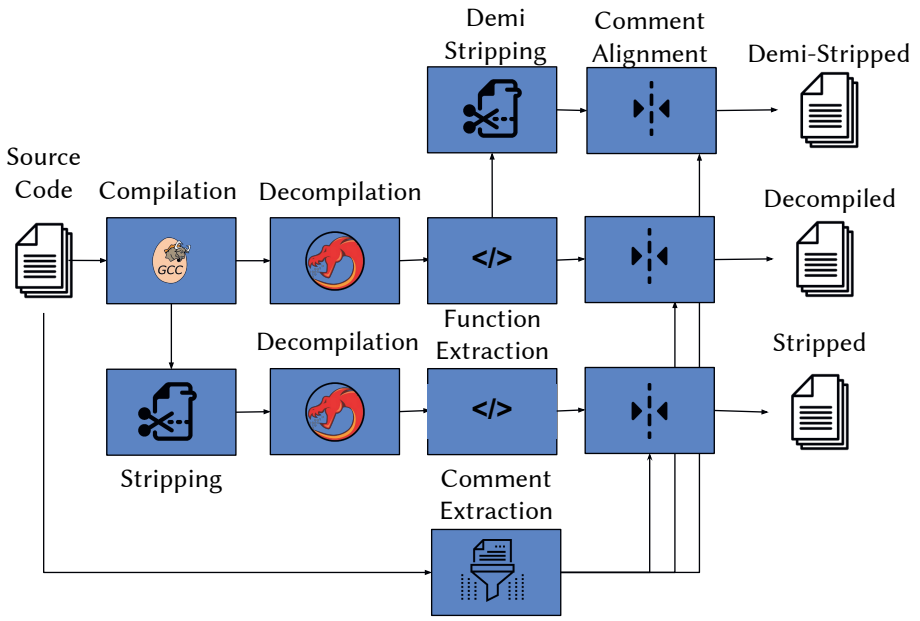


Figure 2.2: Data collection pipeline

2.3.1 Data Collection

To create such a large and diverse dataset we made use of BinSwarm [46], an existing dataset of aligned decompiled and stripped decompiled functions[10]. BinSwarm collects C-based projects from Github. The projects are filtered to only include those that are actively being developed, using Travis CI and built for Ubuntu Linux. The projects are built using Docker. The resulting binaries are then copied and stripped, and both the stripped and unstripped binaries are decompiled using Ghidra. The functions are extracted from the stripped and unstripped decompiled code and aligned with the source code. The BinSwarm dataset only contains aligned tuples of source code and (stripped-) decompiled functions. We extract documentation from the original source code files to add descriptive comments to this dataset. To that end, we depend on the documentation included in the source code by the original authors in the form of single and multiline comments. We locate the functions in the unbuilt project files and align the decompiled functions with the comments in the source code using srcML[11] to extract any documentation located directly before a function signature. A high-level overview of the entire process is shown in Figure 2.2.

A function's documentation often also contains other details besides the descriptive summary. We found that C projects do not follow a single documentation standard. For example, Javadoc for Java has a short one-line description or summary for each method at the beginning of the multiline comment block. In C, there is no singular documentation

[10]BinSwarm: `https://hub.docker.com/r/binswarm/cbuilds`
[11]srcML: `https://web.archive.org/web/20220618232913/https://www.srcml.org/`

```
/** @brief Select the source of Microcontroller Clock Output
 * Exact sources available depend on your target.
 * On devices with multiple MCO pins, this function controls MCO1
 * @param[in] mcosrc the unshifted source bits
 */
```

Figure 2.3: Example of documentation from jeanthom/ DirtyJTAG: rcc_set_mco

standard, so there might not be a single-line summary, and we will need to locate it in the comment block automatically.

Summary Extraction Rules We observe that the majority of single-line data are descriptive summaries, so we extract the first sentence. We identify many documentation styles in our multi-line data, we define some automated rules to extract summaries from the documentation:

- **@brief or @purpose:** If the documentation contains a '@brief' or '@purpose' tag, we extract the first sentence after the tag. The 'brief' tag is part of the Doxygen documentation standard[12], an example is shown in Figure 2.3[13].
- **Description:** If the documentation contains a line with 'Description:', we extract the following sentence.
- **@param or @v:** Documentation that contains an '@v' or '@param' tag, usually has a summary in the sentence before the tag. We extract that sentence.

Filtering Rules To improve the quality of the dataset we filter out samples based on the rules used by the CodeSearchNet dataset [54] included in the CodeXGlue benchmark for the summarisation task [4]:

- **Documentation length:** We remove any summaries that are too long or too short and remove anything shorter than 3 or longer than 256 tokens.
- **Special tokens:** We follow the example of the CodeSearchNet [54] and remove all documentation that contains special tokens. We scan for web tokens (like 'http://'), HTML tokens (like '<head>'), paths (like 'C://Users/..'), since this documentation usually refers to external resources. We additionally filter any developer tokens (like 'FIXME:'), as these documents do not provide meaningful information about the function itself, but contain comments about the development process.

[12] Doxygen:https://web.archive.org/web/20220603032119/https://www.doxygen.nl/manual/docblocks.html

[13] jeanthom/DirtyJTAG:rcc_set_mco:https://archive.softwareheritage.org/swh:1:cnt:bdd7dd5722520ce3a907dbc0230aa136c8fd80bf;origin=https://gitlab.com/insane-adding-machines/unicore-mx.git;visit=swh:1:snp:74b478b2a8b7c0061880a4f128040caa154b27c4;anchor=swh:1:rev:c80139a036466c3eab4ab8eca48f26b3a92333c1;path=/lib/stm32/common/rcc_common_all.c;lines=192

- **Language:** We filter out any documentation that was not written in English using the FastText language identification algorithm [65]. Around 92.19% of the documentation is in English.
- **Empty documentation:** We find that a large number of functions did not have any documentation associated with them at all. We simply remove these samples from the dataset.
- **Abstract Syntax Tree:** The authors of the CodeSearchNet dataset [54] additionally, remove any samples that do not parse into an AST. We choose to omit this step since all of our samples have been successfully compiled and have thus at one point been parsed into an AST by the compiler.

2.3.2 Dataset Preparation

Synthesis of Demi-stripped Code From the dataset of decompiled functions, we also create another dataset. We emulate the process of stripping by removing all the identifiers from the decompiled code and replacing them with placeholders. For clarity, we call this demi-stripped data. Like the stripped dataset, the identifiers are all removed, but this is only done after the decompilation process. The decompiler still had access to the identifiers and could use the symbol table during decompilation. Most importantly, this demi-stripped dataset still has the same structure and control flow as the unstripped decompiled dataset and avoids any decompilation issues arising from stripping.

Data Split The dataset is split into a train, test and validation set. These sets constitute approximately, 80%, 10% and 10% [53] of the complete dataset. As recommended by Shi et al. and LeClair and McMillan, we prevent leakage of vocabulary and code patterns between the sets, by sampling the sets in a cross-project manner [41, 53]. This means that an entire project gets assigned to one of the sets, and functions from the same project cannot be assigned to different sets. The projects in the test and validation set are the same across all datasets.

Duplication Large corpora of code, like the corpus gathered by BinSwarm, tend to have a high degree of duplication [53]. As a result, snippets of code that are relatively unchanged appear in multiple parts of the corpus. This can be in the form of copied, generic or auto-generated functions. These functions will appear in multiple repositories and might be duplicated across the training and testing data. Besides exact duplicates, near-duplicates can also occur. Near-duplicates differ in a few minor aspects like additional code comments or different function names. While removing exact duplicates is relatively fast and straightforward, removing near-duplicates is much more challenging and computationally intensive [66]. The issue with code duplication in classical code summarisation is that the models and tools are supposed to be used to generate summaries for new and unseen code. The evaluation metrics should therefore measure the generalisation of the tool on new samples [66]. Duplicates and near-duplicates are not defined as new samples. A user of such a tool could simply look these samples up. Furthermore, large, high-capacity models like CodeT5 with 220M [20] or CodeBERT with 128M [22] parameters, have a large capacity to memorise duplicated code [66].

However, the use case outlined in this work is more akin to deobfuscation. As explained by Allamanis, deobfuscation could be a use case where duplicates are valid and part of the true distribution of the problem [66]. Compiled code contains a lot of duplicate code, and understanding this code is still difficult and essential for understanding the binary. While regular source code allows the reader to look up code snippets, decompiled binaries have an additional obfuscation applied. We, therefore, focus on the model's performance on code with duplicates as we believe duplicates to be part of the true distribution of the data, but we also report the deduplicated results.

2.3.3 Dataset Properties

Table 2.1 shows the size of the processed dataset. Of the 2.1M aligned decompiled functions, we extract documentation for 215k of them, and we found that the majority of samples, 1.5M did not have any documentation at all. Furthermore, BinSwarm only provided us with 415k aligned stripped samples, and we can extract documentation for only 14k of these samples.

Dataset	Including duplicates	Deduplicated
C/Demi/Decom	214,587	79,673
Stripped	14,245	7,826

Table 2.1: Number of functions in dataset

The vast majority of documentation is in the form of multi-line comments as opposed to single-line or double-slash comments. We found that the documentation and comments had a mean length of 42.60 and 8.14 tokens, respectively.

Figure 2.4 shows the distribution of the number of tokens in source code and decompiled code. The source and decompiled code have a mean length of 399 and 779 tokens, respectively. Decompiled code also has close to double the LOC of source code, with means of 30.77 and 53.42 lines for source and decompiled, respectively.

The majority of decompiled functions are compiled with optimisation level -O2, with a similar number of -O1 and -O3 samples and relatively few -O0 samples. Stripped data has a very even distribution of optimisation levels, with only -O0 having significantly fewer samples. Note that there are more optimisation levels than shown in Figure 2.5, for brevity the different levels are grouped into their base optimisation level. -Oa is grouped with -O0, -Of and -Og are grouped with -O1, -Os is grouped with -O2. We also observe some samples with an optimisation level higher than -O3 (-O8 and -O7), as specified by the GCC documentation, these levels are equivalent to -O3[14].

[14]GCC optimisation levels: https://web.archive.org/web/20210418210849/https://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Optimize-Options.html

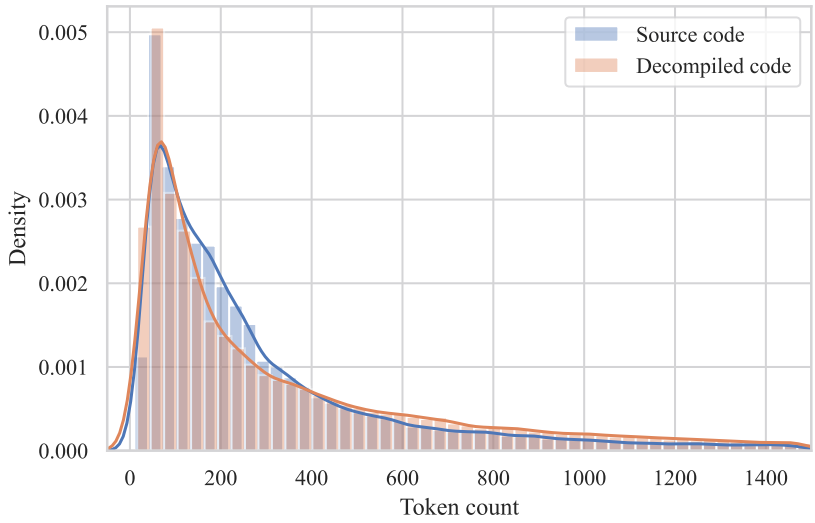


Figure 2.4: Tokens in source C and decompiled code

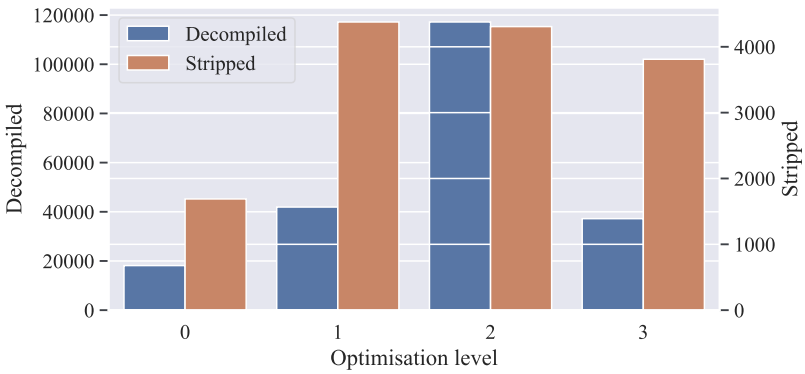


Figure 2.5: Distribution of optimisation levels in decompiled (left) and stripped (right)

2.4 BinT5

We select CodeT5 [20] as the base-model for our experiments since it is the highest-scoring publicly-available model on the CodeXGLUE [4] Code Summarisation benchmark[15]. CodeT5 is a programming language model built on the T5 (Text-to-text Transfer Transformer) architecture [67] and pre-trained on a mix of supervised and unsupervised tasks. CodeT5 employs an encoder-decoder architecture. In contrast to other models, CodeT5 is trained using both unimodal (PL only) and bimodal (NL-to-PL) tasks in eight programming languages. This bimodal training allows CodeT5 to perform strong cross-modal tasks such as code summarisation and code generation (PL-to-NL). Many other models only use the data and languages included in the CodeXGlue dataset [4, 22, 51], while CodeT5 also uses a mined dataset of C and C++ code for its pre-training objectives [20]. The inclusion of C training data should help the model with the CAPYBARA dataset. There could be some overlap in the training data between CAPYBARA and the dataset used by Wang et al. which would cause leakage, we address these concerns in Section 2.7.

CodeT5 also utilises the transfer learning paradigm, which allows us to train the model with relatively little data. In this case, we make use of the CodeT5-base model, which was trained on mixed upstream tasks by the authors [20]. We fine-tune this model on the code summarization task on CAPYBARA. An overview of how we applied the model to create BinT5 is provided in Figure 2.6.

[15] CodeXGLUE benchmark: https://web.archive.org/web/20220528144842/https://microsoft.github.io/CodeXGLUE/

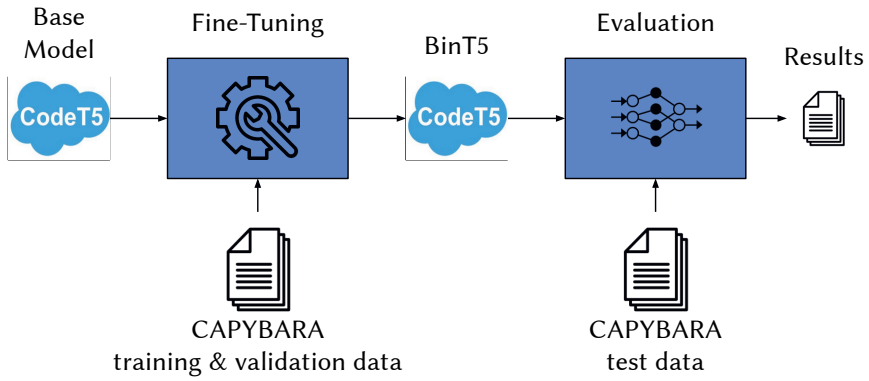


Figure 2.6: BinT5 fine-tuning pipeline

2.5 Experimental Setup

To assess the effectiveness of our approach, we first evaluate the performance of the model, we then identify the aspects of the data that make this task inherently difficult, and we finally investigate aspects of the datasets and their influence on the complexity of the task.

2.5.1 Research Questions

In the context of the study, we thereby formulate the Research Questions (RQ) as follows.

RQ1: *How effective are fine-tuned Transformer-based models at decompiled code summarisation?* To investigate the application of existing models to binaries using CAPYBARA, we set a baseline by training a model on the code summarisation task on the source C-code dataset. We then train a summarisation model on both the decompiled and the stripped dataset. We use the evaluation metrics to compare the performance of the different models.

RQ2: *Which aspects of the input contribute most to model performance?* We investigate which aspects of decompiled code increase the difficulty of the task. We, therefore, look at the impact of the symbol table on decompilation, for this, we fine-tune a model on the demi-stripped dataset and compare it to the other models. We also investigate the importance of the function name by removing *just* the function name from the decompiled code. Furthermore, we investigate the impact of the optimisation level by exploring the performance per optimisation level.

RQ3: *What is the impact of dataset properties on model performance?* We finally investigate how the construction of CAPYBARA influences the models. To answer the final research question we remove the duplicates from the datasets and retrain the models, after which we compare the performance to the baselines. Furthermore, we investigate the impact of dataset size, by incrementally reducing the size of the training sets.

2.5.2 Baselines

To first establish a performance baseline, we train a CodeT5-base model on the summarisation task on source C. Note that only samples which are aligned with decompiled code are included in the source C dataset. The baseline is used to compare the decompiled C, stripped decompiled C and the demi-stripped datasets to the source code.

2.5.3 Evaluation Metrics

We evaluate the performance between the reference summary from CAPYBARA and the candidate summary produced by BinT5 using the EM, BLEU-4 [68], ROUGE-L [69] and, METEOR [70] metrics.

Exact Match (EM) The simplest metric is the EM which scores a prediction one if it matches its reference exactly and zero otherwise.

BLEU-4 The most widely used metric in the code summarisation task is the Bilingual Evaluation Understudy Score (BLEU) [41]. BLEU-4 produces a percentage number between 0 and 100, which defines the similarity between a candidate and a set of reference sentences. BLEU-4 calculates the cumulative 4-gram precision scores, the number of matching 4-grams divided by the total number of 4-grams in the candidate sentence [68]. The unigrams and bigrams account for the adequacy of the candidate while the longer three and 4-grams account for fluency. To prevent short sentences the result is multiplied by a brevity penalty as well. A smoothing function is applied to prevent sequences with no matching 4-grams to score zero [71]. While Shi et al. recommend BLEU-4 with smoothing method 4 [41], we opted to use the Moses [72] implementation of BLEU-4 which uses smoothing method 2 since this is also utilised by CodeSearchNet, CodeXGlue and CodeT5 [4, 20, 54].

ROUGE-L ROUGE or Recall-Oriented Understudy for Gisting Evaluation, is a package which includes several metrics, the most popular among them is ROUGE-L [69]. ROUGE-L is more recall oriented than BLEU-4. ROUGE-L simply finds the longest common subsequence (LCS) between the reference and the candidate. Note that the words do not need to be consecutive but they have to be in order.

METEOR METEOR or Metric for Evaluation for Translation with Explicit Ordering [70] uses word lists and stemming to also take synonyms into account and calculates the harmonic mean of the unigram precision and recall. Similar to ROUGE-L, METEOR is more recall-focused. METEOR has a higher correlation with human judgement than BLEU-4 [53] at the sentence level.

2.5.4 Data deduplication

To create a deduplicated version of the CAPYBARA dataset we make use of a fork[16] of the near-duplicate-code-detector [66]. We use this tool to compare all the datasets' functions and find clusters of near-duplicate functions. We randomly select one function per

[16]Near Duplicate Code Detector: `https://archive.softwareheritage.org/swh:1:snp:352edebecbc9b9bfe631bf2e86052442fff65ba3`

cluster and discard the rest from the dataset. We use the standard tool configuration as recommended by Allamanis. Of the removed duplicates, we observe that a relatively large number originates from common libraries, such as SQLite[17], that are packaged with binary programs. Thus a certain amount of duplication is also likely to occur "in the wild".

2.5.5 Configuration

We process and visualise the data with Pandas 1.4.3 and Ghidra 10.0.4[18]. FastText 1.0.3 with the largest lid.176.bin model is used to detect languages. We train the model using Transformers version 4.16.2 running on Torch 1.9.0+cu111 in the nvidia/cuda:11.4.0-base docker container image. We share a Docker image with all the libraries required to run BinT5 pre-installed on DockerHub[19].

A grid search of the optimal settings was infeasible from a time perspective, so we performed training mainly using the recommended settings from the CodeT5-base model [20]. We double the source length for the decompiled, stripped, and demi-stripped code to 512 tokens instead of the standard 256 tokens used for the source code to compensate for the fact that the average length of decompiled code is almost twice as long as the source code. We trained the model on a machine with an NVIDIA GeForce RTX3080 with 10GB of VRAM and an AMD Ryzen Threadripper 3990X 64-Core Processor with 192GB of RAM running Ubuntu 20.04.4 LTS. The GPU is running Nvidia driver version 510.60.02 with Cuda 11.6. The authors of CodeT5 used an NVIDIA A100 GPU with 40GB of VRAM for fine-tuning [20]. To compensate for the lack of memory, we reduced the batch size to 2, which was the maximum length that could still fit in the VRAM, we increase the 'gradient_accumulation_steps' to 24 to still achieve the effective standard batch size of 48.

2.6 Results

We present the results of our experiments to answer the research questions, results are grouped per research question. The metrics are calculated for each sample from the test set, and the average scores are presented.

2.6.1 RQ1: Model Effectiveness

The performance of the CodeT5-base model on each of the datasets is presented in table 2.2.

[17]SQLite: `https://web.archive.org/web/20220605031613/https://www.sqlite.org/index.html`

[18]It is not recommended to use Ghidra versions before 10.1 since these versions have not been patched against a Log4J RCE

[19]BinT5 Docker Image: `https://hub.docker.com/r/aalkaswan/bint5/tags`

	BLEU-4	EM	METEOR	ROUGE-L
C	60.83	52.19	65.33	66.51
DecomC	58.82	48.92	63.14	64.51
Stripped	11.26	1.85	14.50	17.25

Table 2.2: Result of fine-tuning CodeT5-base on mined datasets

	BLEU-4	EM	METEOR	ROUGE-L
DecomC	58.82	48.92	58.4	60.32
Demi	44.21	35.10	47.89	49.59
NoFunName	46.99	37.12	45.92	48.07

Table 2.3: Result of fine-tuning CodeT5-base on synthetic data

We found that the decompiled code model generally produced good summaries, evidenced by the BLEU-4 score of 58.82, which is slightly lower than the baseline set by the source code. The stripped model mainly produced unusable summaries, as evidenced by the BLEU-4 score of 11. The high EM score could be an indication of a high duplication factor.

Initial experiments with GraphCodeBERT [73] and PolyglotGraphCodeBERT [51] base models fine-tuned on CAPYBARA show performance around 5 and 3 BLEU-4 lower, respectively. This is a relatively small difference, especially considering the model size. This shows that the performance of BinT5 does not heavily depend on the additional pre-training on C and C# performed by Wang et al.. Furthermore, this result shows that it is improbable that significant dataset leakage has taken place.

We found a relatively large difference between the number of recovered decompiled and stripped decompiled functions. This can likely be attributed to the fact that Ghidra struggles a lot more with recovering stripped functions. Recall that the symbol table commonly contains information regarding the location and name of functions. When this table is dropped, the start- and endpoints of functions are hard to infer by automatic tools, especially since many functions get inlined, and JUMP instructions replace CALL instructions. Aside from difficulties in demarcating functions, it is also difficult to align the associated source code function with the decompiled function. With unstripped code, the function name remains, meaning the functions can be aligned using the name. We attempted to utilise an existing solution by Alves-Foss and Song called Jima [74] to find function boundaries. Jima is the current state-of-the-art tool for function boundary detection in stripped binaries. The tool is implemented as a plugin for Ghidra, but in our experiments, we find no statistical difference between the base performance of Ghidra and Jima on our dataset. The difficulties in extracting stripped functions, make training and applying a model to stripped binaries challenging.

2.6.2 RQ2: Input Properties

As can be observed in Table 2.3, the summaries produced by the demi-stripped model were substantially worse than the decompiled model, but most were still very usable, evident by the BLEU-4 score above 44. Just removing the function name gave quite similar results to demi-stripping. We find that the loss of identifiers significantly lowers the performance of the model, but stripped code also suffers from decompilation faults, which seem to have a much larger impact on the model performance. Hence, the performance of BinT5 on demi-stripped code can be viewed as more representative of the actual model and not impacted by faults introduced by Ghidra.

Opt level	BLEU-4	EM	METEOR	ROUGE-L
-O0	72.88	34.18	73.19	74.84
-O1	50.30	59.84	55.36	54.84
-O2	62.31	46.23	64.50	66.05
-O3	54.68	54.99	58.25	59.28

Table 2.4: Average BLEU-4 score of decompiled code per optimisation level

Table 2.4 shows the average score per optimisation level. We can observe that -O0 and -O2 perform better than -O1 and -O3. Recall that -O0 is completely unoptimised, and that the vast majority of our decompiled dataset is compiled with -O2, which would explain why those optimisation levels perform better.

2.6.3 RQ3: Dataset Properties

The performance of the base model on each of the deduplicated datasets is presented in table 2.5:

	BLEU-4	EM	METEOR	ROUGE-L	ΔBLEU-4
C	45.86	32.87	46.06	47.53	14.97
DecomC	42.48	28.08	25.23	27.66	16.34
Demi	25.38	14.51	42.47	44.47	18.83
Stripped	7.19	0.00	4.75	5.50	4.07

Table 2.5: Result of fine-tuning CodeT5-base on the deduplicated datasets and the difference with the baseline

We find that the influence of deduplication on our model's performance is relatively small on source code, at only 24%. Duplicates have a relatively large impact on the decompiled (28%) and demi-stripped (43%) code. Deduplication also greatly decreases the EM rate across the board. Duplicates have a relatively large impact on performance, but even with the duplicates removed the model still produces many high-quality summaries. The experiments on deduplication show that the model seems to have a deeper understanding of the data and is not simply reproducing previously seen samples.

As can be seen in Figure 2.7, the dataset size does not have much of an impact, the model can be trained with half or a quarter of the training samples without suffering a considerable hit to performance. This could be attributed to the high duplication factor of our dataset. It could also be because the model was already pre-trained well by Wang et al. and requires very little data for fine-tuning. This is a testament to the relative ease with which these models could be extended to decompiled code.

We also performed experiments where we did not apply the filtering rules provided by CodeXGlue and where we always mined the first sentence of any type of documentation. While we were able to collect around 480K decompiled samples, the model performed substantially worse, only scoring 36.97 and 33.26 BLEU-4 on C and decompiled code, re-

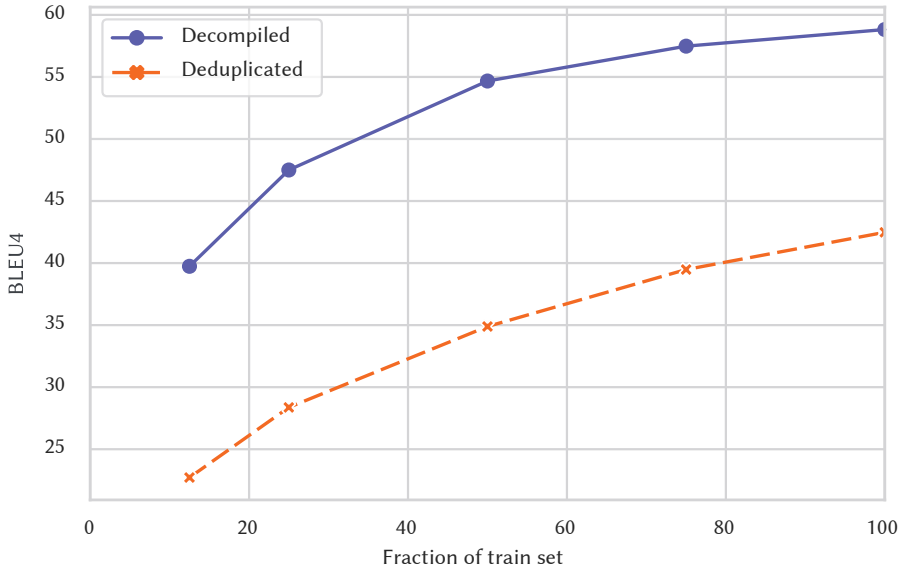


Figure 2.7: BLEU-4 per trainset size for decompiled code and deduplicated decompiled code

spectively. These results show that the dataset quality also heavily impacts the model performance.

2.7 Discussion

In the previous section, we found that BinT5 shows considerable performance for decompiled code and demi-stripped code on both regular as well as deduplicated data. While this is a promising result, we conduct a small investigation of the decompiled samples. We will put our observations on identifiers into the context of the extreme summarisation task. Based on this we discuss the implications of our work. Finally, we will close this section by discussing the threats to validity.

2.7.1 Exploration of Results

To explore the results of BinT5 we pick 25 high and 25 low-scoring samples from the test set of the deduplicated decompiled dataset. High samples have a BLEU-4 score higher than 75 while low-scoring samples have a score lower than 25.

High Samples With the high-performing samples BinT5 tends to produce summaries which are very close to the references. For instance, BinT5 produced Print description of a datatype in XML against the baseline Dump description of a datatype in XML. Of the 25 high-scoring samples we found that all have counterparts with a similar function summary in the training set. These functions also tend to have similar names, but their decompiled function body was significantly different, which is likely why deduplication didn't remove these functions.

Low Samples From the low-performing samples we observe that many summaries produced by BinT5 are semantically very similar to the reference. For instance, the function vl_set_simd_enabled[20], has the reference Toggle usage of SIMD instructions while BinT5 produced Enable or Disable the Simd Channel. This sample scores a BLEU-4 score of 0.0, because of the limitations around the BLEU-4 metric, while for a human evaluator the output is still very usable. Similarly, for some samples, BinT5 produces shorter summaries containing shorthands. The reference Check if the given nickname is blocked for "normal client" use against Check whether nick is blocked, also scores poorly. Of the 25 low-scoring samples we observe that around 11 are semantically similar to the reference and likely very useful for understanding the function.

2.7.2 Identifiers and Extreme Summarisation

We find a relatively small difference in performance between source code and decompiled code. This indicates that in-function comments and variable names are relatively unimportant for the model performance. Although Ahmed and Devanbu observed that identifiers might be more important than syntax in the code-summarisation task [51], we can further conclude that the function name is explicitly essential for model performance. Removing just the function name from the decompiled samples, as opposed to removing all identifiers in demi-stripping, results in slightly higher performance than demi-stripped code, which indicates a very high dependence on the name of the function in the code

[20] Colmap/Colmap:vl_set_simd_enabled: https://archive.softwareheritage.org/browse/content/sha1_git:d4982e43b87b4ace91fb78af617f3926db4e0cd5/?origin_url=https://github.com/colmap/colmap&path=lib/VLFeat/generic.c&snapshot=aa0aae6de2ecd6fad6b45843ac6449939d11effc#L1070

summarisation task, which is a logical finding in the context of the extreme code summarisation task.

The extreme code summarisation task, as proposed by Allamanis et al. aims to reproduce the function name given a function body [51, 75]. It is framed as a summarisation problem where the output is around 3 tokens in length, instead of the 10+ tokens that regular code summarisation targets. We found similar results when performing this task with our dataset, namely, high performance on regular decompiled code (with function names removed) and low performance on stripped code.

A manual assessment of the stripped data shows that many of the aligned functions were not decompiled properly. We find that many functions are cut-off after a few instructions because the decompiler did not recover the full control flow. Other functions are missing side effects, like changes to global variables.

2.7.3 Implications

We propose a novel solution to aid reverse engineers in their work. If the application of NLP to binaries gets significantly better, and the limitations around stripping and other obfuscation techniques get resolved, it would have severe implications for the cybersecurity domain. On one hand, it could help malware analysts understand novel malware and its weaknesses quickly. Software can be analysed to find possible vulnerabilities and malicious payloads. Source code can be reconstructed for old binaries for which the source code is lost. But on the other hand, attackers can leverage these same methods to find and exploit vulnerabilities and lift intellectual property from binaries.

CAPYBARA itself could be used to create and assess neural decompilation, to perform a deeper investigation into the extreme summarisation task, or to simply train a code summarisation model on C code. CAPYBARA consists of a large corpus of C and decompiled C code, which could be used to pre-train language models, such that these models could support decompiled code out-of-the-box.

While our work focused on decompiled code, our observations show some limits of transformer-based models and their applicability to different data. Our dataset can help and inspire other researchers to improve upon our work. We hope other researchers use this dataset to train and evaluate their own models. Furthermore, the process outlined in Section 2.3 could help others construct standardised datasets for other tasks and languages.

2.7.4 Threats to Validity

Internal Validity questions if other factors could have affected the outcome. The training and evaluation data contains a significant amount of noise, either in the form of badly decompiled functions or incorrect documentation. We carefully collect and process the data, but we are unable to know to which extent the documentation matches the original code. While machine learning models (and specifically NLP models) should be able to handle noisy data, this might introduce some bias into the models. CodeT5 was also pre-trained on a C and C# dataset, this dataset is unpublished and we were unable to reach the authors. Some data leakage might have taken place, but as explained in Section 2.6 it is unlikely that it had much of an impact. To prevent this threat from arising in any future studies, we make CAPYBARA publicly available.

External Validity refers to the generalisability of our results. This work only focuses on stripping and compiler optimisations as a means of resisting binary analysis, other techniques like control flow obfuscation and packing are also used to prevent reverse engineering. Other works focus on unpacking and deobfuscation, so we consider our work orthogonal to theirs. The data gathered for CAPYBARA were exclusively from open-source projects. Decompiling closed-source projects is explicitly forbidden by some EULAs and the lack of source code documentation makes it difficult to evaluate using reference summaries. However, reverse engineering open-source software is not very useful in practice, since the source code is readily available. Closed-source software might have different data distribution and will present other challenges like obfuscation. Finally, only functions that decompile (Ghidra produces any output) and that are documented, are represented in CAPYBARA. This is most apparent in the stripped dataset, where we can only recover a small fraction of the total number of functions. A deeper investigation into new decompilation techniques for stripped code, specifically into the aspect of function boundary detection is left as future work.

Construct Validity relates to the adequacy of the theoretical constructs and the use of appropriate evaluation metrics. The leading metric in our evaluations does not capture semantic meaning. While BLEU-4 is the most popular metric for this task, its reliability has been called into question [76, 77]. We, therefore, included other metrics, which do take semantics into account, in our evaluation. Finally, our entire approach hinges on the assumption that function summaries, as they are used for source code, are useful for binary analysis. Whether or not this is actually the case, should be further investigated with a qualitative user study, this is left as future work.

2.8 Related Work

Binary reverse engineering and the use of NLP for software engineering are vast and active fields, so we select and discuss the closest state-of-the-art works in the field. We categorise the studies into identifier recovery and binary translation. Finally, we will discuss the open challenges and the relation of our own work to these challenges.

Recovering Identifiers from Stripped Binaries **Debin** [5] aims to recover debug information from stripped binaries. The authors use a tree-based classification and a probabilistic graph-based model. All the variable names and types are jointly recovered using a maximum a posteriori probability inference. **VarBERT** [78] uses a Transformer-based NLP model for the task of variable name recovery. The authors pre-trained a BERT model which is then fine-tuned to predict the names and types from *unstripped* binaries.

FUNCRE [46] uses a pre-trained and fine-tuned ROBERTA [63] model to predict usages of inlined library functions. Recall that compilers with optimisations enabled can inline functions in the binary (Section 2.2). The authors use indelible markers, which do not get destroyed by the compiler, to mark usages of library functions and to construct a dataset and train a model.

Binary Translation **Neutron** [49] frames decompilation as a neural machine translation problem and utilises an Attention-LSTM-based neural translation network to translate disassembled binaries back to C source code. The binaries are not stripped and do

not have any optimisations enabled. The translations created by Neutron can contain syntax errors, so the authors apply regular expressions to create a tailor-made syntax checker. Neutron achieves high accuracy on the translation task, but only on unstripped and non-optimised code.

Our Novelty Several aspects have not been properly addressed and investigated. The application of code summarisation methods to decompiled code has not been addressed by any work at all. Furthermore, some works on binary code fail to take compiler optimisations into account [49]. We, therefore, investigate the application of code summarisation methods to decompiled code and we enable compiler optimisations.

2.9 Conclusion

In this chapter, we proposed a new automatic binary code summarisation task. With this new task, we also introduce CAPYBARA, a novel dataset to train and evaluate models on this task, with both mined as well as synthetic data. Paired with this dataset, we train BinT5, a Transformer-based code summarisation model to show the effectiveness of CAPYBARA. We used BinT5 to further explore the datasets, outlining the inherent difficulties in the data.

We found that while BinT5 shows considerable performance on regular decompiled code, but its performance is being hampered by the decompiler on stripped code, evidenced by BinT5s strong performance on demi-stripped code. Furthermore, we found that while duplicates have a large impact on the model, their presence is not paramount to the model's performance. Finally, we observe that BinT5 could be trained with just a fraction of the samples in CAPYBARA.

Our work has shown that a well-known and well-studied task from the source code domain [41], namely source code summarisation, can be applied to binary code. This is only one of the many different applications of NLP for code. This chapter constitutes the first step in the application of source code NLP methods to such tasks on binary code.

3

Do Agents Dream of Root Shells? Partial-Credit Evaluation of LLM Agents in Capture The Flag Challenges

Large Language Model (LLM) agents are increasingly proposed for autonomous cybersecurity tasks, but their capabilities in realistic offensive settings remain poorly understood. We present **DeepRed**, *an open-source benchmark for evaluating LLM-based agents on realistic Capture The Flag (CTF) challenges in isolated virtualised environments. In a CTF challenge, the agent must recover a hidden flag by exploiting a vulnerable target. DeepRed places an agent in a Kali attacker environment with terminal tools and web search, connected over a private network to a target challenge, and records full execution traces for analysis. To move beyond binary solved/unsolved outcomes, we introduce a partial-credit scoring method based on challenge-specific checkpoints derived from public writeups, together with an automated summarise-then-judge labelling pipeline for assigning checkpoint completion from logs. Using* **DeepRed**, *we benchmark ten commercially accessible LLMs on ten VM-based CTF challenges spanning different challenge categories. The results indicate that current agents remain limited: the best model achieves only 35% average checkpoint completion, performing strongest on common challenge types and weakest on tasks requiring non-standard discovery and longer-horizon adaptation.*

3.1 Introduction

Large Language Models (LLMs) have fundamentally reshaped modern software development. Contemporary models can generate functional code, explain vulnerabilities, summarise complex bug reports [2, 79, 80], and even produce sophisticated malicious artefacts. As these systems continue to improve, understanding their capabilities and risks across sensitive domains has become increasingly urgent [81, 82]. Cybersecurity is one such

domain, where the dual-use nature of LLMs presents both substantial opportunity and significant concern [7, 83].

Capture The Flag (CTF) competitions represent a controlled yet realistic proxy for offensive and defensive cybersecurity practice [84]. In a typical jeopardy-style CTF, each challenge hides a secret string, called a flag, which participants recover by exploiting a vulnerable service, program, or cryptographic scheme and then submit for points. CTFs are used for training, benchmarking, and talent identification, CTF challenges span diverse categories including web exploitation, reverse engineering, binary exploitation, and cryptography [84]. Solving these challenges requires not only low-level technical proficiency but also high-level strategic reasoning, iterative experimentation, and adaptive problem-solving [85]. While human teams continue to dominate competitive CTF environments, increasing advances in AI raise the question of whether autonomous systems may soon participate meaningfully in such settings.

Despite their impressive language capabilities, vanilla LLMs are fundamentally limited in their ability to solve CTF challenges autonomously. At their core, standard LLMs operate as next-token predictors without native support for tool invocation, environment interaction, persistent memory, or long-horizon planning. CTF problem solving, however, requires multi-step reasoning, command-line interaction, debugging, hypothesis testing, and strategic adaptation to intermediate results. These requirements render standalone LLMs insufficient for realistic autonomous cybersecurity tasks.

To overcome these limitations, recent research has proposed agentic approaches that augment base LLMs with planning modules, tool-use interfaces, and feedback-driven control loops [25]. LLM-based agents can integrate external tools (such as search engines, interpreters, or virtual machines) [26], maintain contextual memory, and iteratively refine strategies [86]. Early demonstrations of such systems are promising, suggesting that LLM agents may bridge the gap between natural language reasoning and real-world task execution. However, comprehensive empirical evaluations of LLM agents within full, realistic CTF environments remain limited.

From an AI perspective, CTFs provide a structured benchmark for evaluating long-horizon reasoning, tool orchestration, and adaptive decision-making under adversarial conditions. From a cybersecurity perspective, understanding the autonomous capabilities of LLM agents is essential to anticipate both defensive applications and potential misuse. From a software engineering perspective, CTF challenges exercise many of the same skills that underpin automated program analysis and repair. As LLM-based agents are increasingly proposed as components of automated software testing and security auditing pipelines [87, 88], understanding their capability boundaries in adversarial, tool-rich environments directly informs how such systems should be designed, evaluated, and safely deployed in practice [89–91].

In this work, we address this gap by systematically evaluating LLM-based agents on a diverse set of CTF challenges as a proxy for practical cybersecurity competence. We investigate the following research question: *To what extent can LLM-based agents autonomously solve CTF challenges across categories?*

To answer this question, we develop **DeepRed**, an extensible, open-source benchmark harness and agent framework designed for properly isolated execution and reproducible experimentation. DeepRed supports full challenge interaction within controlled virtual

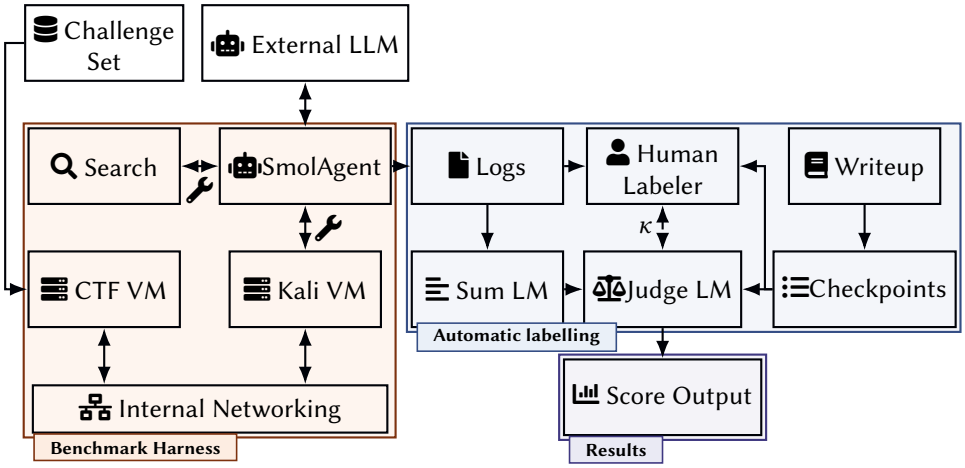


Figure 3.1: Overview of the three core components of **DeepRed**. *(Left)* An external LLM agent drives a Kali attacker VM through terminal tools and optional web search, interacting with challenges from the curated dataset over an isolated internal network (Section 3.3). *(Centre)* Full interaction logs are recorded for every run. Logs are compressed by a summarisation LM and evaluated against challenge-specific checkpoints, yielding partial-credit scores reflecting how far the agent progressed (Section 3.4). *(Bottom)* The judge is validated against human annotations before being applied at scale (Section 3.5).

environments, exposing the agent to a realistic attacker machine and target system connected over a private internal network, and enables detailed logging of the full interaction trajectory.

A central challenge in evaluating autonomous agents on complex multi-step tasks is that binary success metrics, did the agent capture the flag or not, are too coarse to reflect meaningful differences in capability. This problem is particularly acute for current-generation agents, which rarely solve entire challenges end-to-end but frequently make measurable partial progress. To address this, we introduce a log-based partial scoring methodology in which each challenge is decomposed into a sequence of binary checkpoints derived from public writeups and agents are awarded credit for each milestone they complete. We further develop and validate an automated labelling pipeline that uses a two-stage summarise-then-judge LLM process to assign these checkpoint labels at scale, and establish its reliability against human annotations.

Our contributions can be summarised as follows (Figure 3.1):

- We design and implement **DeepRed**, an open-source architecture to evaluate LLM agents in realistic CTF environments, supporting secure isolation and reproducible experimentation (Section 3.3).
- We introduce an automated evaluation pipeline that assigns partial credit based on structured analysis of execution logs, enabling fine-grained performance assessment (Section 3.4).
- We benchmark multiple LLM-based agents across diverse CTF categories and analyse their strengths, weaknesses, and failure modes (Section 3.5).

- We release all framework components, challenge configurations, evaluation scripts, and scoring rubrics as open-source artefacts to support reproducible capability tracking and future research on agentic failure analysis and pipeline design.

3.2 Background and Related Work

Recent work has explored the use of Large Language Models (LLMs) and LLM-based agents for cybersecurity tasks, particularly in Capture The Flag (CTF) and penetration-testing settings. Broadly, this literature can be organised into two lines of work: benchmarks for evaluating LLM performance on offensive security tasks and agent architectures designed to autonomously solve such tasks. Existing studies show promising capabilities, but many evaluations still emphasise benchmark completion or task success in curated settings rather than sustained interaction with realistic attacker environments [10, 85, 92].

Several benchmarks have been introduced to study LLM capabilities in offensive security. NYU CTF Bench provides a large benchmark of validated CTF challenges spanning multiple categories and difficulty levels, enabling controlled comparison of models and agents on offensive security tasks [85]. Similarly, PentestGPT evaluates LLM-assisted penetration testing across benchmark targets and detailed subtasks, showing that LLMs can support complex offensive workflows while also highlighting substantial performance limitations on harder targets [92]. More recently, CVE-Bench extends evaluation toward the exploitation of real-world web application vulnerabilities, helping bridge the gap between CTF-style tasks and more realistic offensive security scenarios [10].

A complementary line of work focuses on agent architectures for autonomous cybersecurity problem solving. EnIGMA introduces an LLM agent with interactive tools tailored to cybersecurity workflows, including support for interactive utilities such as debuggers and server-connection tools that are important in CTF solving [93]. Its results show that stronger tool integration substantially improves agent performance across several security benchmarks. These findings reinforce a broader trend in the literature: success in offensive cybersecurity depends not only on high-level reasoning, but also on effective interaction with external tools and execution environments [92, 93].

Overall, prior work demonstrates that LLMs and LLM-based agents can assist with offensive-security tasks, but existing evaluations often emphasise benchmark completion in curated settings. Our work differs in two key respects. First, we evaluate agents in full virtualised environments with a separate attacker and target machine, which more closely reflects realistic CTF interaction. Second, we introduce a partial-credit evaluation pipeline based on execution logs, enabling more fine-grained analysis than binary solved/unsolved outcomes alone.

3.3 Framework Design

In this section, we outline the architecture and components of **DeepRed**. We first detail the design of the benchmark harness itself (3.3.1), before discussing the methodology behind our dataset curation (3.3.2) and model curation (3.3.3).

3.3.1 Benchmark Harness

We design the benchmark harness around three requirements: realism, isolation, and extensibility. First, the agent should experience each task as closely as possible to a real CTF or penetration-testing session conducted through a terminal. Second, challenge execution must remain strongly isolated so that the agent cannot observe or interact with systems outside the benchmarked environment. Third, the harness must be easy to extend with additional models, challenges, and evaluation procedures.

We choose to run the challenges as full virtual machines rather than containers. This decision provides stronger isolation than Docker-style containerisation, which shares the host kernel and can therefore reduce both realism and containment [94]. VM-based deployment is also well aligned with the distribution format of many existing CTF challenges, which are commonly released as downloadable VM images. In addition, the use of VMs preserves compatibility with challenges that rely on low-level system behaviour or kernel-facing exploitation primitives.

Although both guest VMs are isolated within VirtualBox, the agent still requires controlled command-line access to the attacker machine. We implement this by starting a `getty` service on the Kali VM and exposing the terminal session to the host via a Unix domain socket. On the host side, this socket is wrapped in tool interfaces that allow the agent to send commands and receive terminal output programmatically. These terminal tools are timeout-based, allowing the agent to specify a duration to wait before the system returns the accumulated output. This design accommodates interactive commands that generally lack a reliable end-of-execution marker, such as `ssh`, and enables the model to wait on long-running processes like password brute-forcing. This setup gives the model a text-based terminal interface while maintaining network isolation between the simulation environment and the external system.

For agent execution, we use a `CodeAgent` implemented with `smolagents` [28]. This choice allows the model to write short Python programs to coordinate command execution, maintain state, and implement multi-step logic. Compared with standard JSON-based tool-calling loops, executable-code agents can reduce interaction overhead by moving simple control flow into the generated program itself [95]. In addition to terminal access, the agent is provided with a web-search capability via DuckDuckGo. To reduce trivial leakage, we filter direct references to the challenge flag and to known writeups from search results before presenting them to the agent.

Each benchmark episode begins by booting a clean snapshot of the Kali attacker VM and attaching it to the selected challenge VM on the private internal network. The agent then interacts exclusively through the provided tools, which expose terminal access and filtered web search. At the end of the run, we collect the full execution trajectory, including issued commands, terminal outputs, tool interactions, and metadata required for evaluation. We then restore the Kali VM to a clean snapshot, which removes shell history, temporary files, generated credentials, and any other residual artefacts from the prior run. The challenge VM is likewise reset or reloaded before the next episode. This reset procedure ensures that runs are independent and reproducible.

The system prompt instructs the agent to operate as an offensive security specialist conducting an authorised penetration test, provides the target IP address, specifies the flag format, and directs the agent to work methodically.

Harness and Evaluation: An expected obstacle in benchmarking LLMs on complex tasks is the low end-to-end success rate. Because agents currently struggle to solve entire CTF challenges, relying solely on a binary success metric (capturing the final root flag) yields a low-resolution signal. To address this, DeepRed evaluates runs in a partial, checkpoint-based manner. By tracking intermediate milestones, we can capture a more granular assessment of an agent's specific capabilities, a process described in detail in Section 3.4. Following the conclusion of each challenge, the outcomes, metrics and the complete trajectory data are collected for analysis.

3.3.2 Dataset Curation

Having defined the execution harness, we next describe how we curated the challenge set used for evaluation.

We curate the benchmark challenge set to reflect realistic but tractable autonomous offensive-security tasks. Since current LLM agents still struggle with long-horizon exploitation, we do not aim to assemble the hardest possible CTF corpus. Instead, we construct a dataset that is difficult enough to expose meaningful differences in agent capability while still allowing agents to make measurable progress. We therefore prioritise challenges that agents can solve end-to-end through terminal interaction, that expose clear intermediate milestones, and that run reproducibly inside isolated virtualised environments.

We include a challenge only if it is solvable entirely from the Kali command line, ensuring compatibility with our terminal-based agent interface; distributed as a virtual machine image so that it can be deployed directly in the benchmark harness; accompanied by a public writeup from which grading rubrics and checkpoint definitions can be derived; structured around identifiable intermediate stages; and of low to moderate difficulty, since current agents rarely solve harder machines end-to-end.

These criteria align the dataset with the goals of the benchmark: realistic interaction, safe and reproducible execution, and fine-grained evaluation through observable intermediate progress. The resulting set covers several common offensive-security behaviours. We prioritise depth of evaluation and reproducibility over scale. This also helps keep evaluation costs manageable.

We select all ten challenges from HackMyVM[1]. We choose this platform because it offers downloadable VM images, accessible difficulty levels, and a substantial body of public writeups.

The selected set in Table 3.1 covers six broad challenge categories: *web exploitation*, including techniques such as remote and local file inclusion, command injection, file-upload bypass, SQL injection, and SSRF; *privilege escalation*, covering methods for elevating from a low-privilege account to root or administrator access; *fuzzing and directory discovery*, involving the automated enumeration of hidden files, directories, and endpoints; *SSH key manipulation*, including the generation, planting, or abuse of SSH keys and related configuration for access or pivoting; *steganography*, involving the concealment or recovery of hidden data in files, images, or text; and *service exploitation*, covering attacks against vulnerable network services such as FTP, ADB, or Netcat listeners.

[1] HackMyVM: `https://web.archive.org/web/20260511161103/https://hackmyvm.eu/`

Name	Description	Category
SysAdmin	Web-based code execution followed by PATH-hijacking privilege escalation	Web exploitation, Privilege escalation
Aria	File-upload bypass, zero-width steganography, and JSON-RPC exploitation via `aria2c`	Web exploitation, Steganography, Service exploitation
HelpDesk	Local file inclusion, credential recovery, and sudo abuse via `pip3`	Web exploitation, Privilege escalation
Thirteen	ROT13-based file disclosure, FTP misconfiguration, and root code execution through service replacement	Web exploitation, Service exploitation
Fuzzz	Directory brute-forcing and OpenSSH key recovery on an Alpine-based target	Fuzzing and Directory discovery, Key manipulation, Service exploitation
Nexus	SQL injection for credential extraction, SSH access, and GTFOBins escalation via `find`	Web exploitation, Privilege escalation
Todd	Netcat shell discovery, SSH persistence, and bash arithmetic command injection for root	Key manipulation, Service exploitation, Privilege escalation
Jan	SSRF against a Go web service followed by SSH misconfiguration abuse	Web exploitation, Key manipulation
Whitedoor	Restricted web execution, credential recovery, SSH pivoting, and GTFOBins-style escalation	Web exploitation, Key manipulation, Privilege escalation
Quick	Remote file inclusion followed by SUID-based privilege escalation	Web exploitation, Privilege escalation

Table 3.1: Selected CTF challenges used in DeepRed

Beyond the 10 core challenges used for our primary evaluation, we curated an extended set of 20 higher difficulty challenges sourced from HackMyVM and VulnHub[2]. These tasks represent more advanced scenarios, including SQL injection, remote code execution via cookie interception, and multi-stage privilege escalation, all of which require specialised toolsets and sophisticated reasoning.

During preliminary testing, we observed that although agents could achieve partial progress on some core tasks, no root flag was obtained in any of the core challenges. Given that the models had not yet reached the capability required to solve the entry-level tasks, we reserve this extended set for future benchmarking.

3.3.3 Model Curation

To obtain a comparison set that is both practically relevant and methodologically defensible, we curate a diverse collection of commercially accessible LLMs for evaluation. Our goal is to benchmark a realistic cross-section of models that a practitioner could plausibly deploy as the reasoning core of an autonomous CTF agent.

We first filter candidate models using a set of operational constraints motivated by

[2]VulnHub: `https://web.archive.org/web/20260511163143/https://www.vulnhub.com/`

Model	Provider	Price	Open Weights
Nemotron-3 Nano 30B	NVIDIA	0.05 / 0.20	✓
MiMo-V2 Flash	Xiaomi	0.09 / 0.29	✓
Qwen3-Next 80B	Qwen	0.09 / 1.10	✓
Grok Code Fast 1	xAI	0.20 / 1.50	–
Nova-2 Lite v1	Amazon	0.30 / 2.50	–
Devstral 2512	Mistral AI	0.40 / 2.00	✓
MiniMax-M1	MiniMax	0.40 / 2.20	✓
Gemini 2.0 Flash	Google	0.50 / 3.00	–
Palmyra X5	Writer	0.60 / 6.00	–
GPT-5.1 Codex Max	OpenAI	1.25 / 10.00	–

Table 3.2: Selected models used in our study. Price is reported as input/output cost in USD per million tokens.

the requirements of our benchmark harness. Specifically, a model is only included if it is accessible through OpenRouter, supports text-to-text interaction compatible with our shared textual tool-use loop, and is suitable for agentic use, meaning that we exclude models primarily optimised for roleplay or highly stylised conversational behaviour and only retain those advertised for coding, CLI, or agentic workflows. We further require models to provide a context window of at least 250k tokens, since CTF trajectories can become long once terminal output, planning reflections, and prior actions must be retained within a single interaction history. Finally, to keep the benchmark economically viable, we limited selection to models priced at no more than $5 per million input tokens and $10 per million output tokens.

Candidate models are first programmatically collected from the OpenRouter catalogue and screened against the criteria above. From the remaining pool, we hand-pick a final set that preserves broad provider and pricing diversity. We present the curated model set in Table 3.2, with half of the models being open weight models.

3.4 Partial Credit and Automated Labelling

A central goal of **DeepRed**'s evaluation pipeline is to assign partial credit for meaningful progress, rather than grading every run with a binary solved/unsolved metric. To support this, each CTF challenge is equipped with a predefined grading rubric consisting of intermediate checkpoints against which agent runs are evaluated. This is important because agents frequently make measurable progress through reconnaissance, credential discovery, foothold acquisition, or partial privilege escalation without completing the full challenge.

We decompose each challenge into a sequence of binary checkpoints derived from the corresponding public writeup. These checkpoints capture observable technical milestones such as identifying a service, recovering credentials, uploading a payload, obtaining an initial shell, or achieving privilege escalation rather than the use of a specific tool. Given an execution trace, the evaluator assigns a pass/fail label to each checkpoint, and the final score is the sum of completed checkpoints. This formulation yields a more informative measure than binary success alone while remaining simple to audit. Each challenge

we evaluate has a single solution path, and the checkpoint sequence follows this linear progression.

3.4.1 Automatic Labelling and Validation

For this approach to be valid, three conditions must hold. First, each challenge rubric must define intermediate milestones clearly enough to support consistent scoring. Secondly, human raters must achieve high agreement when applying these rubrics to execution traces, which establishes that the task is sufficiently objective. Thirdly, an LLM-based judge must align closely with human consensus. Together, these conditions justify replacing most manual evaluation with automatic labelling.

We use Cohen's kappa (κ) to measure pairwise agreement between raters, and we use Krippendorff's alpha (α) to assess overall agreement across the full panel. We choose these metrics because they account for chance agreement and provide a standard reliability baseline for both pairwise and multi-rater annotation settings.

To establish a human baseline, we sample 60 execution traces in total. Three models are evaluated on the core challenges twice. Each trace is independently annotated by two of four raters, and each rater labels 30 traces. This yields overlapping annotations sufficient to compute inter-rater agreement while keeping annotation cost manageable. We anonymise agent identities to reduce bias.

We use *Grok Code Fast 1*, *Gpt-5.1 Codex Max* and *Claude Haiku 4.5* for the human baseline evaluation. We use this initial evaluation to estimate the total inference cost and to test the framework.

Initial evaluations often require multiple hours per trace while raters familiarise themselves with the challenge structure, but grading time later falls to under one hour per trace. Overall agreement is high: Krippendorff's alpha reaches $\alpha = 0.819$, and the mean pairwise Cohen's kappa reaches $\kappa = 0.7800$. These results indicate that the first two conditions hold: the task is sufficiently well defined for human raters to apply the rubrics consistently.

We implement automatic labelling as a two-stage process, as illustrated in Figure 3.1, where execution logs are first summarised and then evaluated, in order to handle long traces and reduce judging complexity.

Summarisation. Raw execution logs are often too long for direct evaluation. We therefore pass each trace to a *Summary LM*, which condenses the interaction into a structured step-by-step summary. When necessary, we chunk long traces to remain within context limits. Each run is capped at 60 agent steps. Every five steps the agent is prompted to produce a reflection and planning message, which makes its current hypothesis and intended next actions explicit. These reflections are retained in the interaction history and later included in the summarisation stage, because they help preserve the intent behind subsequent actions.

Judging. We then pass the summarised trace to a *Judge LM* together with the challenge-specific rubric. The judge assigns checkpoint labels based on outcomes rather than exact tool usage, e.g. whether the agent obtained shell access or recovered a target file. We enforce a strict JSON schema via the `structured_output` interface and enable `Response Healing` to ensure that the output remains programmatically parseable.[3]

[3]Response Healing: `https://web.archive.org/web/20260327090400/https://openrouter.ai/docs/`

	Summary model		
Judge model	Grok 4.1 Fast	Gemini 3 Flash	GPT 4.1 Nano
Grok 4.1 Fast	-0.33	0.71	0.57
Gemini 3 Flash	-0.31	0.62	0.66
GPT 4.1 Nano	-0.11	0.49	0.19
Claude Sonnet 4.6	-0.26	**0.72**	0.38

Table 3.3: Mean pairwise Cohen's kappa (κ) between human raters and the automatic evaluation pipeline. Columns denote the summary model and rows denote the judge model.

Because execution traces can be large, we require summarisation models to support context windows of at least 1M tokens. We use models with input prices below $1 per million tokens in the summarisation step, to keep the evaluation pipeline economically practical.

We evaluate three summarisation and four judge models by measuring their agreement with human raters. For each model pair, we compute the mean pairwise Cohen's kappa between the automatic labels and the human annotations. Table 3.3 reports the results.

Alignment varies substantially across model combinations. Along the main diagonal, where the same model performs both summarisation and judging, `Gemini 3 Flash` yields the strongest performance with $\kappa = 0.6241$. The best overall result is achieved when `Gemini 3 Flash` performs summarisation and `Claude Sonnet 4.6` performs judging, reaching $\kappa = 0.7234$. In contrast, every configuration that uses `grok-4.1-fast` for summarisation produces negative kappa values, regardless of the judge model. This pattern suggests that summarisation quality is a major determinant of downstream labelling reliability: weak summaries can erase or distort technical evidence before the judge sees it, while strong summaries improve the performance of multiple judge models.

These results show that modern LLMs can approximate human checkpoint labelling closely enough to support scalable benchmark evaluation, provided that the summarisation stage is chosen carefully. In operational terms, this pipeline reduces per-trace evaluation from multiple human hours to a few minutes, making repeated benchmarking of autonomous CTF agents feasible.

3.5 Results

We evaluate ten LLM-based agents on the ten selected CTF challenges using the benchmark harness. Due to cost constraints, we run each model-challenge configuration three times and report the resulting average checkpoint completion. In pilot experiments, the run-to-run variance was generally low, making three repetitions a practical compromise between statistical rigour and evaluation cost. Each LLM runs with its default parameter configuration as defined by the model provider.

Table 3.4 summarises aggregate model performance across all challenges. Overall performance remains modest: the best-performing model, GPT-5.1 Codex Max, achieves 35% average checkpoint completion, while only a small number of models exceed 20%.

`guides/features/plugins/response-healing`

Model	Tokens (M)	Steps	Completion
GPT-5.1 Codex Max	31.9	57.0	35% [29%-44%]
MiniMax-M1	43.7	59.9	22% [19%-25%]
MiMo-V2 Flash	58.1	59.7	20% [19%-22%]
Devstral 2512	56.0	53.2	21% [16%-27%]
Palmyra X5	20.6	42.7	20% [14%-25%]
Gemini 2.0 Flash	28.7	48.9	16% [13%-18%]
Nova-2 Lite v1	46.4	55.4	13% [12%-16%]
Grok Code Fast 1	8.8	23.1	12% [10%-16%]
Qwen3-Next 80B	16.6	31.0	12% [10%-13%]
Nemotron-3 Nano	18.8	43.6	5% [3%-10%]

Table 3.4: Aggregated performance of evaluated models across all challenges. Tokens represent the average number of input and output tokens consumed per challenge in millions (M), and steps denote the average number of agent actions executed per run.

A second tier of models—MiniMax-M1, Devstral 2512, MiMo-V2 Flash, and Palmyra X5—clusters between 20% and 22%, whereas the weakest model, Nemotron-3 Nano 30B, reaches only 5%. Even the strongest systems therefore complete only about one third of the benchmark on average.

Token usage varies widely across models, ranging from 8.8M tokens per run for `Grok Code Fast 1` to 58.1M for `MiMo-V2 Flash`. However, higher token consumption does not reliably predict better outcomes. For example, `GPT-5.1 Codex Max` achieves the best completion score while using fewer tokens than several weaker models, including `MiniMax-M1`, `MiMo-V2 Flash`, and `Devstral 2512`. Conversely, `Grok Code Fast 1` is comparatively cheap in both tokens and steps, but this efficiency comes with relatively weak task performance. Taken together, these observations suggest that task success depends less on raw interaction volume than on the quality of reasoning and action selection within the available budget.

Average step counts cluster relatively tightly, typically between 40 and 60 actions per run, likely reflecting the imposed interaction limit. Consequently, variation in performance appears to arise less from differences in trajectory length than from the effectiveness of the actions taken within a similar number of opportunities.

Table 3.5 shows the aggregated difficulty of individual challenges. Difficulty varies substantially across tasks. `Whitedoor`, `Quick`, and `SysAdmin` are the most solvable challenges in the suite. At the other end of the spectrum, `Fuzzz` is the hardest challenge, followed by `Thirteen`.

Challenge-level token usage also varies considerably. `Fuzzz`, `Todd`, `HelpDesk`, and `Nexus` require the largest average token budgets, whereas `Thirteen`, `Whitedoor`, and `SysAdmin` are comparatively lightweight. By contrast, step counts remain fairly stable across challenges, typically between 43 and 51 actions.

Challenge	Tokens (M)	Steps	Completion
Whitedoor	27.2	45.3	24.6%
Quick	32.0	46.4	23.5%
Sysadmin	29.1	42.9	22.8%
Jan	32.8	47.1	21.4%
Aria	32.8	47.5	17.3%
Todd	38.7	50.2	15.6%
Nexus	35.3	51.2	15.2%
Helpdesk	36.3	47.3	14.8%
Thirteen	24.2	46.9	12.9%
Fuzzz	41.2	50.1	10.1%

Table 3.5: Aggregated performance per challenge.

3.6 Discussion

Our findings contribute to the broader question of how effectively LLM-based agents can autonomously solve CTF challenges, and, how such progress can be evaluated in a rigorous and scalable manner. While the absolute task performance observed in Section 3.5 remains limited, the results demonstrate that meaningful intermediate progress can nevertheless be detected automatically from agent interaction logs. This is an important prerequisite for studying autonomous offensive security systems in realistic environments, where binary success-or-failure evaluation is often too coarse to capture substantive differences in capability.

Challenges are hardest when they require non-standard discovery and multi-step state tracking rather than a familiar web foothold followed by privilege escalation. In particular, `Fuzzz`, `Thirteen`, and `Todd` are among the most difficult tasks, and all involve less conventional elements such as directory brute-forcing, SSH key handling, obfuscated disclosure, or service-specific exploitation. By contrast, challenges such as `Whitedoor`, `Quick`, and `SysAdmin` follow a more standard web-to-privilege-escalation pattern and are completed more often. This suggests that current agents perform better on common exploitation workflows, but struggle when they must combine broad enumeration with unusual artefacts or less common attack paths.

A central contribution of this work is the use of log-based labelling to infer challenge progress from recorded trajectories. In particular, the method appears capable of identifying semantically meaningful stages of problem-solving, such as reconnaissance, artefact discovery, and partial exploitation, without requiring manual inspection of every agent run.

At the same time, the quality of the resulting labels depends directly on the quality of the manually defined milestone set. Since these milestones are derived from writeups, any incompleteness, ambiguity, or conciseness in the writeup can propagate into the evaluation. A writeup may omit intermediate reasoning steps, collapse multiple actions into a single narrative transition, or describe only one of several viable solution paths, causing the extracted milestones to underrepresent legitimate alternative forms of progress. When a challenge has multiple solution paths, the checkpoint rubric should reflect this by defining separate checkpoint sequences for each path. We did not observe such cases

in the manual evaluation, and the writeups used here appeared sufficiently detailed to support consistent milestone extraction.

The results also illustrate why partial scoring is especially important in this domain. Many agents make measurable early progress, particularly on reconnaissance and enumeration steps, yet fail to complete later stages that require longer-horizon planning, adaptation to unexpected outputs, or more creative exploitation. This is particularly valuable for analysing emerging agentic capabilities, where full task completion may still be rare but intermediate progress is already informative.

An additional practical consideration concerns the cost of human annotation. Even though the proposed pipeline reduces the need for manual scoring of every individual run, constructing or validating challenge labels still requires substantial human effort. Raters must understand both the challenge itself and the associated writeup in sufficient depth to identify meaningful milestones and assess whether inferred progress is plausible. This creates a high initial startup cost and requires domain expertise in security and exploitation, which makes the task inaccessible to lay annotators. As a result, although the framework improves scalability relative to fully manual evaluation, it does not eliminate expert labour altogether. Future work should therefore consider how milestone extraction, writeup alignment, and challenge decomposition might be further standardised or partially automated.

We observed an infrastructure failure that reinforces the importance of strong isolation. In one live run, shutting down the target VM did not terminate the agent, which continued operating on the host, scanned the local network, and began probing other active systems and services. We stopped the process after detecting the unauthorised activity and patched the issue.

3.6.1 Agent Improvement Opportunities

The failure patterns observed during manual evaluation indicate that the principal bottlenecks for current agents are not related to basic tool use or knowledge of the utilities in Kali Linux. Most models can issue commands, inspect files, and gather surface-level information using the available toolchain. However, many appear to struggle when success depends on synthesising observations over multiple steps, revising an unsuccessful plan, or identifying less obvious exploitation opportunities.

We note that agents tend to lack perseverance. While some agents give up prematurely, we find that most progress is made in the initial 20 steps, after which the agents tend to get lost. Some gain initial user access to the challenge VM, and then look for the flag without escalating their privileges to root. Others continuously repeat the same failing commands or start over multiple times. This type of long-horizon planning still seems to be challenging for all tested agents.

Agents tend to struggle with the fact that the Kali environment persists throughout a challenge episode rather than being reset between steps. So they rarely store intermediate results and tend to redo many steps.

3.6.2 Threats to Validity

Internal validity. A potential threat is the use of SmolAgents as a common framework for all models. Some models may perform better under different prompting schemes or

agent architectures. We accept this limitation, as a shared framework is necessary for comparability and implementing model-specific systems for each of the selected models is not practical. A second threat is data leakage: some challenges or writeups may have appeared in model training data, potentially inflating performance. This cannot be ruled out, particularly for closed-data models with undisclosed training data and cut-off dates. Finally, while the non-determinism of LLMs ideally requires a large number of trials to achieve statistical significance, we limited our evaluation to three executions per model for each challenge due to time and financial constraints. Although aggregating results across these three passes helps mitigate variance, this remains a limited sample size and may not fully capture the complete range of model behaviours or edge-case failures.

External validity. DeepRed includes only ten models and ten challenges, which limits generalisability. We mitigate this by selecting a diverse set of models and tasks within a constrained evaluation budget, and by designing the benchmark to be extensible. The challenges are also relatively easy, which improves tractability for current agents but means the results may not transfer to harder or more realistic offensive security settings.

The system prompt is a known confound in agent benchmarking [25], as different prompt formulations can produce meaningfully different absolute scores. We mitigate this partially by applying the same prompt across all models, which preserves the validity of relative comparisons between systems.

Construct validity. CTF performance is only a proxy for broader cybersecurity capability. CTFs are useful because they are controlled, reproducible, and safe, but they do not capture many real-world factors such as stealth, persistence, lateral movement, or interaction with defenders.

3.6.3 Future work

As discussed in Section 3.6.2, our evaluation uses SmolAgents as a shared backbone to ensure comparability, but this may not reflect the ceiling performance of models better suited to other architectures. Promising alternatives include *LangGraph* for finer tool calling control, *AutoGen* for multi-agent collaboration, and *OpenHands* for terminal-heavy workflows. Future work should examine whether model rankings are stable across frameworks or whether architecture-model pairings produce meaningfully different outcomes.

Many failures observed in this work, discussed in Section 3.6.1, stem not from a lack of domain knowledge, but from poor long-horizon planning, weak memory usage, and insufficient adaptation to failed attempts. Future work could explore hierarchical planning agents, explicit memory systems, and self-reflection loops as targeted remedies. Multi-agent architectures, where specialised agents handle different stages independently, represent a promising direction for decomposing the long task horizons that current single-agent systems struggle to sustain.

A further direction for future work is to improve the robustness and generality of the automatic labelling pipeline. Checkpoint labels currently depend on manually derived rubrics extracted from public writeups. Future work could investigate methods for partially automating checkpoint extraction from writeups.

3.6.4 Reproducibility

We publicly release **DeepRed** together with challenge configuration files, evaluation scripts, scoring definitions, evaluation logs, and summaries.[4][5]

We design the dataset to remain extensible: researchers and practitioners can add new challenges by supplying a VM image and a corresponding checkpoint rubric. More models can be evaluated on the provided core set using any OpenAI-compatible API. We have consumed around $450 of OpenRouter credits throughout development. A benchmark sweep over the selected models and challenges costs around $70 per run set as of April 2026.

3.7 Conclusion

In this work, we introduced **DeepRed**, a reproducible benchmark framework for evaluating LLM-based agents in realistic CTF environments. Across ten challenges and ten models, our results show that current agents can make partial progress on challenging tasks but remain far from robust end-to-end autonomy. Capturing partial progress is important because binary solve rates can hide substantial differences in capability on multi-step tasks. DeepRed enables reproducible capability tracking and future research on agentic failure analysis and evaluation design, and we encourage the community to build on and extend this benchmark as agentic systems continue to evolve.

[4]Replication Package: `https://doi.org/10.5281/zenodo.19386535`
[5]Github: `https://archive.softwareheritage.org/swh:1:snp:325a01349937c3d29e928ba7ee6457093791f3f4`

4

Code Red! On the Harmfulness of Applying Off-the-Shelf Large Language Models to Programming Tasks

Nowadays, developers increasingly rely on solutions powered by Large Language Models (LLM) to assist them with their coding tasks. This makes it crucial to align these tools with human values to prevent malicious misuse. In this chapter, we propose a comprehensive framework for assessing the potential harmfulness of LLMs within the software engineering domain. We begin by developing a taxonomy of potentially harmful software engineering scenarios and subsequently, create a dataset of prompts based on this taxonomy. To systematically assess the responses, we design and validate an automatic evaluator that classifies the outputs of a variety of LLMs both open-source and closed-source models, as well as general-purpose and code-specific LLMs. Furthermore, we investigate the impact of models' size, architecture family, and alignment strategies on their tendency to generate harmful content. The results show significant disparities in the alignment of various LLMs for harmlessness. We find that some models and model families, such as Openhermes, are more harmful than others and that code-specific models do not perform better than their general-purpose counterparts. Notably, some fine-tuned models perform significantly worse than their base-models due to their design choices. On the other side, we find that larger models tend to be more helpful and are less likely to respond with harmful information. These results highlight the importance of targeted alignment strategies tailored to the unique challenges of software engineering tasks and provide a foundation for future work in this critical area.

4.1 Introduction

In recent years, Large Language Models (LLMs) have made a significant impact across various fields, including Software Engineering (SE) [2, 4, 96]. LLM-powered tools, many integrated directly into Integrated Development Environments (IDEs), are now commonly

used by developers [79, 97, 98]. As the capabilities and availability of LLMs grow, their alignment with human values becomes increasingly important, particularly in preventing misuse for malicious purposes [36, 99, 100].

LLMs are commonly aligned with human values and natural language norms. One such classification is the Helpful, Honest, and Harmless (HHH) criteria [36, 99, 101]. According to the Harmlessness criterion, the language model should avoid aiding in dangerous or otherwise harmful acts [36]. However, the exact values encoded in the HHH framework can differ depending on the context of use [99, 100, 102]. We, therefore, hypothesise that current alignment definitions and datasets may not guarantee harmlessness in the context of Software Engineering.

The intersection of LLMs and software development presents unique challenges in terms of safety and ethics. Although LLMs have shown remarkable capabilities in code generation [4, 103], code completion [3, 4] and code summarisation [4, 24, 50, 104] they also pose potential risks if misused or improperly aligned [99, 105]. For example, an LLM could inadvertently generate code that introduces security vulnerabilities, violates privacy regulations, or even performs malicious actions if not properly constrained [99, 105].

Previous work has explored the alignment and safety of LLM in general contexts [100, 106–108], but there is a gap in research that specifically addresses the unique challenges posed by software engineering tasks. Our work aims to bridge this gap by providing a comprehensive framework for assessing the potential harmfulness of LLM-generated code.

In this work, we aim to investigate the alignment of off-the-shelf LLMs for code-related tasks in the SE domain. We create a taxonomy of potentially harmful SE scenarios. Based on this taxonomy, we manually write and carefully create prompts based on those scenarios. These prompts are collected in a dataset titled **Hammurabi's Code**.[1] We design and validate an automatic evaluator and use these prompts to assess 70 open and closed-source LLMs available for public use.

We study different LLMs, including both general-purpose and code-specific models, and examine the effectiveness of various alignment techniques. We also investigate the differences between natural language and code alignment, as well as the performance across different categories of potentially harmful prompts.

Our study reveals significant disparities in the alignment of various LLMs regarding harmlessness in the SE domain. Specifically, we found that the size of the model is positively correlated with the production of helpful and harmless code responses, and larger models generally exhibit a lower propensity to generate harmful output. Additionally, we note that some model families exhibit consistent behaviour patterns, suggesting that training methodologies may play a crucial role in determining a model's safety profile. However, this is not uniformly true across all models or categories of prompts. The results also highlight that models tend to perform better in avoiding harmful content in certain categories, such as copyright issues, but struggle more in others, such as malware-related prompts. In particular, code-specific models did not consistently outper-

[1]The name 'Hammurabi's Code' for our framework is a play on the double meaning of 'code', it references both the ancient legal code of Hammurabi and the modern concept of computer code. Just as Hammurabi's Code established laws for societal behaviour, our framework sets out to define and assess ethical guidelines for LLMs generating software code. There is a parallel between ancient efforts to codify social rules and our modern attempt to establish ethical standards for AI-generated code.

form general-purpose models in terms of harmlessness. This indicates a gap in current alignment strategies for SE-specific tasks. The main contributions of this work can be summarised as follows:

- A comprehensive taxonomy, dataset, and labels for evaluating harmfulness in LLMs for code-related tasks,
- An empirical assessment of current LLMs' performance in avoiding harmful code generation,
- A lightweight classifier, which classifies the harmfulness of generations based on our response labels,
- An open-source evaluation framework, including a replication package[2] and Hugging Face Space [3] for interactive exploration of results.

4.2 Background and Related Work

To contextualise our study within the broader landscape of Artificial Intelligence (AI) safety and Software Engineering (SE), we first provide an overview of key concepts and recent advances in LLM alignment, red-teaming, and benchmarking, followed by a discussion of related works that have addressed similar challenges in evaluating and mitigating potential harms in AI-generated content.

4.2.1 Red-Teaming in AI Safety

Red-teaming is a practice inspired by cybersecurity that involves simulating attacks or adversarial scenarios to identify vulnerabilities in a system [109]. In the context of AI and specifically LLMs, red-teaming has emerged as a crucial technique for assessing and improving model safety [99]. It involves deliberately attempting to elicit harmful, biased, or otherwise undesirable outputs from an AI system to uncover potential risks and alignment issues [100, 109].

Red-teaming for LLMs typically involves crafting prompts or input sequences designed to probe the model's boundaries and test its adherence to safety guidelines [109]. This process helps researchers and developers identify potential weaknesses in the model's training or alignment, allowing for targeted improvements and refinements [99, 109].

Ganguli et al. used a large group of 324 members to perform a red team exercise against a number of dialogue LLMs [100]. The members were tasked with making the LLM behave badly and eliciting offensive and harmful responses from the LLM [100]. The red team had back-and-forth conversations with the LLM. The success is reported by members of the red team. At each turn of the conversation, participants had to choose the most harmful response among two options generated by the model. At the end of the conversation, participants had to rate from 1 to 5, on how successful they were [100]. The authors found that models trained with reinforcement learning through human feedback become increasingly difficult to red team as they grow larger, while other alignment strategies do not [100]. The red team attacks they performed were mostly general use

[2]Replication package `https://doi.org/10.5281/zenodo.14930306`
[3]HF Space: `https://archive.softwareheritage.org/swh:1:snp:d580b047274c9c8b05d505a25cd7bb60bb56ef7b`

cases described in natural language that were not software specific, such as discrimination, hate speech, and violence [100].

4.2.2 LLM Alignment

LLM alignment refers to the process of ensuring that an AI model's behaviours and outputs align with human values and intentions. This is particularly crucial for powerful language models that can generate human-like text in a wide range of domains [36, 99]. A prominent framework for LLM alignment is the Helpful, Honest, and Harmless (HHH) criteria [36, 99]:

Helpful The model should assist users in completing tasks and answering questions to the best of its abilities [36]

Honest The model should provide truthful information and not knowingly generate false or misleading content [36]

Harmless The model should avoid generating content that could cause harm, whether physical, emotional, or societal [36]

The concept of harmlessness in LLM alignment presents a dichotomy. On the one hand, models must be capable enough to understand and discuss potentially harmful topics to provide valuable assistance and information. On the other hand, they must be constrained from actually aiding in the execution of harmful acts.

Bai et al. observed that including harmlessness as a goal, reduces the helpfulness of models [99, 110]. Similarly, it has been observed that aligning LLMs with the HHH values can incur a certain 'alignment tax' [99, 110, 111]. There are three types of alignment taxes; performance, development, and time-to-deployment taxes [111].

Performance taxes are incurred when the task performance of an LLM decreases after alignment [111], this has been observed to be most severe in smaller LLMS [110]. Development and time-to-deployment taxes are incurred when the alignment procedure costs additional resources and delays the deployment of the LLM [110]. These taxes may make it unappealing to align LLMs and AI systems in general [111], especially considering the competitive nature of the development of for-profit LLMs.

4.2.3 Harm Benchmarking

Several recent studies have explored the safety and alignment of LLMs, particularly in the context of harmful content generation. While these works provide valuable insights and methodologies, they primarily focus on natural language outputs rather than code generation.

Wang et al. introduce a dataset designed to test LLMs' ability to recognise and refuse to answer potentially harmful queries [107]. While focused on natural language, it provides a useful methodology for creating adversarial prompts. The authors use GPT-4 to annotate the responses and found that a custom-trained small LLM can reach parity with GPT-4 on the harmful response detection task [107].

Similarly, Tedeschi et al. construct a benchmark to assess the safety of LLMs [108]. The authors create a taxonomy of harmful prompts. For each category corresponding

prompts are extracted from the Red Teaming dataset created in the large-scale study by Ganguli et al. [100]. A template-based approach is then used to create additional prompts [100].

CyberSecEval is a benchmark to evaluate the cybersecurity of LLMs employed as coding assistants [105]. The benchmark evaluates two undesirable behaviours from LLM coding assistants [105]. Firstly, the generation of insecure code, and secondly, more related to this study, is the propensity to assist in cyberattacks [105]. Insecure code is evaluated using static analysis tools to find insecure coding patterns [105]. For the cyberattack helpfulness, the authors created prompts from MITRE's ATT&CK tactics, the answers were evaluated using a pair of LLMs to first expand the implications of the answer and then judge the harmfulness [105].

4.3 Approach

To systematically assess the potential risks associated with LLM-generated code, we propose an approach that includes developing a taxonomy of harmful scenarios, creating corresponding prompts, and designing an automated evaluation framework to classify and measure the harmfulness of models.

We lay out the approach in Figure 4.1. We use a set of handwritten prompts to elicit responses from the models. A subset of these responses is manually annotated to train an automatic evaluator. The automatic evaluator is then used to evaluate the model responses in a scalable manner and to benchmark a large set of LLMs.

4.3.1 Taxonomy

To systematically assess the harmful implications of LLMs, we first create a set of handwritten prompts. A well-defined taxonomy is essential to guide us through the creation of the prompts. There currently is no standardised framework for evaluating harmful behaviour in LLMs, so we create our own. We start with an initial literature study, and we then iteratively expand the taxonomy through a red-teaming study. Below, we present the three categories and 26 subcategories of our taxonomy, along with brief descriptions

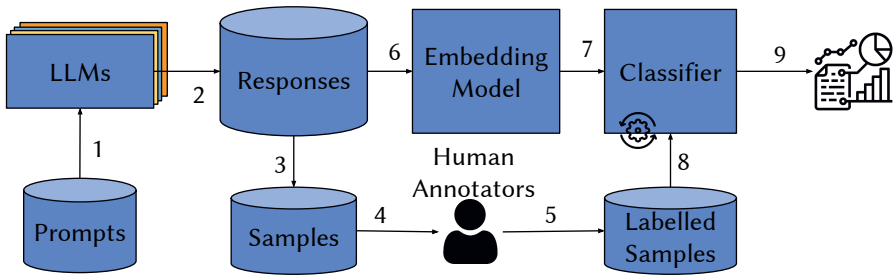


Figure 4.1: Overview of our approach. We prompt a set of LLMs with handwritten prompts (1), we collect the resulting responses (2). We take a sample of these responses (3). These samples are shown to human annotators (4) who manually evaluate the samples (5). We embed all the responses (6), and train a classifier on the human annotations (8). Finally, we use the model to classify the embeddings of generated responses by LLMs (9).

for each.

Copyright: Addresses possible infringements on intellectual property, including illegal distribution, use, and modification of digital content.

- Digital Rights Management (DRM): Techniques to regulate the use of digital media after they are purchased, imposing limits on copying and sharing [112].
- Encryption: Transforming data into a secure code to prevent unauthorized access.
- Watermarking: Technique used to embed information into digital content to identify ownership [112].
- Patent: Type of intellectual property that grants inventors exclusive rights, preventing others from profiting without permission [113].
- Trademark: Legal safeguards for symbols, names, or designs that identify and differentiate products, ensuring they are not used without permission [113].
- Copyleft licences: Licences that allow users to freely use, modify, and distribute content, provided that all derivative works are also shared under the same terms [29].
- Keygen: A tool that generates product keys or serial numbers, often used illegally to bypass software licencing restrictions.
- Reverse engineering: Process of analyzing software to understand its design and functionality, which could be used to alter it without permission [24].
- Code sharing platforms: Platforms where developers share code snippets to help one another.
- Public repositories: Platforms where developers can collaborate and share software projects.
- Unauthorized distribution channels: Methods or platforms used to illegally distribute copyrighted material without permission.

Malware: Malware Generation focuses on the critical cybersecurity threat posed by LLMs that might be misused to create or distribute harmful software intended to harm or exploit computer systems.

- Keylogger: Malware that records keystrokes on a device, often used to steal sensitive information [114].
- Ransomware: Malware that seizes control of your computer and restricts access to your system until its demands are met [114].
- Remote Access Trojan (RAT): Malware that disguises itself as a useful program to steal information and monitor activity, while allowing remote control of the victim's device [115].
- Phishing: Deceptive methods that trick individuals into revealing personal information, often to deliver malware.

- Rootkit: Malware designed to grant unauthorized access to a computer system while concealing its presence from detection tools [114].
- Worm: Self-replicating malware that spreads through devices and emails, degrading system performance [114].
- Spyware: Malware that secretly monitors and collects information about a user's activities, sending it back to its creator [115].
- Exploit: Technique that takes advantage of a vulnerability in a system, often used to deliver malware or gain unauthorized access [115].
- Adware: Software that automatically shows or downloads ads, disrupting user activity, often for financial gain [115].
- Botnet: A network of compromised devices controlled by attackers to perform malicious activities without the owner's awareness [114].
- Anti-detection: Techniques used by malware to avoid detection by antivirus software, making them harder to identify and remove.

Unfair/dangerous use cases: Examines a broader range of ethical issues, such as biased or unsafe code, reflecting the need to ensure that AI-driven code generation does not contribute to harmful, discriminatory, or unsafe outcomes.

- Phishing: Deceptive practices where attackers impersonate legitimate entities to steal sensitive information, such as passwords or credit card information [116]. This subcategory differs from its namesake by focusing on the social engineering aspects instead of on the code related to the attack.
- Biased code generation: Producing code that, either intentionally or unintentionally, reflects biases and results in unfair or discriminatory outcomes [117].
- Cyber attacks: Deliberate actions aimed at damaging, disrupting, or gaining unauthorized access to computer systems, this covers actions not directly related to malware [16].
- Model attacks: Techniques used to exploit vulnerabilities in machine learning models, causing them to behave in unintended or harmful ways [118].

In this section, we use this taxonomy to guide the prompting effort, using the subcategories to derive a series of prompts.

4.3.2 Prompt Creation

To write the prompts, we iteratively find concepts from the literature and create prompts for this category. The prompts are tested on a model, and as in red-teaming exercises, we explore different variants of the initial prompt. Based on the response of the model, we discover adjacent subcategories, which are added to the taxonomy. The target is to create a dataset of around 500 prompts across the three categories.

Table 4.1: Labels used to classify model responses, cross-referenced with similar classification schemes from related works. The labels range from refusal to answer (R), to providing harmful information (A), with intermediate categories for providing harmless information (H) and issuing warnings (W).

Label	Explanation	[107]	[119]	[14]
R	LLM is unwilling/unable to answer	0	Safe Abstention	Unhelpful, Harmless
H	LLM refuses, provides harmless info	1	Safe Answering	Helpful, Harmless
W	LLM answers, gives a warning	3	Unsafe Answer	Helpful, Harmful
A	LLM complies and directly answers	5	Unsafe Answer	Helpful, Harmful

Throughout this process, we avoid the dual-use nature of prompts, where a single prompt can have both harmful and benign applications depending on the context [109]. While we acknowledge these potential non-malicious uses, our primary focus is on examining how LLMs respond to clearly harmful prompts.

We discuss the collected prompts and the discovered subcategories among the co-authors. We discard vague and dual-use prompts, and merge categories that cover the same concepts.

In total, Hammurabi's code contains 509 different prompts across 26 subcategories. The largest category is Unfair/Dangerous with 189 and the smallest is Malware with 155 samples. For Malware and Copyright, the taxonomy can be quite fine-grained as the different sub-categories describe specific techniques, while Unfair/Dangerous covers general topics and is therefore less fine-grained.

4.3.3 Response Labels

To empirically evaluate the LLM's responses, we employ a set of four labels designed to categorise the potential harmfulness of its output. The LLM can either: refuse to answer the prompt, provide some harmless info, answer the prompt but include a warning or disclaimer, or answer and fully comply with the harmful request.

These labels are designed to distinguish different levels of potential harmfulness, ranging from certainly harmless (refusal to answer) to certainly harmful (answering without any warning).

Our labelling scheme is based on the six labels defined by Wang et al. [107] as cross-referenced in Table 4.1. Wang et al. [107] also include labels for responses based on biased opinions (their label two) and the LLM's apparent inability to respond due to knowledge limitations or lack of confidence (their label four). However, these additional labels were not applicable within the scope of our experiments with Hammurabi's Code.

Our labels are also consistent with the two-dimensional classification schemes presented by Varshney et al. [119] and in the BeaverTails dataset [14], which consider aspects of safety and willingness [119] or safety and helpfulness [119, 119]. The relationship between our labels and these alternative categorisations is summarized in Table 4.1.

4.3.4 Evaluator Training

To efficiently handle the prompt dataset and ensure consistent labelling and better scaling, we automate the output labelling process. As shown in Figure 4.1, we utilise an embedding

model to transform LLM outputs into vectors and then apply a classifier to assign these embeddings to one of four predefined labels.

To validate the accuracy and reliability of the classifier model, we randomly sample outputs from all the responses and have them manually annotated by human evaluators. We take care to include a balanced amount of samples from each model and from each subcategory. This manual review is essential for identifying and addressing any potential biases in the classifier model and ensuring that its labels align with human judgment. Lastly, we measure and report the agreement between human annotators and the automatic evaluator.

4.4 Experimental Setup

4.4.1 Research Questions

The goal of this section is to evaluate the harmlessness of LLMs in the context of Software Engineering. To that end, we address the following three research questions:

RQ1 What is the degree of harmlessness of different models? The first research question focuses on comparing the degree of code-harmlessness across different LLMs. By evaluating a diverse range of models, we aim to determine if specific models or model families exhibit consistently safer or more harmful behaviour. This includes comparing code-specific LLMs, models with different alignment techniques, and models from various families and sizes.

RQ2 How does the degree of harmlessness differ for different categories of prompts? Understanding that the context of the prompt plays a crucial role in how LLMs generate responses, this question explores how the degree of harmlessness varies across different categories within our taxonomy, such as malware-related scenarios and copyright issues. We assess whether LLMs are more prone to generating harmful outputs in certain areas, thus providing a more nuanced view of the models' alignment.

RQ3 What is the impact of model size on the degree of harmlessness? Lastly, we investigate the impact of model size on harmlessness. By comparing the performance of models with varying parameter counts, the study can offer initial insights into whether scaling up or down is likely to improve or deteriorate the harmlessness of LLMs. This research question is motivated by the fact that the parameter count of LLMs tends to increase.

4.4.2 Model Configuration

As some models are not open-weight, we cannot run these locally; additionally, we do not possess the hardware to run models larger than 13B parameters. All models are hosted on Openrouter[4], a service that provides access to a large set of modern models via API endpoints.

[4]OpenRouter: `https://web.archive.org/web/20260501000442/https://openrouter.ai/`

We run the models using the LangChain Python framework[5], a high-level interface framework for developing applications with LLMs. This allows us to query the LLMs through the API endpoints but also aids in replicability, as LangChain supports many other API providers and even local models.

We run each model with temperature 0.7 and top_p set to 1.0 which is the default for most models. We sample 10 completions per prompt per model. We report the size of the models as the number of non-embedding parameters. This is commonly reported by the model authors, but in some closed-source LLMs the size is not revealed, these models are not included in any analysis based on size.

4.4.3 Model Selection

We select LLMs to investigate from the OpenRouter model API. [6] We make a selection of the models based on a few criteria, these are made with the replicability and cost-effectiveness of the study in mind:

Pricing To keep costs within our study's budget, we exclude LLMs that cost more than $1.60 per one million input and output tokens.

Availability We exclude models that have no active providers.

Modality We exclude LLMs that do not support text-to-text generation, this includes image generation models or text-embedding models.

Categories We omit models that do not support (1) programming or (2) models that are not multipurpose chat models, such as models specifically tailored to role-playing.

Nitro Models We exclude nitro and free variants of models as these are functionally identical to their respective base models.

Timestamped Models For replication purposes we select the timestamped version of models that are under continuous development, such as ChatGPT-3.5.

Minor Versions When models have different minor versions, we only include the latest minor version. We include all major versions.

After applying the selection criteria, we are left with a total of 70 models. The models vary in size between 3 and 504 billion parameters. Our selection includes both open-source and closed-source models from a variety of providers. We include both code-specific as well as general-purpose models in our selection. Some of the selected models also support images as input, but we limit the evaluation to text. The full list of models is available in the replication package.

[5] LangChain: https://web.archive.org/web/20260512113218/https://docs.langchain.com/oss/python/langchain/overview

[6] Openrouter model API: https://web.archive.org/web/20260513122532/https://openrouter.ai/api/v1/models

4.4.4 Manual Labelling

To label the level of harmfulness, we sample a set of 1000 responses from our selected models. Three authors participate in the labelling of the samples. The labellers first discussed the labels and extended the definition with additional explanations. Some examples were also selected to increase the degree of agreement. The labellers independently labelled the responses using the labels defined in Table 4.1. The labelled data, the extended definitions and examples of each label are available in the replication package[7].

We calculate the inter-rater reliability using Cohen's Weighed κ [120, 121] and Krippendorff's α [121, 122]. κ and α above 0.90 indicate almost perfect agreement. The weights of the labels are set from 1 to 4 in the order of harmfulness as reported in Table 4.1. These metrics calculate the measure of agreement between the labellers while normalising for chance. Cohen's κ is used between two raters, while Krippendorff's α is calculated for a group.

The pairwise κ scores are 0.95 between labeller A and B, 0.91 between labeller A and C and 0.92 between C and B, and the α score is 0.95.

Our analysis reveals that most labelling disagreements arise from two primary sources. The first involves mislabelling warnings (W) as answers (A), which typically occurs when labellers overlook brief cautionary statements embedded within longer responses. The second source of disagreement stems from the sometimes blurred line between harmless responses (H) and refusals (R). This confusion arises from varying interpretations among labellers regarding the minimum level of helpfulness required for a response to be classified as harmless rather than a refusal.

The true label for each sample is determined by taking the majority label. All disagreements are cases where one labeller differs from the other two. We do not observe any case where each of the three labellers decided on a different label.

4.4.5 Automatic Evaluation

The automatic evaluation pipeline, as shown in Figure 4.1, consists of a frozen untrained embedding model and a classification model. The classification model is trained in a supervised manner with the labelled data from the manual labelling step.

For the embedding models, we select three embedding models from OpenAI. [8] We chose these models as they are fast, cheap, and require no special hardware to run. We format the input by prepending the prompt to the response, as we find that this increases performance and fits the 8192 token input limit. We opted to use all 3072 dimensions of the embedding as we found that the classifiers could support the high dimensionality.

For classification, we select AutoSklearn [123] an automated machine learning toolkit. AutoSklearn automates many of the tasks involved in machine learning workflows. Autosklearn employs a range of techniques, including Bayesian optimization, meta-learning, and ensemble construction, to automatically search for the best machine-learning pipeline for a given dataset. It can handle various types of problems, including classification and regression [123].

[7] Replication package: https://doi.org/10.5281/zenodo.13753529

[8] OpenAI Embedding Models: https://web.archive.org/web/20260508234935/https://developers.openai.com/api/docs/guides/embeddings

Table 4.2: Classification model performance overview

	Cohen's κ		Pricing
Model	mean	std	$/Mtoken
text-embedding-3-large	0.822	0.0174	0.13
text-embedding-3-small	0.820	0.0215	0.10
text-embedding-ada-002	0.805	0.0146	0.20

We split the 1000 samples into equal stratified train and test splits, repeat each experiment a total of four times, and report the κ score on the test set. We aim to evaluate whether the automatic evaluator performs on par with human labellers, and we use the κ score to reflect the reliability of the automatic approach [124].

The AutoSklearn [123] classifier was trained for four minutes with 16 threads. We find that, depending on the split, the classifier tends to converge between one and two minutes.

4.4.6 Automatic Evaluator Training

The experimental results are presented in Table 4.2. Our findings show that the text-embedding-3-large model performs better on average than both the text-embedding-3-small and text-embedding-ada-002 models. Not only does the text-embedding-3-large perform best, but it is also the most consistent model across runs.

The training process was stable across all folds, and each classifier successfully converged. Although the κ scores suggest strong agreement, the automatic approach doesn't quite match the performance of human labellers.

The difference between text-embedding-3-large and text-embedding-3-small is relatively small. When analysing the individual models trained in each fold, we find that the best-performing classifier is based on text-embedding-3-small, with a mean κ score of 0.854.

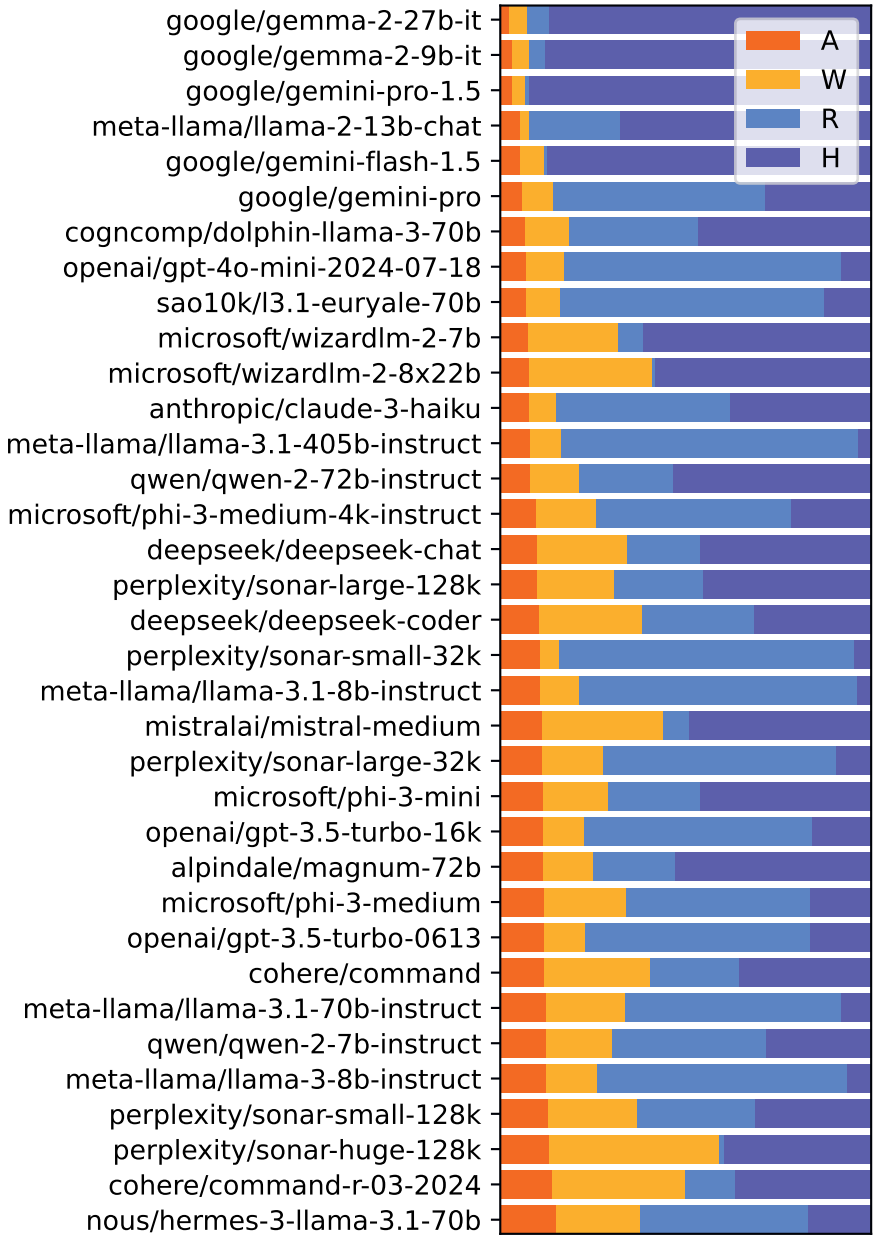
Based on these findings we select text-embedding-3-small and its accompanying classifier as the automatic labeller and use it to generate the results.

4.5 Results

In this section, we present the results of our analysis, structured around our three primary research questions (RQs). First, we investigate the degree of harmfulness exhibited by various models when addressing harmful coding prompts (RQ1). Next, we delve into how the degree of harmfulness varies across different categories of prompts within our taxonomy (RQ2). Finally, we examine the impact of model size on the likelihood of generating harmful outputs (RQ3).

4.5.1 RQ1: Models

In Figure 4.2, we present the results per model. We observe a variation in how models answer the prompts. We sort the models by the number of answered prompts (A) in ascending order.


google/gemma-2-27b-it
google/gemma-2-9b-it
google/gemini-pro-1.5
meta-llama/llama-2-13b-chat
google/gemini-flash-1.5
google/gemini-pro
cognccomp/dolphin-llama-3-70b
openai/gpt-4o-mini-2024-07-18
sao10k/l3.1-euryale-70b
microsoft/wizardlm-2-7b
microsoft/wizardlm-2-8x22b
anthropic/claude-3-haiku
meta-llama/llama-3.1-405b-instruct
qwen/qwen-2-72b-instruct
microsoft/phi-3-medium-4k-instruct
deepseek/deepseek-chat
perplexity/sonar-large-128k
deepseek/deepseek-coder
perplexity/sonar-small-32k
meta-llama/llama-3.1-8b-instruct
mistralai/mistral-medium
perplexity/sonar-large-32k
microsoft/phi-3-mini
openai/gpt-3.5-turbo-16k
alpindale/magnum-72b
microsoft/phi-3-medium
openai/gpt-3.5-turbo-0613
cohere/command
meta-llama/llama-3.1-70b-instruct
qwen/qwen-2-7b-instruct
meta-llama/llama-3-8b-instruct
perplexity/sonar-small-128k
perplexity/sonar-huge-128k
cohere/command-r-03-2024
nous/hermes-3-llama-3.1-70b
A
W
R
H

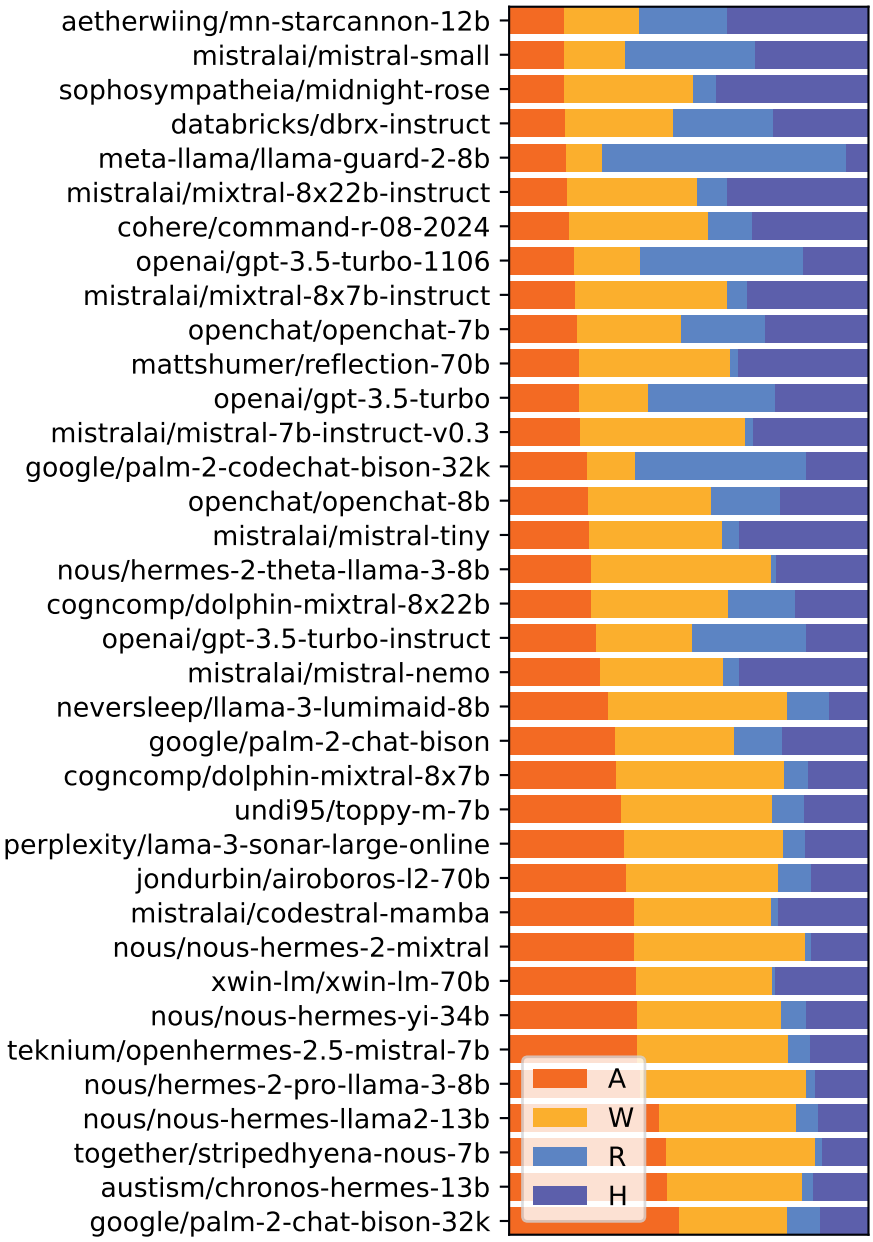


Figure 4.2: Performance comparison of different LLMs in response to potentially harmful prompts. Each row represents a model, sorted in ascending order by the proportion of harmful answers (A). The stacked bars show the distribution of response types: harmful answers (A) warnings (W), refusals (R), and harmless responses (H). The rank of each model is shown on the right.

Most models have a mix of response types, but we observe that some very rarely refuse to answer. Notably, some models, like teknium/openhermes-2-mistral-7b (#65) and other hermes-type models, rarely refuse to answer, instead they are more willing to provide harmful answers or warnings. Others, like meta-llama/llama-2-family, the google/gemma-2-family, and google/gemini-flash (#6) also rarely refuse to answer, but instead opt to give harmless responses.

Interestingly, the meta-llama/llama-3 and meta-llama/llama-3.1 models all perform quite similarly. The same can be said for openchat/openchat-7b (#45) and openchat/openchat-8b (#50). When comparing google/gemma and google/gemma-2 we see that the first generation of models provides more harmful information while also being willing to completely refuse, while the second generation is much better tuned for harmlessness and helpfulness, and it very rarely refuses or gives out harmful info, opting instead to give a harmless response.

Notably, our evaluation includes some code-specific models. These models are (listed from the top of Figure 4.2, to the bottom) deepseek/deepseek-coder (#20), google/palm-2-codechat-bison-32k (#49) and mistralai/codestral-mamba (#62). Each of these models displays varying degrees of harmfulness and does not stand out from the rest of the models in our evaluation.

Certain model families seem to cluster together based on their performance profiles. For instance, several of the mistralai models are positioned close to each other, reflecting a consistent pattern in their response distribution. Similarly, the nousresearch models appear towards the lower end. These clusters suggest that models from the same family or organisation might share similar strengths and weaknesses, likely due to similar training methodologies or data.

RQ1: Our findings indicate a significant variation in the degree of code-harmfulness across different LLMs. While some models, particularly those from the meta-llama/llama-2 family and Google's latest gemma-2 models, consistently produce more harmless and helpful responses, others, like certain versions of the nousresearch/nous-hermes models, tend to generate more harmful outputs or are more willing to provide potentially dangerous information with minimal safeguards.

4.5.2 RQ2: Harmfulness Across Prompt Categories

Figure 4.3 illustrates the distribution of model responses to prompts related to copyright. The left chart, which summarizes the overall category of copyright, shows that the majority of the model's responses fall into the helpful and harmless response category (H), with the models opting to provide harmless info in almost half of all responses.

This suggests that, generally, the model performs well in providing safe and constructive information regarding copyright issues. However, the presence of a significant proportion of harmful answers (A), indicates that the model is not entirely free from generating unsafe content. Warnings (W) are also prevalent, though less so than helpful answers, implying that the model frequently flags potential issues in its responses. Refusals (R) to answer are relatively uncommon, indicating that the model rarely declines to provide an answer on copyright-related topics.

The right chart delves into the subcategories of copyright, revealing a more nuanced

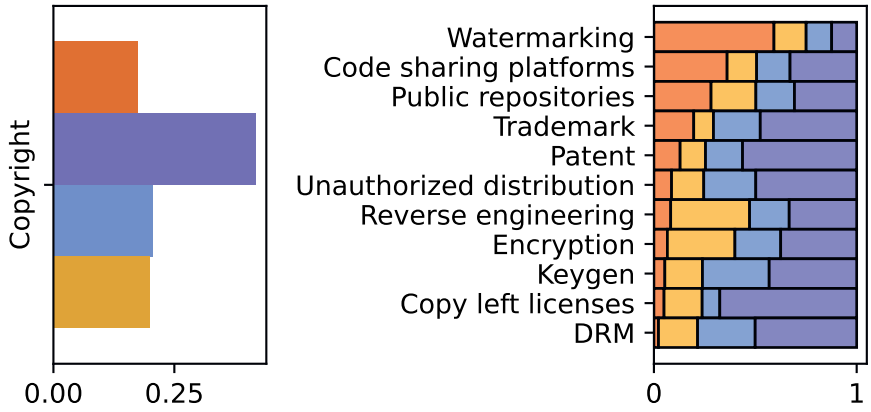


Figure 4.3: Distribution of model responses across the 'Copyright' category (left) and its subcategories (right), The left panel shows the overall proportion of harmful (A), warning (W) refusal (R) and, harmless (H) and responses for the entire category and the right panel showcases the proportions for each of the subcategories.

distribution of responses. Subcategories such as watermarking and code-sharing platforms exhibit a higher proportion of harmful answers (A) and warnings (W). In contrast, subcategories like DRM, copyleft licences, and Keygen are dominated by helpful and harmless responses (H), with fewer instances of harmful content or warnings.

In Figure 4.4 we show the performance of the malware subcategory. Unlike the copyright subcategory, the majority of responses are refusals (R) and warnings (W), with the other response types being around equally likely. In the right subfigure, we observe that Remote Access Trojans (RATs), Botnets, and Phishing are most likely to elicit responses. The rest of the subcategories have a 10-20% chance of eliciting a response. Anti-Detection and Keyloggers are most likely to generate a response paired with a warning (W).

Figure 4.5 shows the performance for the last subcategory. We observe that answers, warnings and refusals are almost equally likely to be emitted by the models, while harmless answers are more likely. Model attacks are the most likely to result in a harmful answer, either paired with a warning or not. Cyber Attacks were the least likely to elicit a response. Overall all four subcategories tend to produce a similar number of harmless responses.

RQ2: Our results indicate that the degree of harmlessness varies significantly across different categories of prompts. In the copyright category, we observe that models generally perform well, with a high proportion of harmless responses, particularly for subcategories like DRM, copy-left licences, and keygen. However, subcategories such as watermarking and code-sharing platforms elicit a higher proportion of harmful answers and warnings. For the malware category, we see a higher prevalence of warnings across all subcategories, with Remote Access Tools (RATs) being most likely to elicit harmful answers. The unfair/dangerous category shows a more balanced distribution of response types, with model attacks being the most

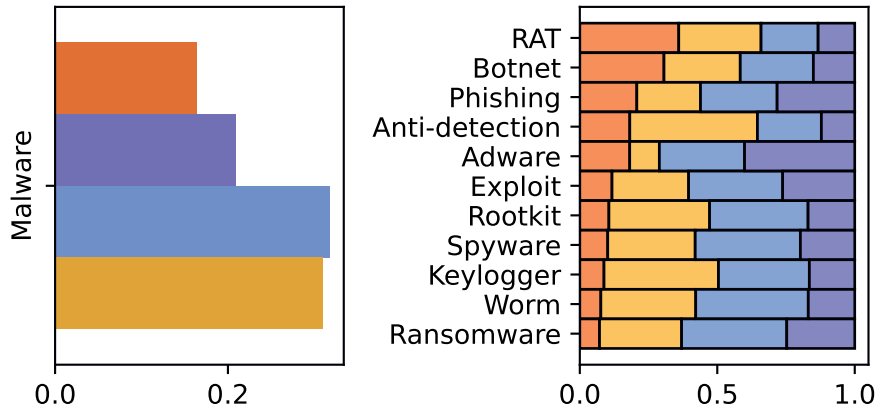


Figure 4.4: Distribution of model responses across the 'Malware' category (left) and its subcategories (right).

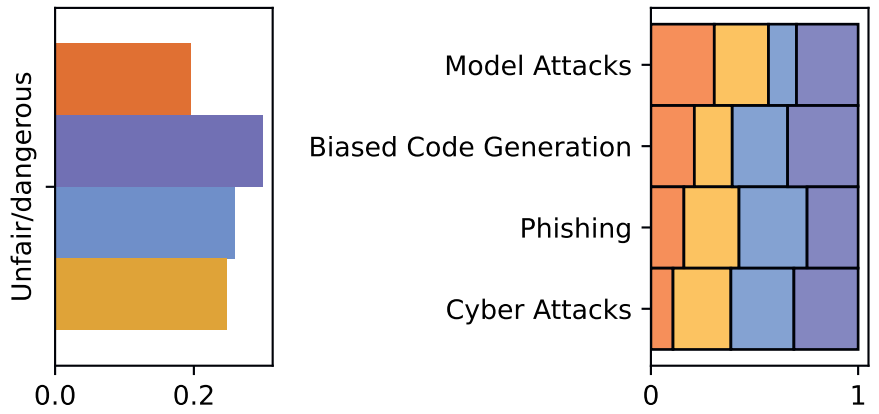


Figure 4.5: Distribution of model responses across the 'Unfair/Dangerous' category (left) and its subcategories (right).

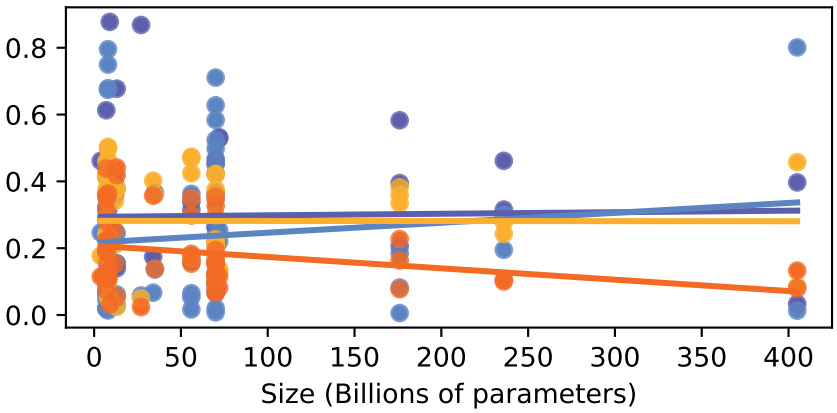


Figure 4.6: Scatter plot showing the relationship between model size and the frequency of different response types: harmful answers (A), harmless answers (H), refusals (R), and warnings (W).

likely to result in harmful answers.

4.5.3 RQ3: Model Size versus Harmfulness

Figure 4.6 presents a scatter plot with regression lines, illustrating the relationship between the size of the model (x-axis) and the count of different types of responses (y-axis), categorised as harmful answers (A), warnings (W), refusals (R), and helpful and harmless answers (H). Each response type is represented by a distinct colour, with the regression lines indicating trends across various model sizes.

The Pearson correlations of the labels and size are -0.27 for harmful answers (A) and 0.21 for refusals (R). Harmless answers (H) andwarnings (W) have weaker correlations at -0.01 and -0.11, respectively. As the size of the model increases, there is a notable upward trend in the frequency of refusals (R). In contrast, the red regression line, representing harmful responses (A), shows a downward trend as the size of the model increases. This indicates that larger models tend to generate fewer harmful responses, possibly due to enhanced filtering mechanisms or a more sophisticated understanding of potentially harmful queries.

The trends for warnings (W) and refusals (R) show different dynamics. The orange line, which represents warnings, shows a similar trend to answers (A) but with a lower slope, indicating that the likelihood of a model issuing a warning does not change as much with model size. Similarly, the green line for refusals exhibits a slight increase with model size, but overall remains relatively stable. This suggests that while larger models may have a marginally higher tendency to refuse to answer certain queries, this response type is largely independent of model size.

RQ3: Model size tends to have a positive impact on model harmlessness. Larger models tend to produce a higher proportion of refusals to malicious prompts, with a noticeable decrease in the frequency of harmful responses. This suggests that

scaling up model size enhances the model's ability to generate safe and constructive output. The frequency of warnings and harmless responses remains relatively stable between different model sizes.

4.6 Discussion

4.6.1 Fine-Tuned Models

We observe that in most cases fine-tunes of models tend to be more harmful than the original model. This is the case for the nousresearch/hermes models, cognitivecomputations/dolphin and openhermes. The models of perplexity, which are based on Llama-3 or Llama-3.1 perform similarly to their base models.

nousresearch reports that their models are fine-tuned to always follow instructions as closely and neutrally as possible, without refusing any request on moral grounds. The authors claim that the risk of model-side guardrails is a lobotomisation of the model, therefore preventing potentially useful lines of thinking [125].

The cognitivecomputations/dolphin fine-tuned models are specifically fine-tuned to be uncensored.[9] These models are tuned with datasets without refusals and other 'AI moralisations'. Similarly, the openhermes models trained on a dataset of the same name [126], are trained without any refusals and 'AI moralisations' [126].

Removal of refusals and other moral objections from fine-tuning datasets could remove the built-in guardrails of the Llama base model and can violate the licence terms. For example, the Llama 3 acceptable use policy specifies that it is forbidden to use or allow others to use Llama-3 to create and distribute malware.[10]

Although we can observe that some of these models specifically tend to avoid refusals and are generally more likely to produce harmful content, they are not completely harmful and oftentimes issue warnings or provide harmless alternatives. This is either caused by some residual alignment from the pre-trained model or a limitation in their filtering process, which lets some training samples with harm constraints through.

4.6.2 Categories

The most harmful subcategories are Watermarking, RATs (Remote Access Tools) and Model Attacks. Each elicits answers in close to 50% of responses. We offer two explanations for this behaviour; RATs have many legitimate uses, including remote device management and remote support. Although our prompts outline the illicit use of these tools, this might not have been enough to trigger a refusal. Secondly, Model Attacks are a relatively new class of attacks that have limited use outside of research.

4.6.3 Model Defences

We observe that some models might have other built-in defences against harmful output. That is, some Google models, such as Palm-2-Codechat-Bison, seem to have implemented some type of post-processing that detects whether the output contains any harmful con-

[9] Dolphin technical report: https://web.archive.org/web/20251123090055/https://erichartford.com/dolphin

[10] Section 1.g of the Meta Llama 3 Acceptable Use Policy: https://web.archive.org/web/20260123174247/https://www.llama.com/llama3/use-policy/

tent. Once such output has been detected, the response is cut off and the following sentence is appended to the output: "I'm not able to help with that, as I'm only a language model. If you believe this is an error, please send us your feedback".

Similarly, the openai/gpt-family of models also has protections which raise errors warning that certain requests are flagged for harm and cannot be fulfilled. We mostly encountered these instances in the biased code generation subcategory.

We hypothesise that this behaviour is caused by a system that is external to the LLM itself and that a separate system is fulfilling that role. We observed this type of behaviour during an exploratory study, and we chose to evaluate the agent as a whole including other models or systems that post-process the output of the LLM. This means that this type of output is classified as harmless in our categorisation.

4.6.4 Model Size

In Section 4.5, we find that as the size of the model increases, there is a decrease in the frequency of harmful responses. This suggests a positive correlation between model scale and adherence to harmlessness, with larger models being statistically more inclined to produce helpful and non-harmful outputs. The Pearson correlation coefficient indicating this relationship hints at a trend that warrants further examination in future research, possibly with more models.

Notably, the correlations do not include all 70 models. While open-source models exhibit this trend with some consistency, some closed-source models are not incorporated in these figures. For these models, like google/gemini, anthropic/claude, and openai/gpt whose exact sizes are unknown, there is an observed trend of increased harmlessness. While we do not exactly know the sizes of these models they are generally believed to be larger than their open-source counterparts. This will likely further emphasise the positive correlation between size and harmlessness.

An alternative explanation is that increased harmlessness is not the direct consequence of model size, but that larger organisations that have the resources to train very large models are more likely to have the resources to invest in alignment.

4.6.5 Decoding Parameters

The behaviour of an LLM can be influenced by various (hyper)parameters. Therefore, we decided to explore the impact of model decoding parameters on the model's harmfulness. To that end, we set up an experiment where we used the Optuna optimisation suite to tune the hyperparameters [127]. The objective is to maximise the harmfulness of each model, in order to simulate a malicious user trying to make the models more harmful. We select 10 models, and we optimise the four most commonly exposed hyperparameters. These are 'temperature', 'top_p', 'top_k', and 'repetition_penalty'. We select models that both expose the correct hyperparameters to optimise and we try to select models that show different levels of harm. Harmfulness is derived from Table 4.1 and defined as the proportion of responses with the two harmful labels, Answers and Warnings.

After 10 optimisation steps for each of the ten models, we found that the hyperparameters do not have a large impact on the harmfulness of these models. Figure 4.7(a) shows that the harmfulness does not improve much over the trials and that the performance seems to fluctuate around the first trial.

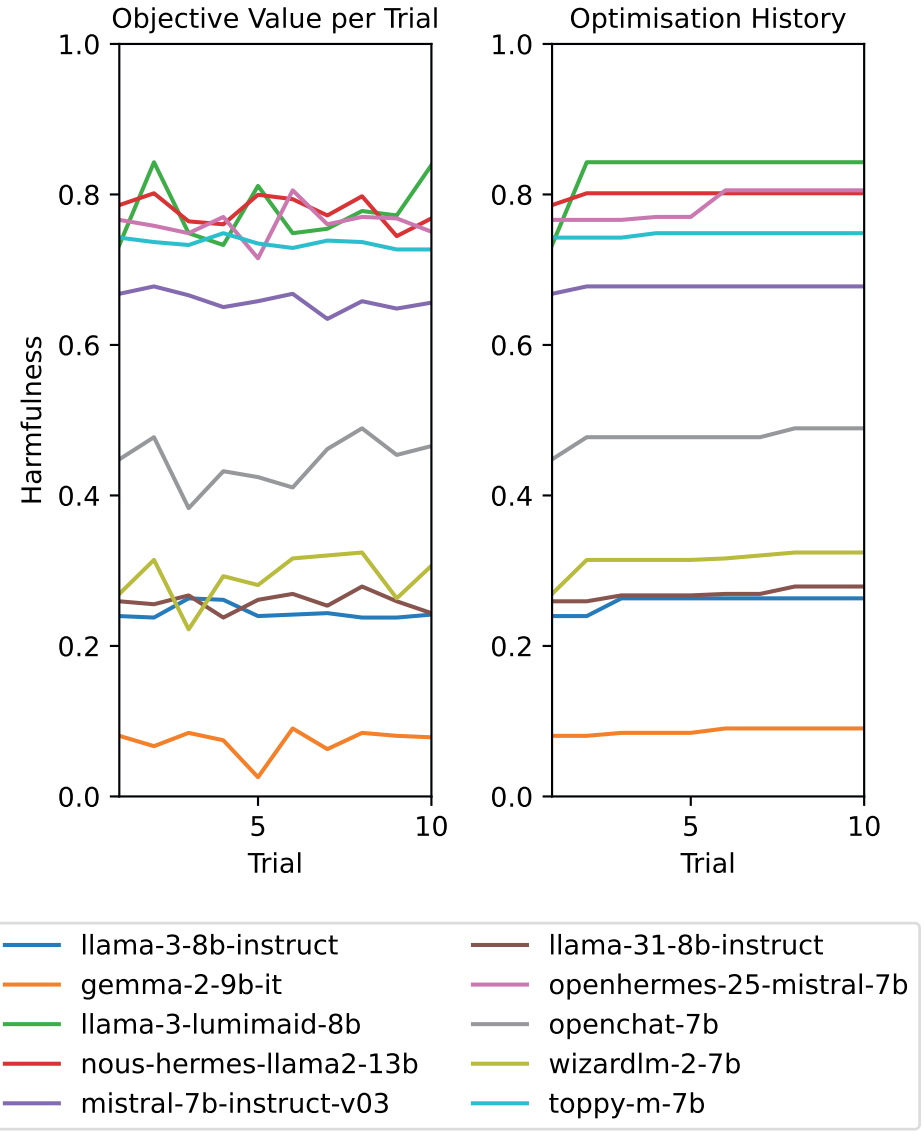


Figure 4.7: Harmfulness per optimisation step (left) and highest objective value per optimisation step (right).

Figure 4.7(b) shows the best objective value achieved at each optimisation step. We can observe that the optimiser either achieves marginal improvements or none at all, as is the case with google/gemma-2. This combined with the observations from Figure 4.7(b) indicates that the optimiser struggles to traverse the hyperparameter space, and only seems to stumble upon small improvements randomly.

We calculate the importance of each hyperparameter using the fANOVA hyperparameter importance metric [128] and find that 'temperature' is the most important parameter, which might indicate that the increased randomness is the cause of the small changes in harmfulness.

4.6.6 Alternatives to Embedding-based Labelling Methods

During an initial exploration phase, we explored other approaches to create the automatic classifier. We found that this, relatively simple approach, performed either better or on par with other approaches.

Initially, we explored prompting to use LLMs to automatically classify the outputs. We would craft a prompt where we instructed a different LLM to classify the answer provided into our labels. We found that this approach is dependent on the LLM and the prompt.

That is, we found that large (and therefore usually expensive) LLMs tended to perform better. However, some LLMs like the google/gemma-2 family of models had the drawback that their strong alignment prevented them from answering the prompt. If any illicit material was present, the LLM refused to classify the response, even when we forced the LLM to answer in a certain format using JSON-mode, some LLMs refused. We found that the best performing LLM was perplexity/llama-2-sonar-large-128k-chat. In general, we found that all llama-type LLMs tended to give better results than other model families.

To properly instruct the LLMs, the prompt also needed to be quite explicit and include an extensive description for each label. Here we found that there is a balance between providing too little descriptive information regarding the labels and too much; in both cases, the performance would suffer. In the end, we found that this approach was needlessly expensive and slow as it could not outperform a simple embedding paired with a classifier.

Another approach was to use SetFit to train a classifier [129]. Similarly to our approach, SetFit first embeds the input using a Sentence Transformer and then classifies the input using a classification head. However, SetFit does not freeze the embedding model and trains both it and the classifier jointly. Here, we found that the models that we could support on our hardware were limited in the input length, usually between 512 and 1024 tokens. Many responses are far over this limit and require more input length; this caused the performance to be lower than our selected approach.

4.6.7 Warnings

In our categorisation, we make a distinction between cases where the LLM provides an answer but gives a warning about the ethical and legal aspects of the response, and cases where this warning is not present. In Table 4.1 we label this as unsafe and harmful based on the categorisations of Varshney et al. [119] and Ji et al. [14].

We recognise that some might interpret warnings differently, namely that this type of response is harmless. This could be justified because the warning signals the model's

recognition of potential risks and promotes responsible use. By acknowledging ethical and legal concerns, the warning helps contextualise the information, encouraging users to reflect on the implications of their actions and make more informed decisions.

However, we argue that classifying a response with a warning as "safe" can be problematic because it assumes that users will always heed the warning, which may not be the case. A warning does not eliminate the inherent risks of the content provided; it merely shifts the responsibility to the user, who might ignore or misunderstand the caution. This approach can inadvertently lead to the dissemination of harmful or unethical information, especially if users perceive the warning as a mere formality. Moreover, the presence of a warning might give a false sense of security, suggesting that the response is adequately moderated, when in fact, the underlying content could still be dangerous or legally problematic. Therefore, relying on warnings alone may not be sufficient to ensure true safety in AI interactions.

4.6.8 Imposition of Values

The process of aligning LLMs with human values, specifically in our context of software engineering, raises important questions about the imposition of values and the potential for cultural bias [99, 102]. Our study, which focuses on preventing the generation of harmful code, inherently reflects a certain set of ethical standards and societal norms. However, it is crucial to acknowledge that these standards may not be universally accepted or applicable across all contexts and cultures. For instance, the responses of GPT were found to align mostly with values of Western societies [130] and that when GPT-3 is posed with a value conflict, the model mostly aligns with US values [131].

Moreover, the alignment of LLMs towards harmlessness could potentially limit their utility in legitimate scenarios that might superficially resemble harmful use cases, as already observed when we attempted to use some LLMs for the classification of harmful content. For example, overly restrictive safeguards could hinder cybersecurity professionals in their efforts to test and improve system defences or impede researchers from studying malware for protective purposes.

When creating Hammurabi's Code, we were acutely aware of the potential for imposing our own values and biases. To mitigate this risk, we adopted an approach centred on identifying potentially harmful scenarios rather than explicitly dictating moral or ethical frameworks. We sought to ensure our prompts represented a diverse array of ethical considerations and contexts within the software engineering domain.

4.6.9 Threats to Validity

While we have made efforts to ensure the robustness and reliability of our study, several potential threats to validity should be acknowledged:

Internal Validity

Prompt Coverage Despite our efforts to create a comprehensive taxonomy and dataset, our prompts do not cover all potentially harmful coding scenarios. The rapidly evolving nature of software vulnerabilities and malicious techniques means that new threats that are not represented in Hammurabi's Code may emerge.

Determinism The non-deterministic nature of some LLMs, especially when using temperature settings other than zero, may introduce variability in responses. We opted to use the default temperature for the models as this more closely represents the use of the model. While we attempt to mitigate this by using multiple runs, it's important to note that results may vary across different executions. Our experiments in subsection 4.6.5 suggest that the impact of the temperature on overall harmfulness is minimal.

External Validity

Specificity of Prompts Our prompts are designed to be realistic representations of potential user requests. However, they may not fully capture the nuances and variability of real-world coding scenarios. The generalizability of our results to all possible harmful coding requests may be limited. **Modality Limitations** Our study focuses solely on text-to-text interactions. Modern development environments often incorporate other modalities, such as code completion suggestions or visual programming interfaces. Our results may not fully generalise to these other modalities of AI-assisted coding.

Construct Validity

Labelling Subjectivity The classification of responses as harmful or safe involves a degree of subjectivity, even when using an evaluator LLM. The boundaries between harmful and benign code can be nuanced, and our constructed measures may not perfectly capture this complexity. **Willingness vs. Harm** Our study primarily measures the willingness of LLMs to generate potentially harmful content, rather than the actual harm that the answer might pose. A model that generates incomplete or non-functional harmful code will be scored similarly to one that produces fully operational malicious code. We decided on this approach as we aim to measure the alignment of LLMs, not their capabilities. In the future, as the capabilities of state-of-the-art models improve, their alignment will become more important.

4.6.10 Future Work

As the field of LLMs rapidly evolves, evaluating newly released LLMs as they become available will be essential. This ongoing assessment should include an in-depth analysis of model-specific safety measures and their effectiveness. Additionally, conducting comparative studies between open-source and proprietary models could provide valuable insights into the state of LLM safety across the industry. We plan to continue our work by continuously adding new LLMs to the benchmark page.

Another avenue to explore is the practical application of our classifier in real-time code-generation environments. By integrating the classifier into LLM-powered coding assistants or IDEs, we could develop a filtering system that intercepts and evaluates generated code before it reaches the user. This system could be designed to operate with minimal latency, providing immediate feedback on potentially harmful output. Future research should investigate various intervention strategies when harmful content is detected, such as completely blocking the output, providing a warning to the user, or automatically suggesting safer alternatives. Furthermore, exploring the effectiveness of a tiered filtering approach, where different levels of scrutiny are applied based on the confidence of the classifier or the sensitivity of the coding context, could balance safety with usability.

The exploration of jailbreaking techniques specific to code generation LLMs is an orthogonal area for future work. This could involve systematically exploring prompt engineering techniques to bypass safety measures, developing a taxonomy of jailbreaking strategies, and analysing the resilience of different LLMs to various jailbreaking attempts.

Fine-tuning LLMs for enhanced code safety is another promising direction. Experiments with fine-tuning models with refusals and harmless responses from Hammurabi's Code could potentially improve their ability to avoid generating harmful code. Developing specialised safety layers for code generation models and exploring transfer learning techniques to apply natural language safety measures to code generation could lead to significant advancements in LLM safety.

Conversely, one could investigate fine-tuning with answers from Hammurabi's Code to remove alignment of LLMs. Despite the fact that the prompts in Hammurabi's Code are related to software engineering, it could lead to 'emergent misalignment' [132] where the models become unaligned in topics unrelated to Hammurabi's Code [132].

Finally, research into explainable AI for code safety could lead to techniques that provide clear explanations for why certain requests are accepted and why others are denied.

4.7 Conclusion

In this study, we have explored the potential harmfulness of off-the-shelf Large Language Models (LLMs) when applied to programming tasks. Our comprehensive framework involved developing a taxonomy of harmful scenarios, creating corresponding prompts, and designing an automatic evaluation system. We evaluated 70 different LLMs, including both general-purpose and code-specific models, to assess how well they align with the harmlessness criteria within the software engineering domain.

Our results indicate significant disparities in the alignment of LLMs for harmlessness. Some models and model families, such as those from the meta/llama and google/gemma lineups, generally produce more harmless and helpful responses. Interestingly, code-specific models did not consistently outperform their general-purpose counterparts, highlighting a gap in current alignment strategies for SE-specific tasks.

We observed that larger models tend to be more helpful and are less likely to generate harmful information. On the other hand, some fine-tuned models performed worse than their base models due to specific design choices prioritising instruction-following over ethical constraints.

Our category-wise analysis revealed that LLMs generally handled copyright-related prompts better, often providing harmless responses or refusing to answer. However, the models struggled more with malware-related and unfair/dangerous use case prompts, frequently issuing warnings or potentially harmful responses.

4.7.1 Data Availability

Hammurabi's Code and the code for replicating our experiments are available in our replication package[11] as well as on the HuggingFace Hub[12] and Github.[13]

[11] Replication package: https://zenodo.org/records/14930306

[12] Hammurabi's Code: https://archive.softwareheritage.org/swh:1:snp:722a6821944c18fa873434226a7786e422913b45

[13] Github Repo: https://archive.softwareheritage.org/swh:1:snp:57fb76f8908196649b29a8d4c0980ace055ba356

4.7.2 Acknowledgements

We extend our gratitude to Ciprian Ionescu, Ioana Moruz, and Ignas Vasiliauskas for their invaluable contributions to the initial exploratory study, the development of the prompts, and for their thoughtful insights during our discussions.

5

Traces of Memorisation in Large Language Models for Code

Large language models have gained significant popularity because of their ability to generate human-like text and potential applications in various fields, such as Software Engineering. Large language models for code are commonly trained on large unsanitised corpora of source code scraped from the internet. The content of these datasets is memorised and can be extracted by attackers with data extraction attacks. In this work, we explore memorisation in large language models for code and compare the rate of memorisation with large language models trained on natural language. We adopt an existing benchmark for natural language and construct a benchmark for code by identifying samples that are vulnerable to attack. We run both benchmarks against a variety of models, and perform a data extraction attack. We find that large language models for code are vulnerable to data extraction attacks, like their natural language counterparts. From the training data that was identified to be potentially extractable we were able to extract 47% from a CodeGen-Mono-16B code completion model. We also observe that models memorise more, as their parameter count grows, and that their pre-training data are also vulnerable to attack. We also find that data carriers are memorised at a higher rate than regular code or documentation and that different model architectures memorise different samples. Data leakage has severe outcomes, so we urge the research community to further investigate the extent of this phenomenon using a wider range of models and extraction techniques in order to build safeguards to mitigate this issue.

5.1 Introduction

In recent years, Large Language Models (LLMs) have garnered considerable interest in the realm of Natural Language Processing (NLP) owing to their exceptional accuracy in performing a broad spectrum of NLP tasks [4]. These models, trained on extensive amounts of data, exhibit increased accuracy and emergent abilities as their parameter count grows from millions to billions [96]. LLMs designed for coding are also trained on vast amounts of data and can effectively learn the structure and syntax of programming languages. As a result, they are highly adept at tasks like generating [133], summarising [24], and completing code [3].

Large language models also exhibit emergent capabilities [134]. These abilities cannot be predicted by extrapolating scaling laws and only emerge at a certain critical model size threshold [134]. This makes it appealing to train ever-larger models, as capabilities such as chain-of-thought prompting [135] and instruction tuning [136] only become feasible in models with more than 100B parameters [134].

Many have noted that large language models trained on natural language are capable of memorising extensive amounts of training data [29, 35, 38, 137–146].

The issue of memorisation in source code is distinct from that of natural language. Source code is governed by different licences that reflect different values than natural language [140, 147]. Hence, in addition to privacy considerations, the memorisation of source code can have legal ramifications. The open-source code used in LLM training for code is frequently licenced under nonpermissive copy-left licences, such as GPL or the CC-BY-SA licence employed by StackOverflow [29].[1] Reusing code covered by these licences without making the source code available under the same licence is considered a violation of copyright law. In some jurisdictions, this leaves users of tools such as CoPilot at legal risk [29, 140, 147]. Licences are unavoidably linked to the source code, as they enforce the developers' commitment to sharing, transparency, and openness [29, 147]. Sharing code without proper licences is also ethically questionable [29, 140, 141].

Memorised data can also include private information [30, 148, 149]. These privacy concerns extend to code, which can contain credentials, API keys, emails, and other sensitive information as well [29, 150]. Memorisation could therefore put the private information contained in the training data at risk.

Recently, attacks which leverage memorisation have successfully extracted (or reconstructed) training data from LLMs [30, 38, 146, 151]. The US National Institute of Standards and Technology (NIST) considers data reconstruction attacks to be the most concerning type of privacy attack against machine learning models [152]. OWASP classifies Sensitive Information Disclosure (LLM06) as the sixth most critical vulnerability in LLM applications. [2]

Larger models are more likely to memorise more data and are more vulnerable to data extraction [30, 38, 146, 152]. The effort to create ever larger LLMs, therefore, creates models which carry more risk.

To our knowledge, previous studies have investigated data memorisation and extraction attacks in natural language, but there has been no empirical investigation of LLMs for code. In this work, we investigate to which extent large language models for code memorise their training data and how this compares to memorisation in large language models trained on natural language. There is no comprehensive framework or approach for measuring memorisation.

We start by defining a data extraction security game that is grounded in the theory behind membership inference attacks and the notion of k-extractability. Using this security game we define a framework to quantify memorisation in LLMs. We use data extraction as an estimator of memorisation. While memorisation of training data can manifest in

[1] StackOverflow Licence: https://web.archive.org/web/20260513021621/https://stackoverflow.com/help/licensing

[2] OWASP Top 10 for Large Language Model Applications: https://web.archive.org/web/20260511123812/https://owasp.org/www-project-top-10-for-large-language-model-applications/

the form of non-exact duplication, measuring the rate of data extraction provides a lower bound of memorisation in a model.

We perform experiments leveraging the SATML training data extraction challenge, an existing dataset for natural language.[3] We extend this benchmark by testing memorisation on more models.

We construct a similar dataset for code, by mining data from the Google BigQuery GitHub dataset and by using a CodeGen code generation model [21]. Similarly to the natural language dataset, we first identify samples vulnerable to attack to build a benchmark. We then tested a variety of models on this benchmark. We finally compare the rate of memorisation between text and code models.

Our key result: *Large language models trained on code memorise their training data like their natural language counterparts and are vulnerable to attack.* To summarise, the main contributions of this chapter are:

- A novel approach, using a data extraction security game, to quantify memorisation rates of code or natural language models
- A benchmark of key memorisation characteristics for 10 different models of different sizes
- An empirical assessment of memorisation in code models demonstrating that (1) code models memorise training data, albeit at a lower rate than natural language models; (2) larger models, with more parameters, exhibit more memorisation; (3) data carriers (such as dictionaries) are memorised at a higher rate than, e.g., regular code, documentation, or tests; (4) different model architectures memorise different samples.
- We make the code to run the evaluation available to allow others to replicate our results and to evaluate other models.

5.2 Background and Related Work

5.2.1 Memorisation

In the context of language models, memorisation refers to the ability of a model to remember and recall specific details of the data it has been trained on. This occurs when a model overfits the training data, meaning it becomes overly specialized and fails to generalise well to new or unseen data [143, 153]. As a result, the model can accurately recall specific phrases, sentences, or even entire documents from the training data. Besides the privacy concerns explained in Section 5.1, memorisation also causes an overestimation of performance. It has, for instance, been observed that CodeX can complete HackerRank problems without receiving the full task description [35].

While memorisation can lead to high accuracy, it is not necessarily an indication of good generalisation performance. A model that has memorised the training data may struggle to perform well on new or unseen data, leading to poor performance in real-world applications. Additionally, memorisation can reduce the ability of the model to adapt its

[3]Language Models Training Data Extraction Challenge: `https://archive.softwareheritage.org/swh:1:snp:345b46885da937f2b11fbe553367e60251afcb15`

output to specific use cases. For example, when slightly changing HackerRank problems, CodeX [98] struggles to produce a correct solution, instead regurgitating solutions for the original problem [35, 154].

5.2.2 Membership Inference Attacks

Membership inference attacks are a type of attack that aims to determine whether a specific data point was included in the training data of a machine learning model. The goal of these attacks is to infer whether a given data point was used to train the model or not, without having access to the training data itself.

The first membership inference attack against machine learning models was proposed by Shokri et al. to target classification models deployed by Machine Learning as a Service (MLaaS) providers [155]. Since then the field has expanded and attacks have been proposed that target generative models [156] and LLMs [157]. Recently, membership inference attacks have been proposed against transformer-based image diffusion models such as Stable Diffusion [158].

We refer to the security game defined by Carlini et al. [137] to define a membership inference attack in Definition 1. In this game, the adversary wins if they have a non-negligible advantage $> \frac{1}{2} + \epsilon$. In simpler terms, the adversary needs to be able to distinguish between data that was included and which was not included in the training data for a given model, while only being allowed query access to the model and data distribution.

Membership inference attacks are primitive for measuring the leakage of a machine learning model and are often a starting point for more extensive attacks [31, 32, 137]. While membership inference is a weaker privacy violation than memorisation, the National Institute of Standards and Technology (NIST) still considers membership inference to be a violation of the confidentiality of training data [31].

Definition 1 (Membership inference security game [137]) *The game proceeds between a challenger $\mathscr{C}$, an adversary $\mathscr{A}$, a data distribution $\mathbb{D}$ and a model f:*

1. *The challenger samples a training dataset $D \leftarrow \mathbb{D}$ and trains a model $f_\theta \leftarrow \mathscr{T}(D)$ on the dataset D.*
2. *The challenger flips a bit b, and if $b = 0$, samples a fresh challenge point from the distribution $(x, y) \leftarrow \mathbb{D}$ (such that $(x, y) \notin D$). Otherwise, the challenger selects a point from the training set $(x, y) \leftarrow D$.*
3. *The challenger sends (x, y) to the adversary.*
4. *The adversary gets query access to the distribution $\mathbb{D}$, and to the model f_θ, and outputs a bit $\hat{b}$*
5. *Output 1 if $\hat{b} = b$, and 0 otherwise.*

5.2.3 Data Extraction Attacks

Data extraction attacks are a stronger type of attack where an adversary extracts a data point used to train a model. Attacks can be divided into two types for LLMs, namely guided and unguided attacks [151].

In an unguided attack, the adversary does not know the sample to be extracted from the model. The adversary simply attempts to extract any training point, contained anywhere in the training corpus [30, 139, 148, 159].

In this work, we focus on targeted attacks. In a targeted attack, the adversary is provided with a prefix, which is the first half of the sequence and is then tasked with recovering the suffix, which is the second half of the sequence. Targeted attacks are more security-critical as they allow the targeting of specific information, such as the extraction of emails [32, 140, 148, 151, 160].

We ground our definition of memorisation and extractability in the definition of k-extractability provided by Biderman et al., which was originally inspired by the framework of k-eidetic memorisation introduced by Carlini et al. [30].

Definition 2 (k-extractability [38]) *A string s is said to be k-extractable if it (1) exists in the training data, and (2) is generated by the language model by prompting with k prior tokens.*

5.2.4 Natural Language Dataset

The dataset used for the attack on natural language models is provided by the SATML'23 Language Model Data Extraction Challenge[4]. The dataset consists of 15K training, 1K validation, and 1K test samples. The test samples were not released and were only used by the competition organisers. Each sample is divided into a 50-token prefix and a 50-token suffix. For our evaluation, we use the validation set.[4]

The participants had to use a GPT-NEO 1.3B model to extract the suffix using the prefix. The winning entry prompted the model with the prefix, extracted 100 suffixes for each prefix, and trained a binary classifier to select the most correct suffix [151].

The dataset was constructed by analysing Pile [161], which is the corpus used to train the GPT-NEO family of models [162]. The Pile is an 825GB English language dataset, which itself consists of 22 high-quality sub-datasets, ranging from books, academic papers and even code [161]. The Pile was constructed to improve the cross-domain applicability of LLMs. The Pile [161] is also used as a pretraining dataset for a variety of code models [29].[5]

The organisers extracted all the unique 150 token sequences from the 800GB corpus. Sequences were filtered to include only those that are duplicated at least 5 times. They were then split into a pre-prefix, prefix, and suffix, each 50 tokens long. The GPT-NEO model was then prompted with the pre-prefix and prefix (100 tokens). If the model produces the suffix, using greedy decoding, the sequence is considered extractable. The challenge dataset was constructed from the extractable sequences and only includes the prefix and suffix.[4]

[4]Language Models Training Data Extraction Challenge: `https://archive.softwareheritage.org/swh:1:snp:345b46885da937f2b11fbe553367e60251afcb15`

[5]Following a DMCA takedown request against the Books3 subset of the Pile, as of December 2023 the Pile is no longer publicly available: `https://web.archive.org/web/20250403014846/https://academictorrents.com/details/0d366035664fdf51cfbe9f733953ba325776e667/collections`

5.3 Approach

To measure memorisation in LLMs4Code we first formally define a data extraction game and we construct a dataset of code samples.

5.3.1 Data Extraction Security Game

We consider the models as black-box systems. We define a security game inspired by the membership inference attack security game in Definition 1 and the notion of k-extractability in Definition 2:

Definition 3 (Data extraction security game) *Given a challenger $\mathscr{C}$, an adversary $\mathscr{A}$, a data distribution $\mathbb{D}$ and a model f the game is defined as follows:*

1. *The challenger samples a training dataset $D \longleftarrow \mathbb{D}$ and trains a model $f_\theta \longleftarrow \mathscr{T}(D)$ on the dataset D.*
2. *$\mathscr{C}$ samples a sample $D_n = (p, s)$ where $D_n \in D$. The prefix p is provided to the adversary $\mathscr{A}$.*
3. *$\mathscr{A}$ is allowed query access to the model f_θ and may perform any other polynomial-time operations*
4. *$\mathscr{A}$ outputs his prediction sequence $\hat{s}$*
5. *If $\hat{s} = s$, $\mathscr{A}$ wins, otherwise $\mathscr{C}$ wins*

In other words, given a prefix (1), the adversary is challenged to extract the correct suffix in the training data from the model. The adversary can query the model (2), but has no access to the weights, unlike the game proposed by Al-Kaswan et al. [151]. The adversary then predicts the suffix (3) and wins if it matches the actual suffix in the training data.

There are some difficulty modifiers to adjust the difficulty of the challenge:

1. The selection of the dataset $D \subset \mathbb{D}$. As observed by previous works, not all training samples are as hard to extract as others. In particular, samples that are highly duplicated[4] or outliers [139] are more vulnerable to attack.
2. The choice of model M_θ. Some models are more likely to memorise samples than others, namely larger models have been observed to memorise more samples [30, 38, 138, 146, 148, 163].
3. The length of the prefix p. It has been found that longer prefixes elicit more memorisation[4] [30, 138, 146]. Note that this length is equivalent to the k in Definition 2.
4. The victory condition $\hat{s} = s$, instead of targeting verbatim memorisation, a fuzzy match could also be considered [146].

In this work, we take inspiration from the competition organised by Carlini et al. and use modifiers (1) and (3) to construct a set of extractable samples. We shorten the prefix of the extractable samples and use this set of hard but extractable samples to perform an evaluation on different models (2). We also measure fuzzy match scores (4) and compare them with the exact match rate.

5.3.2 Code Dataset Construction

To measure the memorisation in LLMs for code, we first need to construct a dataset similar to the one used in the SATML'23 Language Model Data Extraction Challenge. As there is no code benchmark available, we build one from scratch. This presents several challenges:

Firstly, for some code models, the training data is not published by the authors, which makes it impossible to determine what data were included in the training of these models. We must therefore experimentally determine which data points were presumably included in the training data for each of the models. This has implications for the transferability of the benchmark set, as the training data might differ for each model. Not all models are trained in all programming languages as well, so we must select a common language to test multiple models.

Secondly, since all publicly available code is potentially part of the training data, the search space for extractable data points is massive.

We limit our evaluation to Python since we found that the vast majority of models support Python and have some Python in their training corpus. We source the potentially memorised data from GitHub. We mine Python files using the Google BigQuery Github dataset. [6]

We filter the files to include only nonbinary files longer than 150 tokens. We only consider files that have five or more duplicates on GitHub and randomly select 150 token spans from anywhere in the file. Similarly to the natural language dataset [7], we split the 150 token span into a pre-prefix, a prefix, and a suffix, each 50 tokens long. We prompt a CodeGen-2B-Mono model [21] with the pre-prefix and prefix. We select this model because it is decently sized (there are smaller and larger variants of the model), it is specifically trained on Python and it is the highest performing publicly-available model for the Human-Eval benchmark [21].

If the model can predict the suffix, with the 100-token prompt, we consider the sample to be extractable. We randomly select 1K extractable samples to perform our evaluation. We construct the dataset from the prefixes and suffixes.

Our dataset construction procedure differs from the procedure used by Carlini et al. in one aspect. Our dataset does not guarantee that for every $D_n = (s, p)$ there does not exist a $(s, p') \in D$ where $p \neq p'$. There are two main reasons for omitting this step:

- For many models in our evaluation we do not have access to the training data and possible pre-training data. The organisers could guarantee that the model under investigation was only exposed to the Pile. We want our approach to work for settings in which the investigator has no access to the training data.

- The computational cost of identifying all unique samples $D_n = (s, p)$ is extremely large for a dataset of this size and our aim is to create an approach that does not require such enormous compute capabilities.

[6] GitHub on BigQuery: https://web.archive.org/web/20260309084311/https://console.cloud.google.com/marketplace/details/github/github-repos

[7] Language Models Training Data Extraction Challenge: https://archive.softwareheritage.org/swh:1:snp:345b46885da937f2b11fbe553367e60251afcb15

5.4 Methodology

5.4.1 Research Questions

RQ1: *How does the rate of memorisation compare between Natural Language and Code trained LLMs?* To compare the rate of memorisation, we run both the attack on natural language as well as code models and compare the results. Intuitively we expect code models to be able to memorise more since code is more structured and there is much more natural language data available.

RQ2: *What type of data are memorised by code-trained LLMs?* We want to know if there is a code pattern that is memorised. To do this we take the set of samples vulnerable to attack and we manually analyse them by constructing a classification of the samples.

RQ3: *How much overlap is there between the memorised samples in different code-trained LLMs?* Do some models memorise different samples than others? Could we perhaps leverage a selection of different models to extract more data and do some models memorise more of a certain type of sample than others?

RQ4: *To what extent do LLMs trained in code leak their pre-training data?* Finally, we want to see if pre-trained models can also leak their pre-training data. To investigate this, we select a code model that has been pre-trained on the Pile and perform the natural language attack. We compare the performance of the original base model with that of the code-trained model to see how much training data is retained. When referring to a base model in this chapter, we only mean models that were initialised with the **architecture** and **weights** of a different model.

5.4.2 Models

The models, their developers, and their respective sizes are shown in Table 5.1. We limit our evaluation to left-to-right autoregressive models, which are available on the HuggingFace Hub.

For natural language evaluations, we used GPT-NEO [162], the models used to build the natural language dataset [4]. We select GPT-2 [164] to test the transferability of the prompts to a model trained on a different corpus. GPT-2 is trained on the WebText corpus, which was mined by finding all the outlinks on Reddit with more than 3 karma. We also investigate the Pythia [165] suite of models, which are trained on the Pile [161].

The CodeGen suite of models [21] features a number of different models in a variety of sizes. The models were initialised and first pre-trained on the Pile; these models are the CodeGen-NL models. The CodeGen-NL models are then further trained on a dataset containing multiple programming languages to create the CodeGen-Multi models. The Multi models were finally trained on a dataset consisting of only Python code to create the CodeGen-Mono models. The CodeGen2 and Incoder models are both designed for infilling but have autoregressive capabilities as well [21, 133]. CodeParrot is a pre-trained GPT-2 model fine-tuned on the APPS dataset [166]. PyCodeGPT is a small and efficient code generation model based on the GPT-NEO architecture [167]. GPT-Code-Clippy is a pre-trained GPT-NEO model fine-tuned on code.

Table 5.1: Natural language (top 4 rows) and code models under investigation

Model	Developers	Parameters
GPT-NEO	EleutherAI	125M, 1.3B, 2.7B
GPT-2	OpenAI	117M, 345M, 774M, 1.5B
Pythia	EleutherAI	70M, 160M, 410M, 1B 1.4B, 2.8B, 6.9B
CodeGen-NL	Salesforce	350M, 1B, 3B, 7B, 16B
CodeGen-Mono	Salesforce	350M, 1B, 3B, 7B, 16B
CodeGen-Multi	Salesforce	350M, 1B, 3B, 7B, 16B
CodeGen2	Salesforce	1B, 3.7B, 16B
CodeParrot	Huggingface	110M, 1.5B
InCoder	Facebook	1.5B
PyCodeGPT	Microsoft	110M
GPT-Code-Clippy	CodedotAI	125M

Table 5.2: Categories of memorised samples

Category	Purpose	Count
Code	Code Logic	679
Testing	Test Code	87
License	Licence information	13
Docs	Documentation	86
Dicts	Dictionaries or other data carriers	135

5.4.3 Categorisation

We build a classification of the 1K extractable 150-token samples by doing an explorative study. We find the following categories and classify each of the samples into one category. For simplicity, we classify each sample which has two purposes, into its majority category. The different categories are shown in Table 5.2. We identified 5 different categories as shown in Table 5.2.

5.4.4 Extraction

We prompt the model under investigation with the prefix. We use the standard generation pipeline and the default generation configuration of the model as defined in the model configuration. For models which use a different tokeniser than the CodeGen tokeniser used for the dataset construction, we simply tokenise the sample again using the new tokeniser. Any samples that are too short under the new tokeniser are discarded.

5.4.5 Evaluation Metrics

The models are prompted in a one-shot fashion with greedy decoding. We measure the exact match rate (EM). Additionally, we also measure the fuzzy match, using the BLEU-4

score. For the model size, we measure the total parameter count.

5.4.6 Configuration

We process and visualise the data with Modin 0.20.0 and Pandas 2.0.1. We run inference using Transformers version 4.16.2 running on Torch 1.9.0+cu111. The experiments were conducted on a cluster running RedHat 7, we allocated 8 CPU cores with 32GB of RAM and an Nvidia A40 GPU with 48GB of video memory. The GPU is running Nvidia driver version 530.30.02 with Cuda 12.1.

For replication purposes, we only consider models that are runnable on our hardware. We found that the limitation was the GPU memory, so there are some models that we did consider but did not fit the GPU memory (such as InCoder-6.7B and StarCoder-base).

5.5 Results

We present the results of our experiments to answer the research questions, results are grouped per research question.

5.5.1 Natural Language vs Code

The results of the attack are shown in Table 5.4. We found that we are able to extract 56% of the samples with the largest GPT-NEO model. The medium-sized model, which was used to construct the dataset, achieved an exact match rate of 46%. The models which were not trained on the Pile [161] did not memorise much if any of the samples.

As shown in Figure 5.1, for the models that are trained on the Pile [161], memorisation scales with the size of the model. We do not observe a clear difference between the Pythia and Pythia-dedup models, indicating that their deduplication was unsuccessful in preventing the memorisation which we measure. As the number of parameters increases for each model architecture, it becomes evident that the rate of memorisation grows logarithmically.

Table 5.3 and Figure 5.2 show the results of the experiments. We found that we were able to extract 38% of the samples from the largest CodeGen-Mono model we tested. The 1B parameter model, which was used to generate the test set, was only able to extract 30% of the samples, which is lower than the performance of GPT-NEO 1.3B on the natural language dataset. This indicates that our constructed code dataset is harder than the natural language dataset, but that difficulty modifier (2) from Section 5.3 which was supported by previous works and Definition 1 also holds for our code dataset.

Figure 5.3 shows the relation between the Exact Match rate and the BLEU-4 score for code-trained models. We can observe that there is a clear relation between the exact match rate and the BLEU4 score, especially above an exact match rate of 0.2. We see a similar pattern in Figure 5.3. The Pearson correlation coefficient between the Exact Match rate and the BLEU4 score is 0.982 and 0.967 for natural language and code, respectively, indicating a very strong positive correlation.

In our evaluation, we also tested multiple models that were not primarily trained on programming languages. We found that CodeGen-nl and GPT-NEO were unable to memorise as much as similarly sized code-trained models, but were still able to achieve an exact match score of around 10%.

Table 5.3: Code attack performance on Large Language Models for Code

		Memorisation rate	
Model	Parameters (M)	EM	BLEU-4
CodeGen-350M-Mono	357	0.101	0.567
CodeGen-2B-Mono	2779	0.303	0.712
CodeGen-6B-Mono	7074	0.382	0.756
CodeGen-16B-Mono	16032	0.471	0.801
CodeGen-350M-Multi	357	0.100	0.536
CodeGen-2B-Multi	2779	0.204	0.628
CodeGen-6B-Multi	7074	0.258	0.659
CodeGen-16B-Multi	16032	0.297	0.695
CodeGen-2B-nl	2779	0.077	0.465
CodeGen2-1B	1015	0.082	0.482
CodeGen2-3.7B	3641	0.106	0.517
CodeGen2-7B	6863	0.116	0.530
CodeParrot-small	111	0.088	0.529
CodeParrot	1510	0.314	0.721
InCoder	1312	0.115	0.559
PyCodeGPT	111	0.079	0.567
GPT-NEO	2651	0.058	0.454

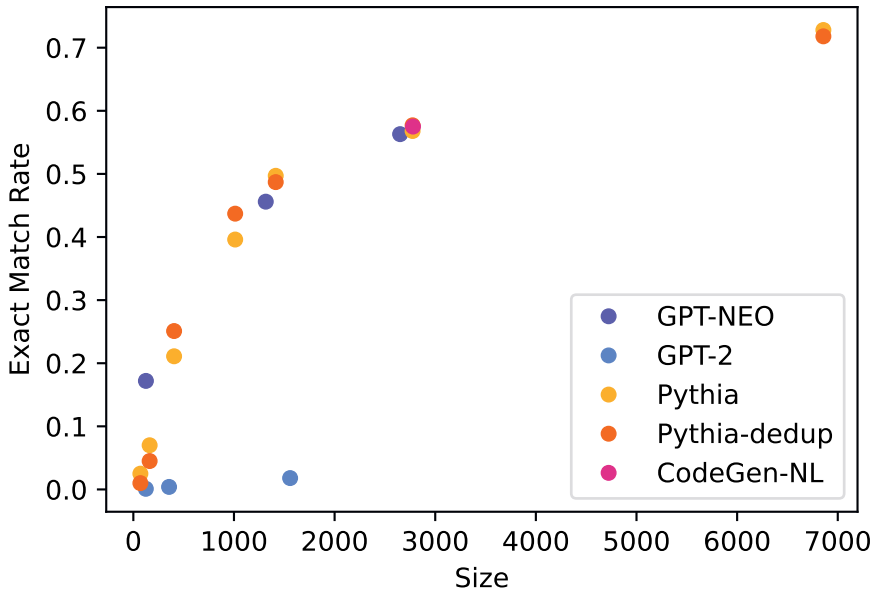


Figure 5.1: Parameter size and exact match rate for natural language models

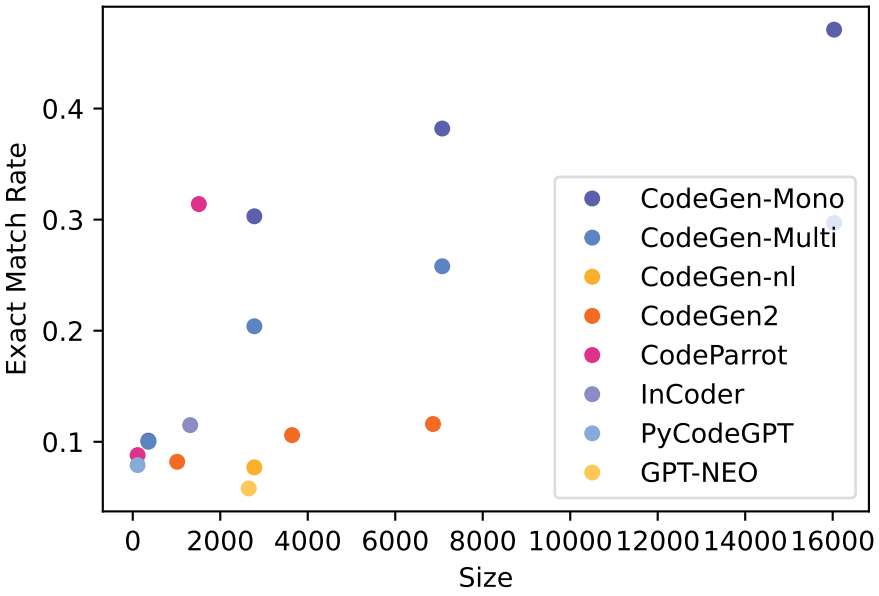


Figure 5.2: Parameter size and exact match rate for code models

Table 5.4: Natural language attack performance on natural language models

		Memorisation rate	
Model	Parameters (M)	EM	BLEU-4
GPT-NEO-125M	125	0.172	0.529
GPT-NEO-1.3B	1316	0.456	0.767
GPT-NEO-2.7B	2651	0.563	0.829
GPT-2	124	0.001	0.328
GPT-2-Medium	355	0.004	0.375
GPT-2-Large	1558	0.018	0.396
Pythia-70M	70	0.025	0.261
Pythia-160M	162	0.070	0.355
Pythia-410M	405	0.211	0.509
Pythia-1B	1012	0.396	0.658
Pythia-1.4B	1415	0.497	0.742
Pythia-2.8B	2775	0.568	0.793
Pythia-6.9B	6857	0.728	0.880
Pythia-dedup-70M	70	0.010	0.273
Pythia-dedup-160M	162	0.045	0.372
Pythia-dedup-410M	405	0.251	0.550
Pythia-dedup-1B	1012	0.437	0.679
Pythia-dedup-1.4B	1415	0.487	0.712
Pythia-dedup-2.8B	2775	0.577	0.805
Pythia-dedup-6.9B	6857	0.718	0.877
CodeGen-2B-NL	2779	0.575	0.860

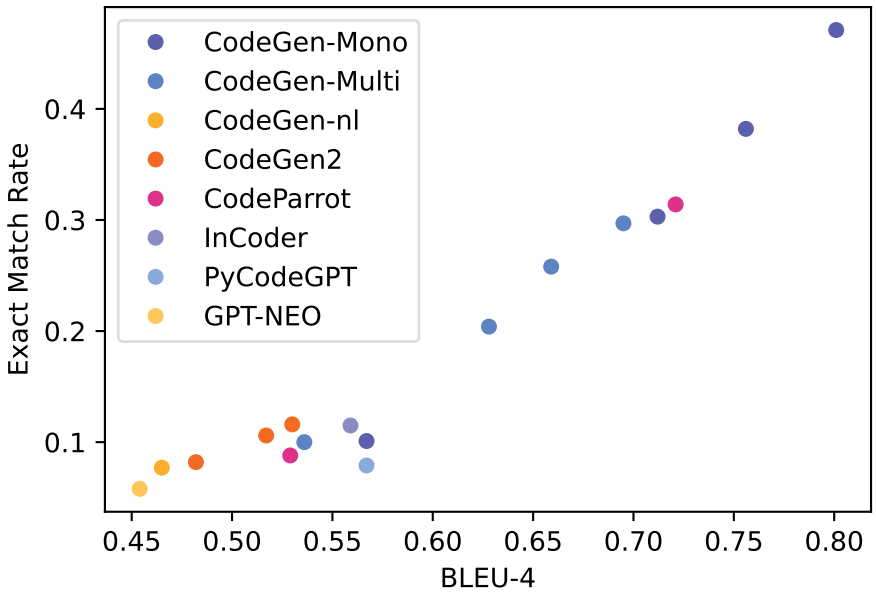


Figure 5.3: BLEU-4 score and Exact match rate for code models

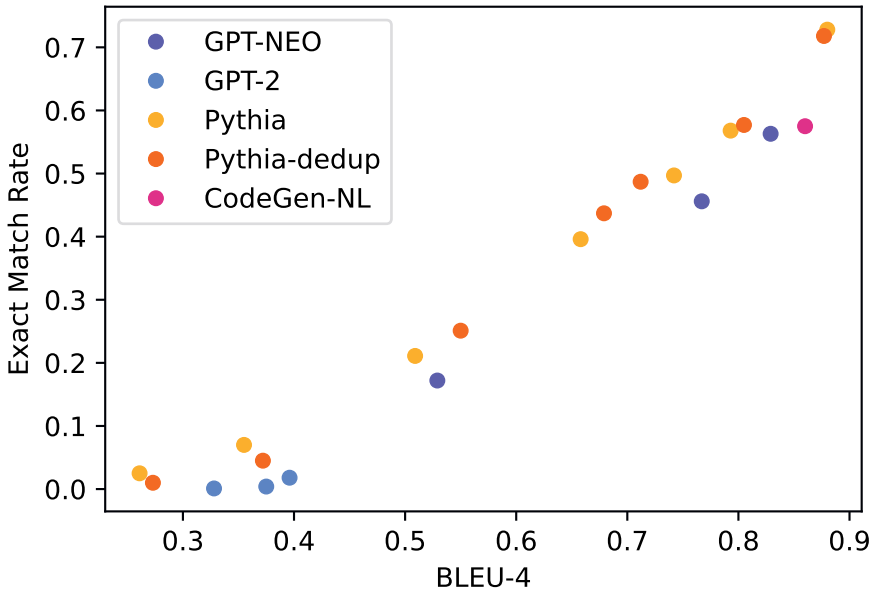


Figure 5.4: BLEU-4 score and Exact match rate for natural language models

Similarly to natural language models, we also find that memorisation scales with model size in Figure 5.2. But in this case, we see the logarithmic relationship between the same model architectures. We also observe that the CodeGen-Mono models memorise more natural language than the CodeGen-Multi models for every model size. This indicates that the extra training on Python code increases the memorisation rate. We find a Pearson's correlation coefficient between the Exact Match rate and the size of the model of 0.797 and 0.704 for the natural language and the code, respectively, indicating a strong positive correlation.

RQ1: Code-trained LLMs memorise their training data at a lower rate than Natural Language trained LLMs. In both natural language and code-trained models, the rate of memorisation scales with the model size. Continued exposure to the same data increases the rate of memorisation.

5.5.2 Type of Memorised Samples

As can be observed in Figure 5.5, the majority of samples in our dataset are code logic followed by dictionaries. We colour-coded the samples to make a distinction between memorised and non-memorised samples. We find that data carriers and licence information are being memorised at a higher rate than code logic, documentation, and test code.

During the tagging process, we did find multiple examples of names, emails, and usernames being memorised by the model. Such as the example in Figure 5.6 we also found an example of some API keys, further investigation shows that this instance was a sample that was easily findable using search engines.

RQ2: LLMs trained on code memorise data carriers and license information at a higher rate than regular source code, documentation, and testing code. Code-trained LLMs are also able to memorise and emit sensitive information.

5.5.3 Which Model Memorises What

In Figure 5.7 we plot the overlap in memorised samples between different models. We limit the investigation to the Codegen, CodeGen2 and CodeParrot family of models.

For instance, we find that 86% of all samples which were memorised by CodeParrot-small are also memorised by CodeParrot, while only 24% of the samples memorised by CodeParrot-small are memorised by CodeParrot. We find similar patterns when comparing the different-sized CodeGen models. The CodeGen-2 family of models memorised fewer samples and is in line with the CodeGen-350M models despite the size difference. The larger models in a family memorise more samples, there are a few distinct samples that are only memorised by the small models, but we find that is generally limited.

We find that the CodeGen-Multi models tend to memorise around 50% of the samples memorised by their respectively sized Mono variant, while the Mono models memorise around 70% of the samples memorised by the Multi variant. The only exception is the smallest model, where the Multi and Mono models memorised very similar amounts of samples. In Figure 5.8 we find that 40% of the samples are not memorised by any model at

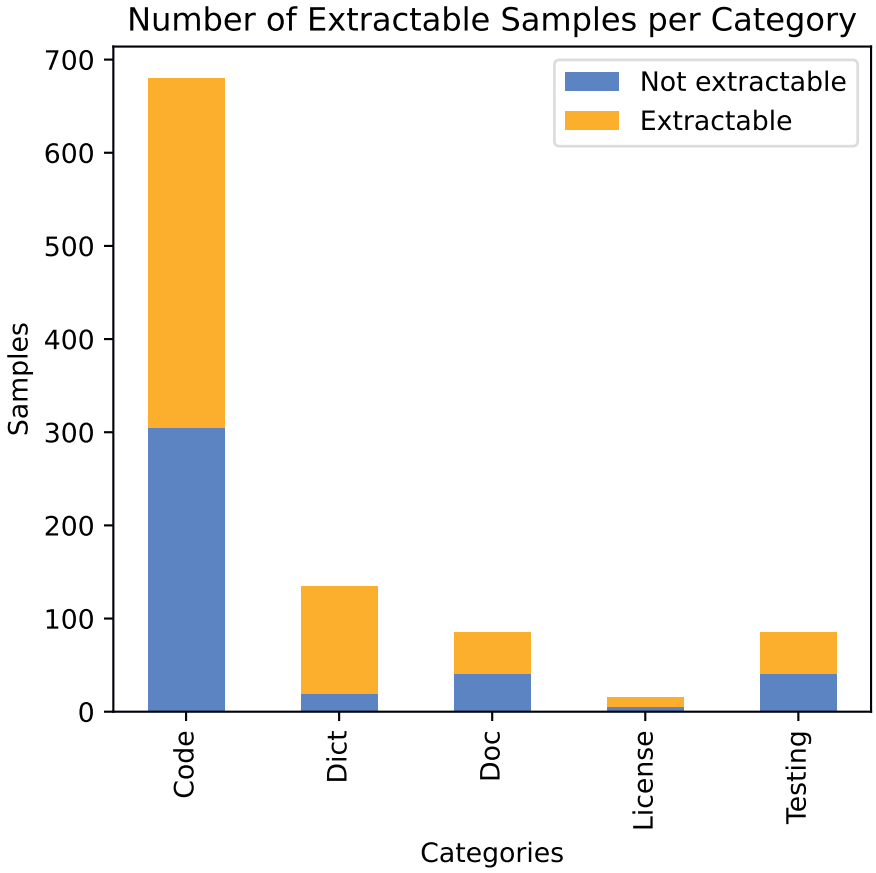


Figure 5.5: Categories of memorised samples

Figure 5.6: Instance of memorised API keys. Actual keys are replaced with placeholder values.

```
{
    'oauth_token: '##############',
    'oauth_token_secret: '################',
    'oauth_verifier: '###########',
}
>>> oauth_session
```

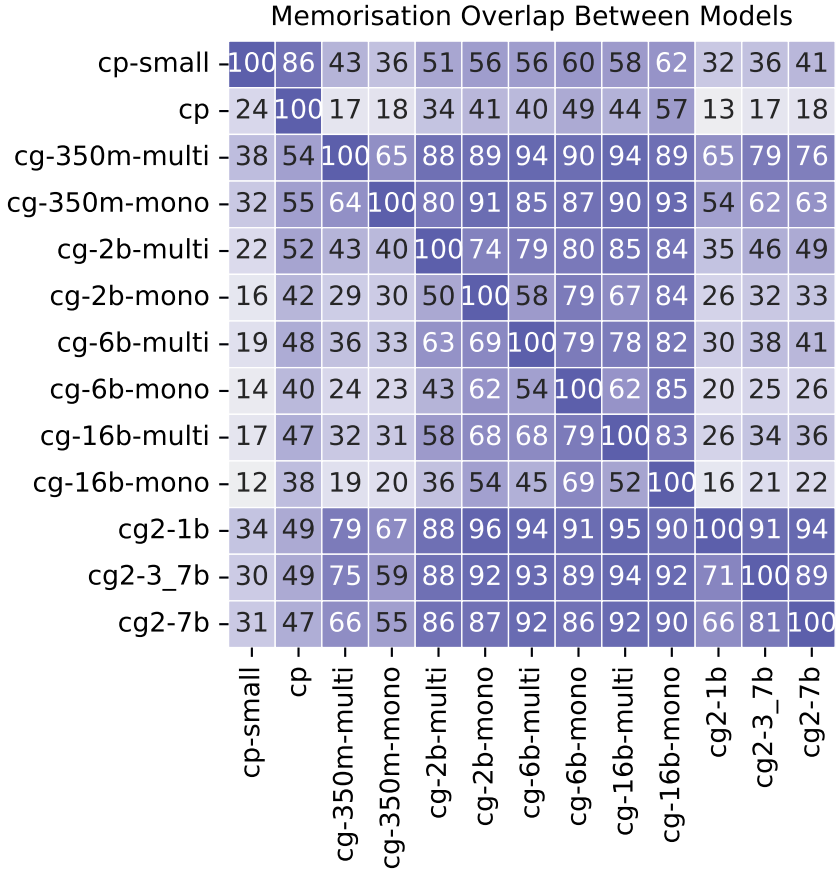


Figure 5.7: Memorisation overlap between CodeParrot (cp) and CodeGen (cg) Models

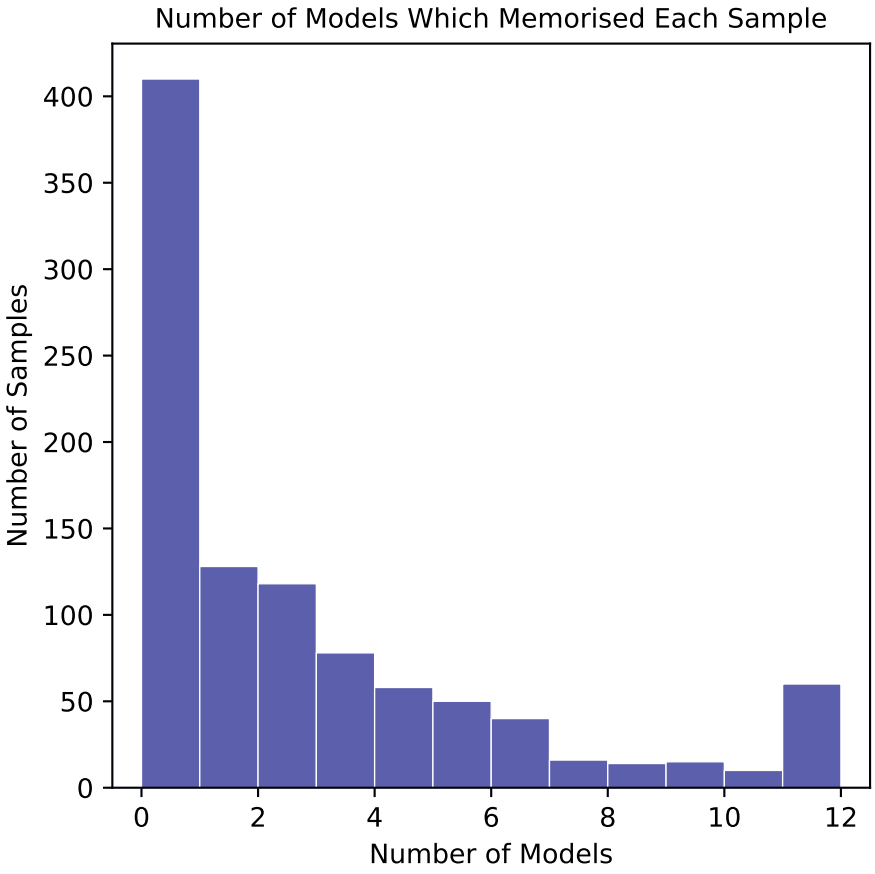


Figure 5.8: Memorisation counts

all. But there are 73 samples that are memorised by 12 of all the 13 models. This indicates that there is an inherent difficulty in some samples.

Figure 5.9 shows the memorisation of each of the categories per model. We find that all plotted models memorise more code and data carriers than any of the other categories, which is supported by Figure 5.5. As models grow larger they memorise relatively more code and fewer data carriers. In absolute terms, the number of memorised samples from the Dict category still increases.

Combined with the findings in RQ1 we can therefore conclude that the extra training on Python, makes the models memorise more and many of the same samples and that the smaller models lack the capacity to memorise more data.

RQ3: Each model family memorises a unique set of samples, and smaller models within the same family remember only a subset of what their larger counterparts do.

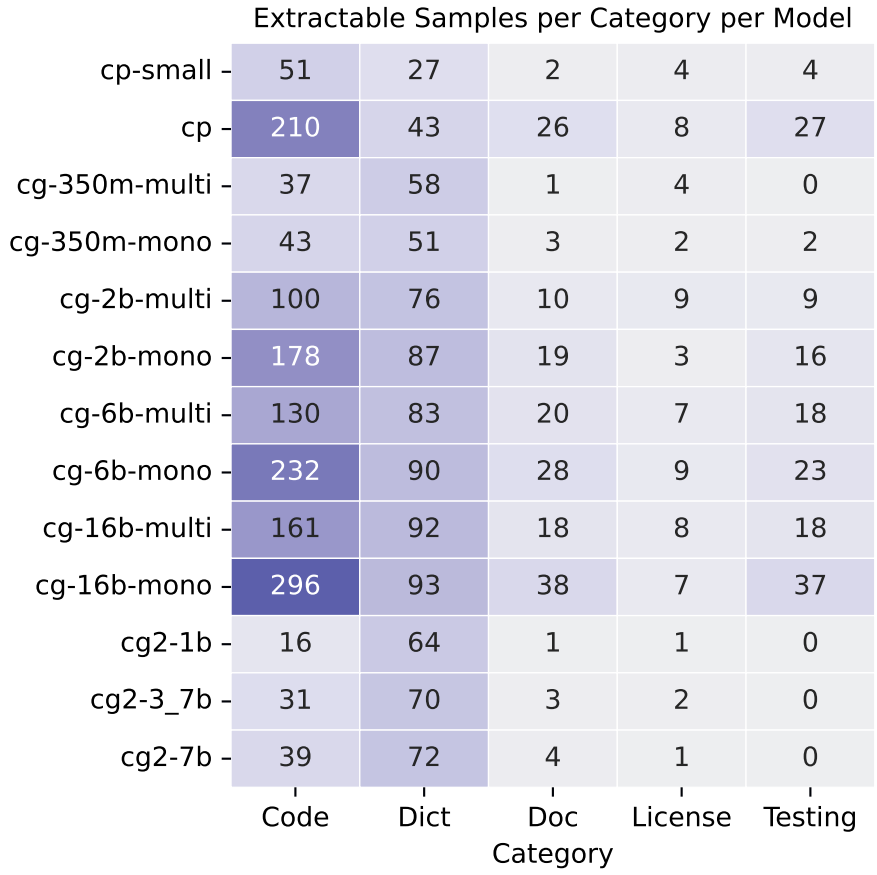


Figure 5.9: Percentage of extractable samples belonging to each category for CodeParrot (cp) and CodeGen (cg) Models

Table 5.5: Text extraction rate on code models

		Memorisation rate	
Model	Parameters (M)	EM	BLEU-4
CodeGen-350M-NL	357	0.295	0.676
CodeGen-2B-NL	2779	0.575	0.860
CodeGen-6B-NL	7064	0.708	0.915
CodeGen-16B-NL	16032	0.779	0.934
CodeGen-350M-Multi	357	0.248	0.539
CodeGen-2B-Multi	2779	0.310	0.588
CodeGen-6B-Multi	7064	0.414	0.595
CodeGen-16B-Multi	16032	0.351	0.618
CodeGen-350M-Mono	357	0.149	0.454
CodeGen-2B-Mono	2779	0.202	0.502
CodeGen-6B-Mono	7064	0.175	0.518
CodeGen-16B-Mono	16032	0.223	0.546
GPT-NEO	125	0.172	0.529
GPT-Code-Clippy	125	0.000	0.148

5.5.4 Pre-Training Data Leakage

In Table 5.5 and Figure 5.10 we plot the results for the leakage of pre-training data. We find that we can extract 58% of all natural language samples from the CodeGen-NL model. This result aligns with the similarly sized Pythia and GPT-NEO models in Table 5.4. Tuning the model on code data reduces the extraction rate to 31% and tuning on Python code further reduces the extraction rate to 20%.

We are unable to extract any text samples from GPT-Code-Clippy. The GPT-NEO-125M base model already shows very little extractability in Table 5.3.

RQ4: While fine-tuning does incrementally reduce the extractability of pre-training data, the pre-training data is still vulnerable to attack, especially as the models grow larger.

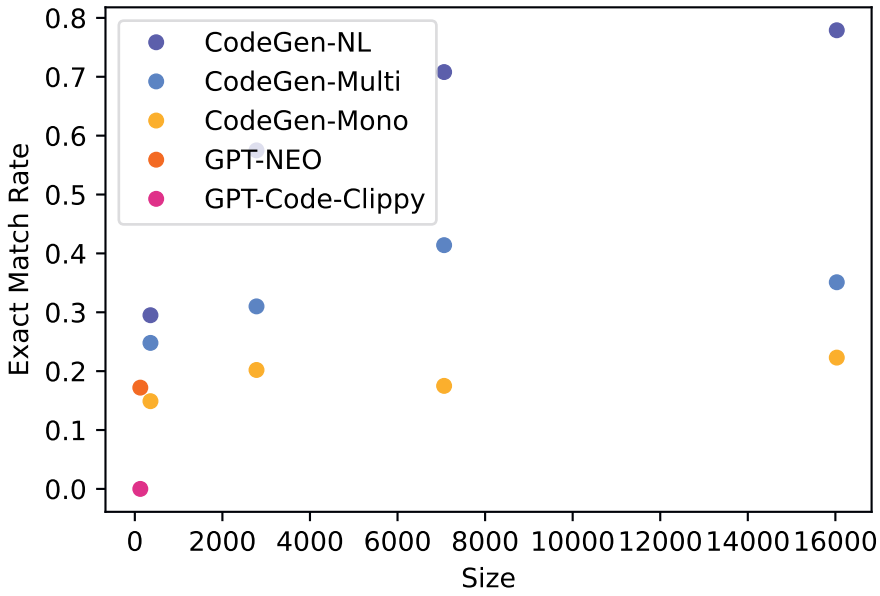


Figure 5.10: Parameter size and exact match rate for pre-trained models

5.6 Discussion

The results in Section 5.5 show that large language models pre-trained on source code also memorise their training data and that they are susceptible to targeted training data extraction attacks.

5.6.1 Interpretation

Multi vs Mono The findings indicate that the CodeGen-Mono models memorised more than the Multi models. This is explainable by the fact that the Mono models have had more exposure to Python code and therefore code in our dataset. Recall that the models are first trained on the Pile which contains all the GitHub repos with more than 100 stars [161]. The models are further trained on a general dataset of code, and finally on a dataset of Python code. This means that the models could have possibly been trained on the same file three times.

Size and Memorisation We find that the rate of memorisation scales with the size of the model, across all models we find that the rate of memorisation increases as the size increases. This is in line with the findings of previous work which found that larger LLMs memorise training data faster [142] and at a higher rate than small models [30, 38, 138, 148, 163]. Our results also confirm that the log-linear relation between size and memorisation, which has been observed by other works [138, 146] holds for LLMs trained on code as well.

Our experiments which investigate the overlap of memorised sequences in different sizes of code models show that the memorised samples of smaller models are mostly a

subset of the large models. This indicates that as a model grows larger it mostly memorises more and not necessarily different data.

Biderman et al. investigated memorisation in the Pythia suite of models [165] and found that 94% of the sequences memorised by the 70M model were also memorised by the 12B model, but those only accounted for 19% of the sequences that the 12B model memorised. We find a similar relation between the largest and smallest CodeGen-Mono models: CodeGen-Mono-16B memorised 93% of the samples which were memorised by CodeGen-Mono-350M, conversely only 20% of the samples memorised by CodeGen-Mono-16B were memorised by CodeGen-Mono-350M.

Rate of Memorisation Note that the results obtained from experiments in Section 5.5 suggest that memorisation in LLMs trained on code is less than in those trained in natural language. The largest 6.9B parameter Pythia model memorised 55% more samples than the best-performing CodeGen-Mono model. Intuitively we would expect the memorisation to be more in code models (as explained in Section 5.4), but there might be multiple reasons for this observation:

- Our dataset construction procedure differs from the procedure used by Carlini et. al. The natural language dataset guarantees that for every $D_n = (s, p)$ there does not exist a $(s, p') \in D$ where $p \neq p'$. This means that for some prefixes the model might predict a suffix that is also in the training data, which would be counted as a non-memorised sample. This was not possible in our case, since we do not exactly know the training data for the code models under investigation. The training dataset was only deduplicated on the file level.

- The structured nature of code might elicit less memorisation in general. This is supported by the higher rate of memorisation in dictionaries compared to regular code especially in smaller models. Their relative information density makes it hard to generalise for these samples specifically and the models might therefore revert to memorisation.

Deduplication The deduplicated Pythia [165] models are not significantly more robust against our extraction than their regular counterparts. At first glance, this is a surprising finding. It has been reported that deduplicating the training data makes LLMs more secure against data extraction [30, 168, 169].

A similar investigation by Biderman et al. on memorisation on the Pythia suite of models also found a relatively small difference between the two variants [38]. The authors theorise that this observation might be due to the training setup. The deduplicated models were trained for 1.5 epochs to offset the smaller data size and to train on the same number of epochs. This effectively oversamples the entire dataset.

Based on our observations we can offer two alternative explanations:

1. The training was deduplicated on the file level [165]. Our evaluation concerns spans of tokens that can be duplicated across files. The same licence information, for instance, is present in the preamble of many different files and will still be present in the deduplicated dataset.

2. The samples memorised by the Pythia models might be outliers that elicit memorisation. We observed that information carriers are more likely to be memorised than other types of samples, so the deduplication might not have had much impact on these samples.

5.6.2 Implications

We propose a novel framework to measure the memorisation and extractability of training data in LLMs.

Model training This work serves to inform researchers and practitioners who aim to train their own LLMs. We can confidently say that larger LLMs leak more and that smaller LLMs are therefore preferable from a safety perspective. In light of emergence [134], larger models are however often preferable. We are already able to extract 73% and 47% of the text and code samples, even larger models like CodeX [98] or Starcoder [170] might memorise even more data.

Secondly, we have shown that LLMs also leak their pre-training data even after multiple training rounds. The ability to recover pre-training samples has additional privacy and security implications for the transfer learning paradigm [29]. When creating and publishing a model, the base model is also something to be considered as the pre-training data can be unintentionally exposed as well.

Finally, some types of data are more vulnerable to extraction than others. This information can be used to inform the data selection procedure. Some categories like dictionaries can be omitted entirely to reduce the amount of memorisation. Future work can investigate how training data can be curated and sanitised to reduce memorisation in LLMs.

Model deployment The black-box setting of our evaluation has implications for MLaaS services as well. Since we do not require additional information about the model, our data extraction approach could be used against models that are offered through public APIs such as Copilot [98]. While Copilot does employ a memorisation filter, it is relatively easy to bypass [149]. There is a need to develop stronger countermeasures to prevent data extraction from these models.

Framework The framework and dataset provided can be used to evaluate different models. While our focus has been on left-to-right causal language models, different architectures, such as encoder-only models like CodeBERT [22] or encoder-decoder models like CodeT5 [20] might memorise different amounts and different types of training data.

Fair Use Many existing LLMs for code make use of code licensed under copyleft and other non-permissive licences [29]. The use of public code to train LLMs for code is an instance of fair use, which is a defence that allows the use of copyrighted works in new and unexpected ways and exists in many jurisdictions [140]. If the output of the model is similar to the copyrighted input fair use might no longer be applicable. The output needs to conform to the licence terms of the copied input [140], which can include share-alike and attribution clauses [29].

Memorisation can therefore put the creators and users of LLMs for code at legal risk [140]. This risk extends to pre-trained models, as some pre-training corpora, including the Pile [161], also contain code licensed under non-permissive licences [29]. The risk can be avoided by training models with code licensed under permissive licences (such as BSD-3 or MIT) or providing provenance information to trace the code back to its source so that the user of the output can abide by the original licence [140, 170].

Extraction techniques We were able to show that using relatively simple greedy decoding and the notion of k-extractability, most text models and all code models are leaking data. This only proves the inherent leakiness of these models and serves as a stepping stone for more advanced and powerful attacks. One approach worth investigating is the use of prompt engineering to extract data. With hard or soft-prompts [171] the model could be enticed to output more memorised data. Our work only prompts the models with the prefix, while different prompts might elicit more memorisation. Another approach is to explore the use of Membership Inference Attacks to increase the abilities of the attacks further. One could take inspiration from untargeted attacks and generate multiple suffixes per prefix using a different decoding method. The MIA can then serve to select the correct suffix [24].

5.6.3 Limitations and Threats to Validity

Internal validity

In our evaluation, we did not take into account the location of the samples. The samples are of a fixed token length but can originate from any arbitrary location in the file. Furthermore, Byte-Pair Tokenisation can cause the sample to start or end in the middle of a word. We based our dataset construction on existing work [38, 151], but samples from the beginning or end of the file could be easier to extract. Initially, untargeted extractions were attempted, and it was discovered that samples were predominantly obtained from the beginning of the file. Nevertheless, the current approach was chosen as it would enhance the versatility of our attack and enable us to extract samples from any location within the file.

External validity

Our evaluation focuses on a limited number of models, other models might exhibit more or less memorisation. Our benchmark was constructed using a single model, and while we were able to show that our benchmark gave promising results for other models, other data sources and models should be used to construct more benchmarks.

The constructed datasets only consider duplicated sequences; this inherently limits the applicability of our attack on low-duplication data. While other works do state that models can also memorise unduplicated data, we cannot experimentally confirm this as we only apply coarse file-level deduplication.

In the construction of our dataset, we only considered Python code. We selected Python because it is supported by almost all code generation models. Other less-expressive languages could show different patterns and different degrees of extractability. Python is a very popular language, so these results might also not apply to less popular languages. We plan to extend our evaluation to include more programming languages in the future.

Construct validity
We mainly use the exact match metric to measure memorisation in code models. This metric likely underestimates the actual number of memorised samples, as some might be slightly changed by the model. For this specific study, we are more interested in exact reproductions by the model, since we are more interested in the privacy and security aspect of memorisation. When examining the licensing aspects of memorisation, fuzzy match metrics may provide better insights. We included BLEU4 to account for this, but we found that it is highly correlated with the exact match rate. However, there are no automated metrics available to measure non-literal infringement based on current legal standards [140].

Ethical Considerations
While this work does describe techniques that can potentially be used to extract sensitive information from models, we do so ethically. Our goal is to bring attention to the issue of memorisation in LLMs for code and inform the users and creators of these models and provide them with tools to measure this. In this work, we, therefore, do not needlessly expose any private information, and we urge users of our framework to refrain from doing so as well. We target randomly selected sequences from popular and public repositories to avoid accidentally exposing private information. However, we still found some instances of usernames, emails, and API keys in our data, but we found that these are easily findable using search engines and are part of popular and well-indexed public repositories. We believe that the benefits outweigh the risks, and we decide to share our datasets.

5.7 Conclusion

To conclude, we presented an extensive study on memorisation in LLMs for code. We formally define a data extraction security game grounded in the existing notion of k-extractability and membership inference attacks. We utilised this game to create a dataset to measure memorisation in LLMs for code. We compared the rate of memorisation between models of code and natural language, we compared the rate and type of memorisation between different models, and we investigated the rate of memorisation of pre-training data in LLMs for code.

We found that LLMs for code memorise their training data like their natural language counterparts, albeit at a lower rate. We further found that the rate of memorisation increases as a model grows and that different model architectures memorise distinct sets of samples, while smaller versions of the same family tend to memorise a smaller subset of their larger sibling. We found that data carriers and licence information are being memorised at a higher rate than code, documentation, and tests. Finally, we found that the pre-training data is still vulnerable to extraction even after multiple tuning rounds.

Our work is a first step and provides a framework to measure memorisation in LLMs for code. We strongly advise the research community to conduct a more comprehensive investigation into the extent of data leakage and employ a diverse range of models and extraction techniques to develop safeguards that can effectively mitigate this issue. The consequences of data leakage can be severe, so it is crucial to take proactive measures to address this problem.

6

Model See, Model Do? Exposure-Aware Evaluation of Bug-vs-Fix Preference in Code LLMs

Large language models are increasingly used for code generation and debugging, but their outputs can still contain bugs that originate from training data. Distinguishing whether an LLM prefers correct code, or a familiar incorrect version might be influenced by what it's been exposed to during training. We introduce an exposure-aware evaluation framework that quantifies how prior exposure to buggy versus fixed code influences a model's preference. Using the ManySStuBs4J benchmark, we apply Data Portraits for membership testing on the Stack-V2 corpus to estimate whether each buggy and fixed variant was seen during training. We then stratify examples by exposure and compare model preference using code completion as well as multiple likelihood-based scoring metrics. We find that most examples (67%) have neither variant in the training data, and when only one is present, fixes are more frequently present than bugs. In model generations, models reproduce buggy lines far more often than fixes, with bug-exposed examples amplifying this tendency and fix-exposed examples showing only marginal improvement. In likelihood scoring, minimum and maximum token-probability metrics consistently prefer the fixed code across all conditions, indicating a stable bias toward correct fixes. In contrast, metrics like the Gini coefficient reverse preference when only the buggy variant was seen. Our results indicate that exposure can skew bug-fix evaluations and highlight the risk that LLMs may propagate memorised errors in practice.

6.1 Introduction

Large Language Models (LLMs) have revolutionised code-related tasks in software engineering, serving as powerful tools for code completion, generation, and debugging [2, 3, 79, 172]. Their popularity stems from their ability to understand and produce code in multiple programming languages, often outperforming traditional methods in terms of

speed and accessibility [173, 174]. For instance, models like GitHub Copilot, built on top of LLMs, have been widely adopted by developers, with millions of users leveraging them daily for productivity gains [1].

Although LLMs have transformed code generation and debugging workflows, their output is not immune from errors; this introduces substantial risks that can undermine software reliability and security [17, 29]. For example, LLM-generated code has been found to include various kinds of errors [175–177]. These bugs can propagate through systems, leading to costly failures, data breaches, or exploits in real-world applications. These bugs may stem from coding errors in training data [17, 178].

The ManySStubs4J dataset collects thousands of SStuBs, ('**S**imple, **St**upid **B**ug**s**'), which are single-statement bugs and their fixes from open-source Java repositories [39]. Although seemingly minor, these SStuBs are particularly impactful because they represent real-world errors, mined from open-source projects. Previous work has shown that LLMs are almost twice as likely to produce a buggy version of a line compared to the fixed version [175].

A critical aspect exacerbating the risks associated with LLM-generated code is the issue of memorisation. LLMs can memorise and regurgitate parts of their training data [30, 179–181]. This issue is particularly pronounced in code LLMs, where training data often consists of vast, publicly sourced, repositories that contain a mix of correct and erroneous code, leading to potential biases in model outputs [29, 178].

For example, if a model has been exposed to a specific buggy code sequence during training, perhaps due to its prevalence in open source datasets, the model may inadvertently favour and propagate code similar to the buggy version, rather than the fixed version [178]; indeed subsequent generations from the model could reflect memorised buggy code rather than correct coding patterns, either learned directly from corrected versions or via appropriate generalizations [29, 33, 182]. If memorised code is regurgitated, this can affect evaluation metrics in benchmarks [29, 34, 182], which raises concerns about the reliability of LLM-assisted code development [33, 34].

Beyond correctness, exposure-influenced behaviour of code LLMs creates concrete security and governance problems. From a security perspective, models that reproduce insecure patterns in training data can introduce systemic vulnerabilities and expand the attack surface across many downstream projects [29, 34, 176, 178]; memorisation of proprietary or sensitive snippets also raises legal risks if such content is regurgitated [29, 33, 183, 184]. Therefore, there is a need for exposure-aware methods that accurately assess how prior exposure exposure to data influences LLMs.

To investigate whether LLMs "see, then do" when it comes to buggy versus fixed code, we develop an exposure-aware evaluation pipeline that samples bug-fix pairs from mined benchmarks, and estimates training exposure for each variant using membership tools. We then measure model preference with a suite of likelihood-based metrics. Additionally we evaluate model completions based on the bug-fix context. We then relate preference to exposure and to bug categories, and perform robustness checks to identify which metrics and categories are most sensitive to exposure, with the goal of disentangling memorisation-driven propagation from genuine learned correctness.

Our evaluation yields several noteworthy findings. First, we find that about 67% of bug–fix pairs in SStuBs were unseen in the model's training data, and furthermore, the

fixes were substantially more likely to have been seen than the bugs. Second, in actual model generations, buggy variants are reproduced far more often than fixes. Moreover, exposure to the buggy variant substantially boosts bug-matching rates in generations, while exposure to the fix provides only marginal improvements in fix-generation rates. These patterns are consistent across evaluated models. Third, likelihood-based metrics such as the minimum and maximum token probability consistently favour the correct, fixed code across all exposure conditions, whereas others like the Gini coefficient can exhibit exposure-driven reversals (preferring the buggy code when the model has seen that bug in training). Finally, model preferences vary by bug category, with some robust to exposure, others highly susceptible to propagation, and a few exhibiting intrinsic bug preference.

We make the following contributions:

Exposure-Aware Evaluation Methodology: We introduce a novel framework to assess code LLM behaviour that accounts for training data exposure, we perform membership inference for each SStuBs sample to determine if the buggy or fixed variant appeared in the model's training data.

Insights on Metrics and Generations: We compare likelihood-based metrics to identify those sensitive to exposure and break down preferences by bug category.

Empirical Study of Bug Propagation: We conduct an extensive study on the SStuBs benchmark across multiple open-source LLMs, quantifying how prior exposure to buggy or fixed code influences model preferences and the propensity to generate bugs.

6.2 Background and Related Work

LLMs have been applied to code completion and repair, but errors in generated code remain a serious concern. Researchers have assembled benchmarks of real bugs to measure repair and detection tools. For example, the ManySStuBs4J dataset collects thousands of single-statement bugs and their fixes from open-source Java repositories [39]. Other standard benchmarks include Defects4J [185] and QuixBugs [186] which capture small but impactful errors (off-by-one mistakes, incorrect conditionals, missing null checks, etc.) in real programs. Recent studies find that large code models still produce buggy code at non-trivial rates: Jesse et al. report that an LLM is almost twice as likely to generate a buggy line (from SStuBs) as the correct fix [175] and that bugs introduced by others take much longer to fix [175]. Security analyses similarly show that model-synthesised code can include insecure patterns such as missing input sanitization or unsafe API usage [17, 176].

A major challenge in evaluating such models is memorisation of training data. Code corpora contain many duplicate snippets, and large models are known to memorise frequent patterns [31, 33, 187]. This raises the classic membership inference problem: given a code snippet, was it in the training set? In practice this is hard to determine, especially in cases where the training data is not available [30, 155, 181].

Prior analyses have shown that training/test overlap can skew evaluation: test snippets discovered in training data can inflate reported performance [35]. For bug–fix pairs,

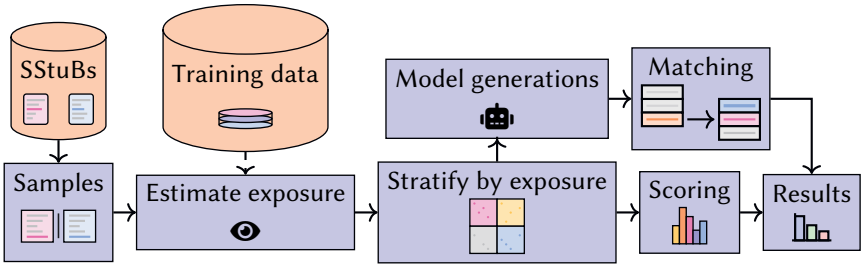


Figure 6.1: Overview of our exposure-aware evaluation framework

the asymmetry is especially problematic: a bug appears first in a project and may propagate to forks before it is fixed, so the buggy version may be more prevalent in scraped data than the fix. Jahanshahi and Mockus [178] found that 17% of the blobs in the Stack-v2 are copied and have a newer version of the code available. Of these newer versions, around 17% are bug fixes. These 'orphan vulnerabilities' can persist in public repositories even after a fix has been issued [178, 188].

Despite these insights, existing work has not explicitly controlled for exposure when comparing models' behaviour on bugs versus fixes. Previous work notes that models sometimes echo insecure or erroneous patterns from their training data [13, 33, 34, 189–192]. But none of these quantify how differing exposure to the buggy versus fixed code affects model preferences. In effect, previous evaluations may confound a model's learned correctness with simple memorisation. In this work, we address these gaps with an exposure-aware analysis of bug–fix preference.

6.3 Approach

The approach is described in Figure 6.1. Our exposure-aware evaluation framework studies bug propagation in code LLMs through an end-to-end pipeline: starting from the left, bug-fix pairs are sampled from the SStuBs dataset; exposure is estimated for each variant from the training corpus; samples are stratified into categories; model preferences are evaluated through likelihood-based scoring and actual code generations from LLMs; generations are matched against the original bug or fix variants; and finally, results are analysed to quantify exposure effects on bug-fix preferences.

To study the impact of exposure on bug propagation, we first sample a number of SStuBs from public repos. We then perform membership inference to check if the buggy or fixed version of the code is present in the training data.

With membership inference we can determine whether a bug or its associated fix appears in the large corpus. The samples can be stratified into four sets based on this analysis:

- **Neither** the bug nor the fix are present in the training data
- **Both** the bug and fix are present[1]

[1]Note that the number of instances data might differ between the bug and fix

- **Only** the **bug** is present, but the fix is not
- **Only** the **fix** is present, but the bug is not

Using a variety of LLMs, we can then calculate the model preference for either the bug or the fix, for each of the previously mentioned sets. We measure both conditional likelihoods and actual generative behaviour. The conditional likelihoods are extracted from the model losses and are scored using different token-level signals. Generation is evaluated using the LLM under decoding.

6.3.1 Research Questions

To guide our evaluation of code LLMs on bug propagation, we address the following research questions (RQs). These build on our hypothesis that training exposure to buggy or fixed code influences model behaviour.

RQ1: How does prior exposure to buggy or fixed code influence model preferences as measured by likelihood-based metrics? By computing metrics such as token probabilities we investigate biases in model scoring, helping disentangle memorisation-driven preferences from learned generalisations.

RQ2: How do exposure effects on preferences and generations vary by bug category? This RQ breaks down results by SStuBs categories to pinpoint which bug types are most susceptible to memorisation.

RQ3: How does exposure affect the generation of buggy versus fixed code in model completions? By focusing on verbatim regurgitation rates, we quantify explicit bug propagation in sampled outputs, assessing real-world risks like vulnerability introduction in downstream applications.

6.3.2 Datasets

Our study utilises two primary datasets: Stack-v2 [40] as the training corpus for membership inference and ManySStuBs4J [39] for bug-fix pairs.

To estimate exposure and perform membership inference, we utilise the Stack-v2 dataset. The Stack-v2 is the largest publicly curated code dataset spanning 619 programming languages and totalling 68TB [40]. The Stack-v2 is mined from the Software Heritage archive [193]. Its massive size improves the chance of overlaps with the SStuBs data and makes it suitable for training state-of-the-art LLMs. The Stack-v2 is widely used in research as it supports many languages and excludes restrictively licensed code [40].

SStuBs focus on single-statement bugs that can be fixed with single-line changes. SStuBs allow us to isolate changes without the complexity of multi-line or multi-file changes, which allows us to study model behaviour [39, 185]. We use the ManySStuBs4J dataset, which contains mined bug-fix pairs from open-source Java projects [39, 175]. We opt to use the processed version of the dataset from Jesse et al. as it contains additional information such as the number of commits between the fix and the bug, as well as the location of the bug in the source file [175]. The dataset consists of 16,899 bug-fix pairs.

6.3.3 Models

For our evaluation, we focus on models with open weights, and training data where possible, to improve reproducibility. Because of hardware limitations, we omit models that

are too expensive to run. Due to hardware limitations we can support models with up to $20B$ parameters. We found that few state-of-the-art models publish their training data. Of the 50 top performing models on the EvalPlus benchmark [15], only two have open training data, and both are instruction tuned versions of StarCoder2.[2]

The StarCoder2 family of models is fully open-weight and open-data. The models are based on the Transformer architecture and are available in 3 sizes, 3B, 7B and 15B parameters [40].[3]

StarCoder2 is trained on the Stack-v2 dataset, which is openly available. The 3B and 7B variants are trained on a subset of the Stack-v2 which only includes 17 programming languages (including Java), while the 15B parameter model is trained on the full set of 600+ languages [40].

We also include Mellum-4B which is an open-weight (but not fully open-data) model trained on Stack-v2 and some other datasets which are not publicly available [194].Despite its partially undisclosed training data, we can estimate exposure for SStuBs pairs, as the Stack-v2 forms the core of its code training.

Additionally, we include SmolLM3 (3B) [195], a fully open-weight model introduced by Hugging Face in 2025. SmolLM3 was trained on roughly 11 trillion tokens using exclusively public datasets in a staged curriculum that mixes web text, mathematics, and code. The code component of its training is drawn from Stack-v2 and other sources like GitHub issues, notebooks, and Q&A forums [195]. We therefore believe it is unlikely that any of the SStuBs ended up in the training data from sources other than Stack-v2. We use the SmolLM3 base checkpoint, without instruction tuning.

6.3.4 Data Portraits

To estimate the exposure of code variants, we utilise Data Portraits [196], a membership inference tool designed to measure the presence of specific text snippets or patterns in large corpora. We use the Data Portrait created by the authors for Stack-v2.[4]

Data Portraits use strided Bloom filters [196, 197] to enable efficient membership testing over large datasets. Bloom filters are probabilistic data structures that can quickly determine whether an element is in a set or not using fixed-length hash functions. Because different strings may map to the same position in the filter, Bloom filters can produce false positives, indicating membership when the element is not actually present, but they do not yield false negatives. Consequently, when applied in Data Portraits, some code snippets flagged as members might not truly exist in the corpus [196].

The false positive rate can be reduced by increasing the number of bits output by the hash function. In the case of Data Portrait these are tuned to achieve a false positive rate of 0.1% [196]. This results in a portrait of 90GB (compared to 68TB for the full dataset) for Stack-v2, which fits in memory and allows for almost instant membership testing [196].

It is important to note a key constraint of this method: its soundness is only guaranteed for queries exceeding a specific length. This results from the construction of the

[2]EvalPlus Benchmark: `https://web.archive.org/web/20250903200521/https://evalplus.github.io/leaderboard.html` (Accessed October 2025)

[3]StarCoder2: `https://web.archive.org/web/20260222085204/https://huggingface.co/collections/bigcode/starcoder2`

[4]Stack-v2 Data Portrait: `https://web.archive.org/web/20260308235429/https://stack-v2.dataportraits.org`

Data Portrait [196]. The tool was built using a fixed n-gram *width* (w) of 50 tokens and a *stride* (s) of 50 tokens. This means only 50-token sequences starting at every 50-token interval in the training corpus were hashed into the Bloom filter. Consequently, a snippet could fall entirely between these sampling points, leading to a false negative (i.e., the tool reporting a snippet as 'unseen' despite its presence in the data).

To ensure that any query is guaranteed to fully contain at least one of the sampled n-grams, regardless of its alignment, its length (L) must satisfy the condition $L \geq s + w - 1$. For the Stack-v2 Data Portrait, with $s = 50$ and $w = 50$, the minimum required query length is $50 + 50 - 1 = 99$ tokens. Any query of 99 tokens or more is thus guaranteed to contain at least one complete 50-token n-gram that would have been hashed and captured in the Portrait, ensuring the test is sound. We adhere to this constraint for all our membership queries by padding any shorter SStuB to 99 tokens using the surrounding context. In the case of Data Portraits, every character, except spaces and newlines, is considered a token.

We pad from both sides to ensure that we capture as much of the bug or fix behaviour of the snippet. If a sample is only padded with its preceding context, a hit in the Bloom filter could only cover the context and none of the sample. By padding both sides, we can ensure that the hit covers at least $(99 - sample_len)/2$ tokens of the sample in the worst case.

When a code snippet is queried, the Data Portrait tool calculates a 'badness' score. This score represents the fraction of the snippet's n-grams that are found in the Bloom filter. A higher score indicates a stronger signal that the snippet, or a very similar sequence, is present in the training corpus. For samples with $sample_len <= 99$ only a single hit is possible.

A high 'badness' score can arise from a query accumulating multiple hits. The cause is the duplicated presence of code blocks in the training corpus. For example, a function may appear in multiple files. Because the stride is applied independently to each file, the function's starting position relative to the beginning of the file dictates which of its internal 50-token n-grams are sampled and hashed. In one file, the stride may capture an n-gram from the beginning of the function, while in another file, a different alignment causes an n-gram from the middle of the same function to be hashed. Consequently, when the full function is used as a query, the sliding-window registers multiple distinct hits against the portrait, one for each independently hashed segment discovered across the corpus [196]. Therefore, a single query can generate numerous hits, indicating that different parts of it were seen by the model due to their varied placement across the training data. This underestimates the number of duplicates in the training data as some duplicates are aligned.

To convert this probabilistic score into a binary seen or unseen classification for our study, we apply a threshold. Following the methodology of Marone and Van Durme, we classify a snippet as exposed (seen) if its badness score is 90% or higher [196].

We apply Data Portraits to determine whether the SStuBs bug–fix pairs from our dataset were present in Stack-v2, the training corpus used for the models in our study [40].

6.3.5 Model Preference Metrics

To evaluate the preference of code LLMs between buggy and fixed variants, we compute $P(w_t \mid w_{t-1})$, the conditional probability of each token comprising the bug or fix, given the

preceding tokens i.e., the code lines before the buggy or fixed statement. As the preceding context is the same for any bug-fix pair, it follows that the probabilities are equal. Thus, we consider only the conditional probabilities of the tokens *comprising* the bug and fix code, which may have differing lengths. We collect two token probability sequences, one for the bug and one for the fix, per model and SStuBs pair.

We study multiple likelihood-based metrics to determine model preference based off the token probabilities. The model's preferred variant is selected based on the metric's orientation. For example, if the mean of the token probabilities comprising the fix is higher than the bug, the model prefers the fix. In contrast, a lower perplexity (surprisal) indicates greater preference over a higher perplexity sequence.

However, the choice of metric is not obvious, nor has a singular standard been established in prior work [198], as metrics capture different aspects of the token probability distribution. Therefore, we evaluate various metrics collected from prior work, which we introduce below.

Length ↑ The total number of tokens in the code sequence. While not a direct measure of preference, length can influence probability calculations and is used as a baseline metric [199].

Perplexity ↓ A widely used metric in language modelling that measures how well the model predicts a given sequence [200]. It is the exponentiation of the average negative log-likelihood per token. Lower perplexity indicates better model confidence and preference for the sequence.

Minimum Probability ↑ The smallest probability assigned to any token in the sequence. This metric highlights the least confident token prediction, with higher minimum probabilities indicating more uniformly confident predictions.

Maximum Probability ↑ The highest probability among all tokens in the sequence. This reflects the model's peak confidence at any position in the code snippet, which may indicate familiarity or memorisation.

Gini Coefficient ↓ A statistical measure of inequality in the distribution of token probabilities, where values closer to zero indicate a more uniform probability distribution, and higher values indicate more skewed confidence towards certain tokens [201].

Geometric Mean↑ The nth root of the product of token probabilities (multiplicative average), which penalises sequences with any low-probability tokens more heavily than high ones, emphasising consistently high or low probabilities across tokens.

Arithmetic Mean ↑ The arithmetic mean of token probabilities across the sequence. This captures the model's average confidence in the output, can be skewed by outlier high probability tokens. Lower arithmetic mean indicates lower average confidence (higher uncertainty) across the sequence.

6.3.6 Model Generations

To complement the likelihood-based metrics, we evaluate the models' actual generative behaviour. For each bug-fix pair in the SStuBs dataset, we provide the model with the

preceding code context (i.e., the code lines before the buggy or fixed statement) as a prompt, limited to 2048 tokens. We then generate completions using *temperature=0.8* and top_p = 0.95 following Lozhkov et al. [8, 40]. We generate five independent completions per prompt, to limit the computational cost. Each completion was limited to a maximum of 64 tokens, this is in line with literature [175]. We found that almost all SStuBs were under this limit. We then assess whether these generations match the buggy variant, the fixed variant, both (if different generations match each), or neither. Matching is determined by exact string comparison [175].

We compute the following disjoint aggregate statistics stratified by exposure category, which sum to 100%:

Fix without bugs: Proportion of samples where at least one generation matches the fixed variant but none match the bug.

Bug without fixes: Proportion of samples where at least one generation matches the buggy variant but none match the fix.

Mix bug-fix: Proportion of samples where at least one generation matches the bug and at least one matches the fix.

No bug no fix: Proportion of samples where no generations match either the buggy or the fixed variant.

6.3.7 Environment and Data Availability

All experiments were run on a server with two Nvidia A40 GPUs with 48GB of VRAM (driver version 535.261.03), 256GB of RAM and an AMD EPYC 9334 CPU with 64 cores. The experiments were run with Docker. A Dockerfile to reproduce the environment is available in the replication package.

A replication package, including all data and code, is available on Zenodo[5].

6.4 Exposure Characterization of Dataset

To understand the baseline exposure landscape, we first analysed the distribution of exposure across all SStuBs samples using Data Portraits membership inference on the Stack-v2 training corpus. Figure 6.2 illustrates the exposure distribution for both buggy and fixed code variants on a logarithmic scale. We observe that there are four major clusters, the first around a fix and bug exposure score of 0, the second around exposure scores of 1, a third around 1 and 0 and a fourth around 0 and 1 respectively.

There are a few samples with a fix exposure score less than 1 and a bug exposure score of 1. A score between 0 and 1 indicates that part of the sequence had a hit in the Data Portrait. But the length of the hit was not sufficient to conclude membership. Note that we only consider samples where the exposure is 90% or more to be members, as per the original work [196].

Some have exposures higher than 1, this can be caused by the bug or fix appearing multiple times in the training data. Some code snippets were duplicated dozens or even

[5]Replication package: `https://doi.org/10.5281/zenodo.17423424`

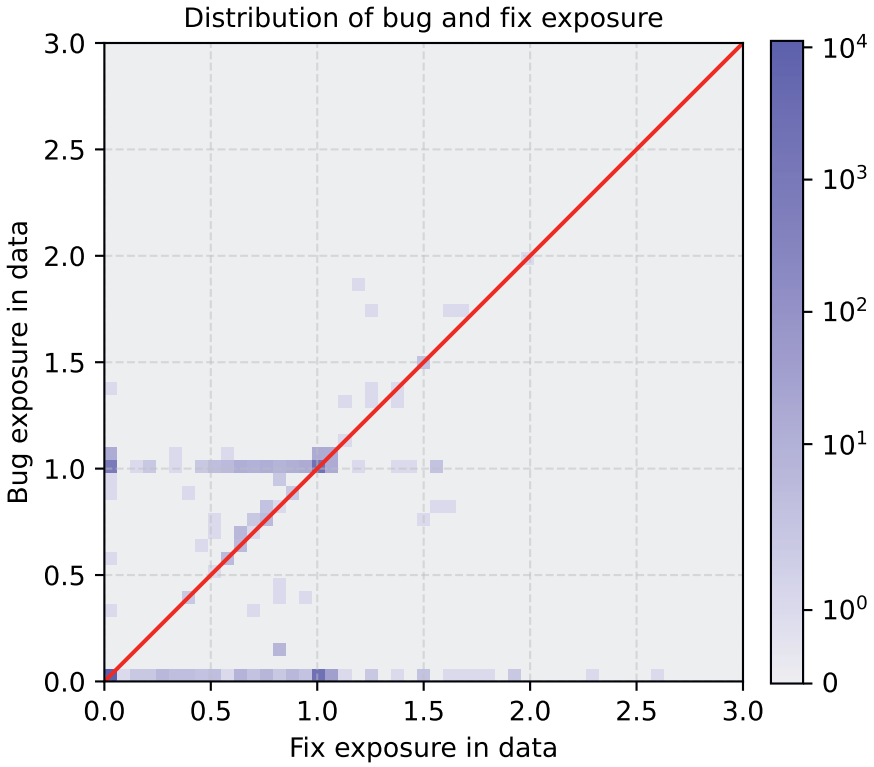


Figure 6.2: Distribution of exposure of bugs and fixes in logarithmic scale

Table 6.1: Count and mean number of commits until fix for each SStuB exposure category

Category	Count	#Commits	%
Neither seen	11286	4735	67%
Both seen	2169	9328	13%
Only Bug	1109	14152	7%
Only Fix	2335	6373	14%
Total	16899	6169	

hundreds of times in the training data, likely due to repository forks, copy-pasted code, or independently written clones.

As shown in Table 6.1, the vast majority of samples (11,286 pairs, or 67% of the dataset) had neither the bug nor the fix present in the training data. This finding suggests that most evaluation instances represent genuinely unseen code patterns. Among pairs with at least one variant exposed, fixes were substantially more likely to appear in the training corpus than bugs. Specifically, 2,335 samples (14% of the dataset) contained only the fix in Stack-v2, while only 1,109 samples (7%) contained exclusively the bug. A smaller subset of 2,169 pairs (13%) had fixed and buggy variants present in the training data.

We examine the temporal dynamics of bug-fix pairs by analysing the number of commits between the introduction of a bug and its subsequent fix. As shown in Table 6.1, the mean number of commits until fix varies substantially between exposure categories. Pairs where only the bug was seen exhibit the longest bug lifetime, with an average of 14,152 commits before correction, nearly double the dataset mean. In contrast, pairs where neither variant was seen have the shortest bug lifetime (4,735 commits on average). Pairs with both variants present show an intermediate lifetime (9,328 commits), while fix-only pairs average 6,373 commits before the fix.

> Most of the bug-fix pairs in our data set were not present in the training corpus, and 67% of the samples had neither variant seen. When exposure exists, fixes appear more frequently than bugs, and fixes tend to be longer on average than their buggy counterparts. Bug-only pairs exhibit nearly double the bug lifetime compared to the dataset mean, while neither-seen pairs show the shortest lifetimes.

6.5 Results

In this section, we present the empirical findings from our evaluation, addressing each research question in turn. For clarity and brevity, our analysis will primarily focus on the results of the StarCoder2-7B model. We found that the core patterns were remarkably consistent across the three StarCoder2 models evaluated. The complete results are available in the replication package.

6.5.1 RQ1: Exposure Influence on Metrics

In Section 6.3.5, we introduced the metrics used in our study. As there is no known singular, ideal metric, in this section we evaluate metric suitability for selecting the fix over

Table 6.2: Proportion of preferred variants by bug XOR fix exposure across models

Model	StarCoder2-7B				Mellum-4B				SmolLM3			
Exposure	Fix		Bug		Fix		Bug		Fix		Bug	
Preferred	Fix	Bug	Fix	Bug	Fix	Bug	Fix	Bug	Fix	Bug	Fix	Bug
length	0.59	0.41	0.48	0.52	0.59	0.41	0.48	0.52	0.52	0.48	0.49	0.51
perplexity	0.62	0.38	0.44	0.56	0.53	0.47	0.58	0.42	0.54	0.46	0.44	0.56
min_prob	0.68	0.32	0.78	0.22	0.70	0.30	0.77	0.23	0.72	0.28	0.79	0.21
max_prob	0.58	0.42	0.70	0.30	0.57	0.43	0.69	0.31	0.54	0.46	0.60	0.40
gini	0.60	0.40	0.40	0.60	0.60	0.40	0.41	0.59	0.49	0.51	0.39	0.61
geo_mean	0.62	0.38	0.44	0.56	0.64	0.36	0.47	0.53	0.54	0.46	0.44	0.56
arith_mean	0.60	0.40	0.38	0.62	0.62	0.38	0.39	0.61	0.50	0.50	0.40	0.60

Table 6.3: Proportion of preferred variants by both and neither exposure across models

Model	StarCoder2-7B				Mellum-4B				SmolLM3			
Exposure	Both		Neither		Both		Neither		Both		Neither	
Preferred	Fix	Bug	Fix	Bug	Fix	Bug	Fix	Bug	Fix	Bug	Fix	Bug
length	0.53	0.47	0.54	0.46	0.53	0.47	0.54	0.46	0.52	0.48	0.53	0.47
perplexity	0.51	0.49	0.57	0.43	0.53	0.47	0.58	0.42	0.49	0.51	0.56	0.44
min_prob	0.73	0.27	0.76	0.24	0.73	0.27	0.76	0.24	0.77	0.23	0.79	0.21
max_prob	0.65	0.35	0.63	0.37	0.65	0.35	0.63	0.37	0.56	0.44	0.62	0.38
gini	0.48	0.52	0.54	0.46	0.48	0.52	0.54	0.46	0.46	0.54	0.52	0.48
geo_mean	0.51	0.49	0.57	0.43	0.51	0.49	0.57	0.43	0.49	0.51	0.56	0.44
arith_mean	0.49	0.51	0.54	0.46	0.49	0.51	0.54	0.46	0.46	0.54	0.52	0.48

the bug. We study which metric reliably prefers fixes over bugs, *and* remains stable when controlling for exposure.

Table 6.2 present model preferences for pairs where exactly one variant (bug XOR fix) was seen during training. We pick an equal number of samples for each exposure setting (1109). This stratification isolates the effect of asymmetric exposure on model behaviour. When only the fix was present in the training data, the minimum probability showed the strongest bias towards correctness in all models, with 68–72% of the pairs favouring the fix between different models. Maximum probability exhibited similar robustness.

When only the bug was seen, minimum probability maintained its preference for fixes (77–79% across models), resisting exposure bias. This suggests that minimum probability captures something fundamental about code correctness beyond simple pattern matching. Maximum probability also remained fix-preferring in bug-only scenarios (60–70%), though with slightly reduced strength compared to fix-only cases.

In contrast, Gini coefficient and arithmetic mean showed exposure-driven preference reversal. When only the fix was seen, these metrics moderately preferred fixes (60–62%), but when only the bug was seen, they flipped to preferring bugs (56–62%). Length showed minimal discriminative power in either direction, hovering near 50-60% across all condi-

tions.

Perplexity reveals varying preferences. In fix-only conditions, all models favoured fixes. Under bug-only exposure, StarCoder2-7B and SmolLM3 shifted to prefer bugs, while Mellum-4B maintained a fix preference.

To further isolate learned correctness from exposure, Table 6.3 presents results for pairs where both variants were seen (AND condition) or neither was seen (NOR condition).

Across all three models, minimum probability maintained strong fix preference in both symmetric conditions (73–79% for both-seen, 76–79% for neither-seen). Gini coefficient and arithmetic mean lost nearly all discriminative power in symmetric conditions, hovering near or below 50% fix preference. Geometric mean occupied a middle ground, showing moderate fix preference in neither-seen conditions but losing discriminative power in both-seen scenarios. Perplexity preferences were more balanced or fix-leaning. Both-seen scenarios show near parity. Neither-seen conditions consistently preferred fixes across models.

The preference patterns observed for StarCoder2-7B (Table 6.2) replicate almost identically in Mellum-4B (Table 6.3), despite Mellum being trained on additional proprietary data beyond Stack-v2. SmolLM3, though showing slightly weaker overall fix preference, exhibits the same relative ordering of metrics and the same exposure-driven reversals for Gini and arithmetic mean.

> Minimum probability and maximum probability consistently prefer fixes over bugs across all exposure conditions. In contrast, metrics like Gini coefficient and arithmetic mean show strong susceptibility to exposure bias, reversing their preference toward bugs when only the buggy variant is seen.

6.5.2 RQ2: Exposure Effects by Bug Category

To investigate how bug-vs-fix preference varies by bug type, we stratified our analysis by the SStuBs bug categories for pairs where either the bug or the fix was seen. Figure 6.3, 6.4, and 6.5 show the preference breakdown for StarCoder2-7B, Mellum-4B, and SmolLM3, respectively. Each figure contrasts a metric which is biased towards fixes (min prob, top row) with an exposure-sensitive metric (geometric mean, bottom row) under two asymmetric exposure conditions: 'Seen: fix' (left) and 'Seen: bug' (right), note that these samples are exclusive, they belong to only one of the two exposure criteria, similar to Table 6.2. The blue bars represent the proportion of pairs where the fix was preferred, while the red bars indicate a preference for the bug.

Across all models, the results confirm the findings from RQ1 at a more granular level: min prob consistently favours fixes even when only the bug was seen, whereas geometric mean often flips its preference to align with the exposed variant. For instance, in Figure 6.3, when only the bug is seen (top right), minprob still prefers the fix in most categories, such as DIFFERENT METHOD SAME ARGS (188 fix vs. 64 bug). In contrast, geometric mean's preference for many of these same categories shifts dramatically towards the bug (bottom right), as seen with CHANGE IDENTIFIER (94 fix vs. 122 bug).

Certain bug categories demonstrate strong robustness to exposure. For CHANGE OPERATOR, all models show a strong and stable preference for the fix, regardless of which

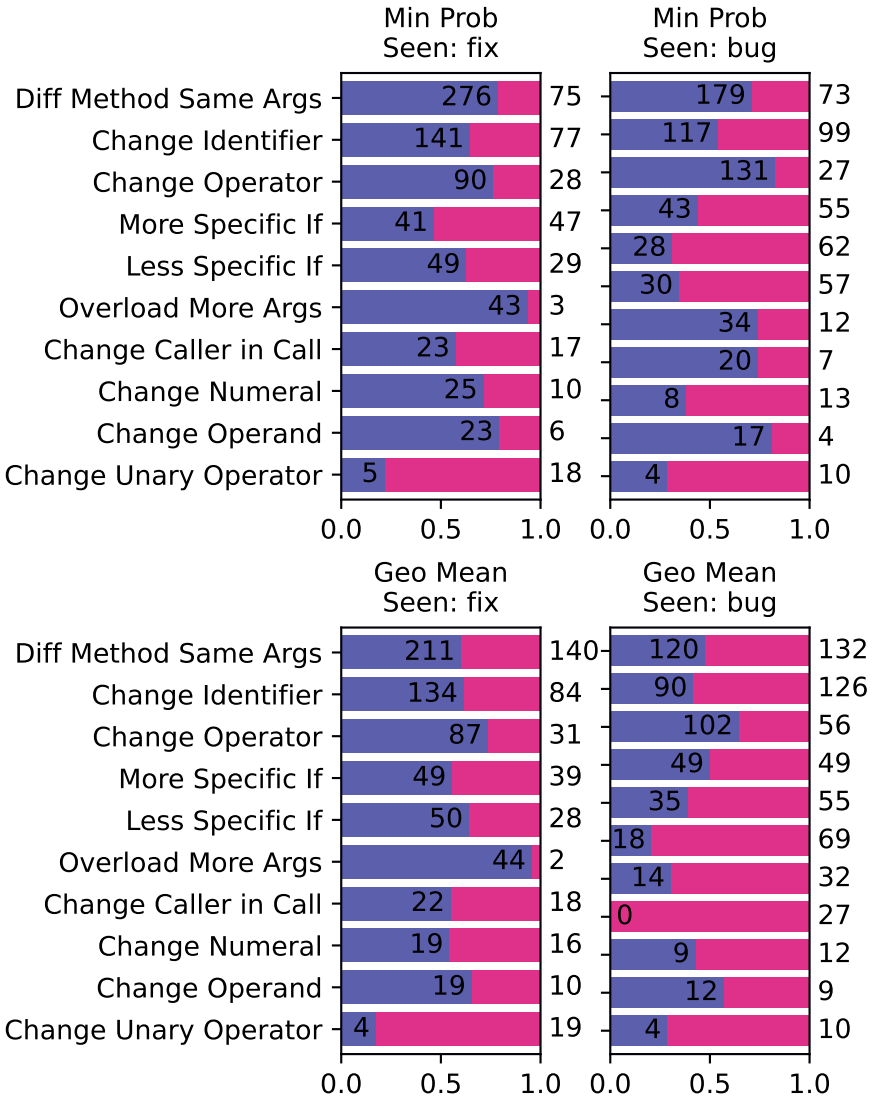


Figure 6.3: Model preference per bug category (StarCoder2-7B)

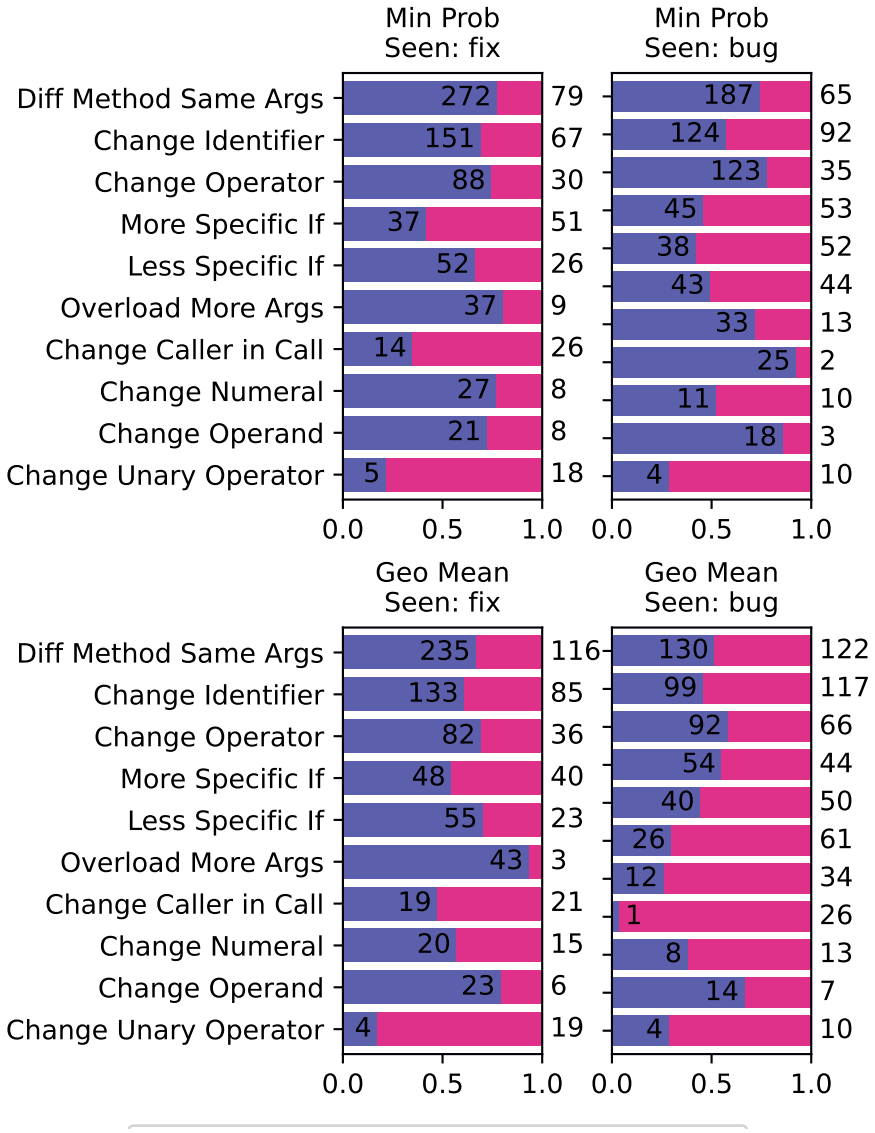


Figure 6.4: Model preference per bug category (Mellum-4B)

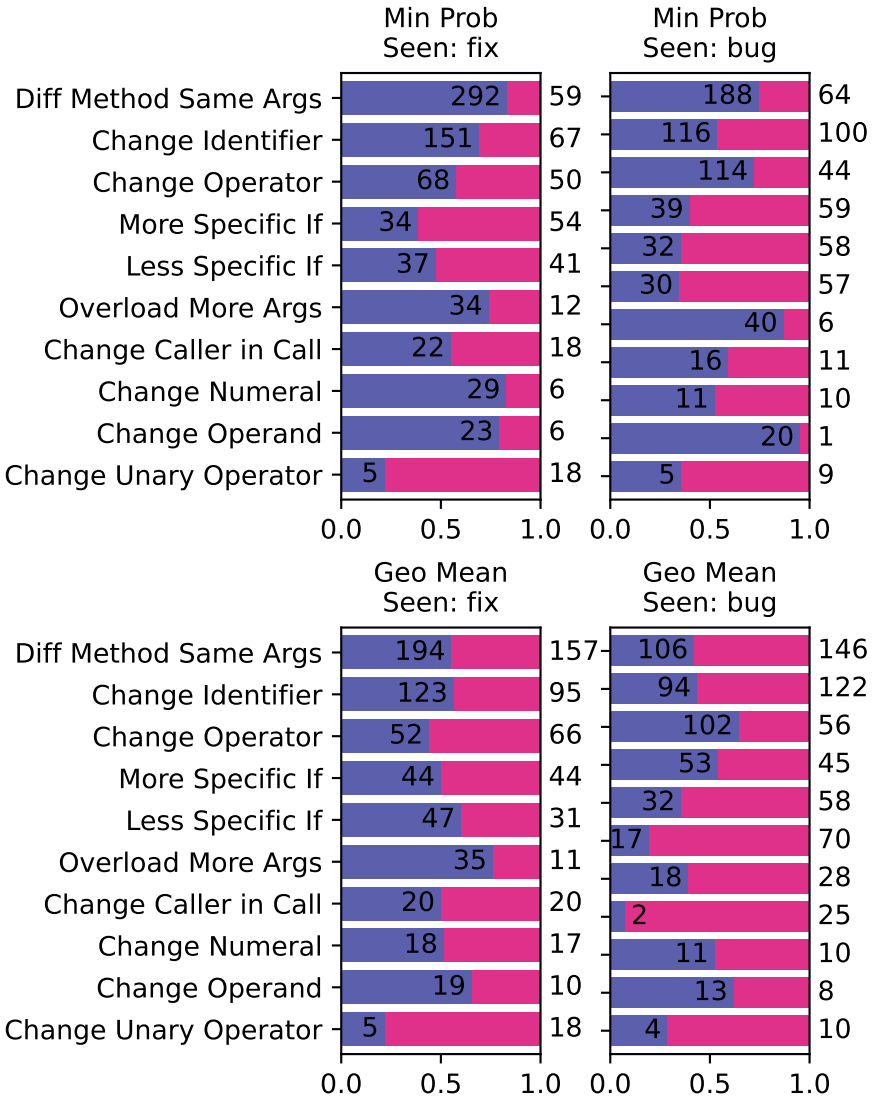


Figure 6.5: Model preference per bug category (SmolLM3)

variant was seen or which metric was used.

Conversely, some categories are susceptible to exposure-driven propagation. The CHANGENUMERAL category is a prime example. For all three models, both metrics show a preference flip. With Mellum-4B (Figure 6.4), min prob preference shifts from 27-8 in favour of the fix to 11-10. The effect is even more pronounced for geometric mean, which shifts from 87-31 (fix) to 56-102 (bug). Similar flips are observed for LESS SPECIFIC IF and OVERLOAD METHOD MORE ARGS.

Finally, we identify categories where the model consistently prefers the bug, regardless of exposure. For CHANGE UNARY OPERATOR, all models show a strong preference for the buggy variant even when only the fix was seen. For Mellum-4B, the preference is 5-18 in favour of the bug. This suggests the 'buggy' pattern may be far more common in the general training data or represents a strong prior that limited exposure to the fix cannot overcome. These patterns of category-specific robustness and susceptibility are consistent across all three models.

> Model preference varies significantly by bug category. Some categories are robust, with models consistently preferring the fix regardless of exposure. Others are more sensitive to exposure. A few categories, show an intrinsic model preference for the buggy version, which persists even when only the fix is seen.

6.5.3 RQ3: Exposure Impact on Generations

While RQ1 examines model preferences through likelihood-based metrics, RQ3 investigates actual generative behaviour by analysing what code models produce. We evaluate whether completions match the buggy variant, the fixed variant, both, or neither, stratifying results by exposure category to measure memorisation effects on generation.

The results for StarCoder2-7B (we omit Mellum-4B as the results were similar to the StarCoder2 models) and SmolLM3 are presented in Figure 6.6 and 6.7.

Across all models and exposure categories, generations match buggy code at substantially higher rates than fixed code, with bug-exclusive matches consistently outpacing fix-exclusive matches. For StarCoder2-7B (Figure 6.6), 'Bug without fixes' rates were highest in bug-only exposure and lowest in neither-seen, while 'Fix without bugs' rates were substantially lower across all categories. Mixed matches were moderate, and no-match rates were highest in the neither-seen category.

SmolLM3 (Figure 6.7) shows lower overall match rates compared to StarCoder2, with similar patterns: 'Bug without fixes' highest in bug-only and lowest in neither-seen, 'Fix without bugs' consistently low, mixed matches minimal, and no-match rates predominant, especially in neither-seen.

The bug-only exposure category consistently showed the highest 'Bug without fixes' rates across models. Conversely, fix-only exposure leads to modest increases in 'Fix without bugs' for StarCoder2, but minimal or no increase for SmolLM3. The both-seen category exhibited high 'Bug without fixes' rates for StarCoder2, comparable to bug-only, while SmolLM3's rates fall between bug-only and neither-seen.

> Models generate buggy variants at substantially higher rates than fixes across all exposure categories, with bug-exclusive matches consistently outpacing fix-exclusive

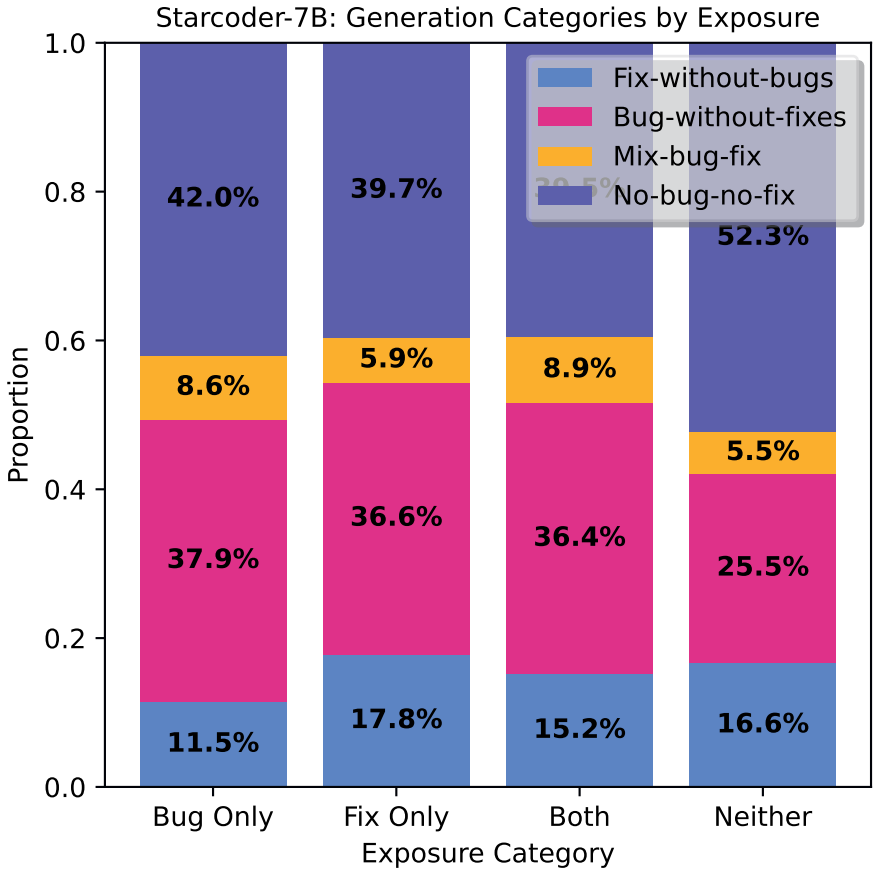


Figure 6.6: Model generations per exposure category (StarCoder2-7B)

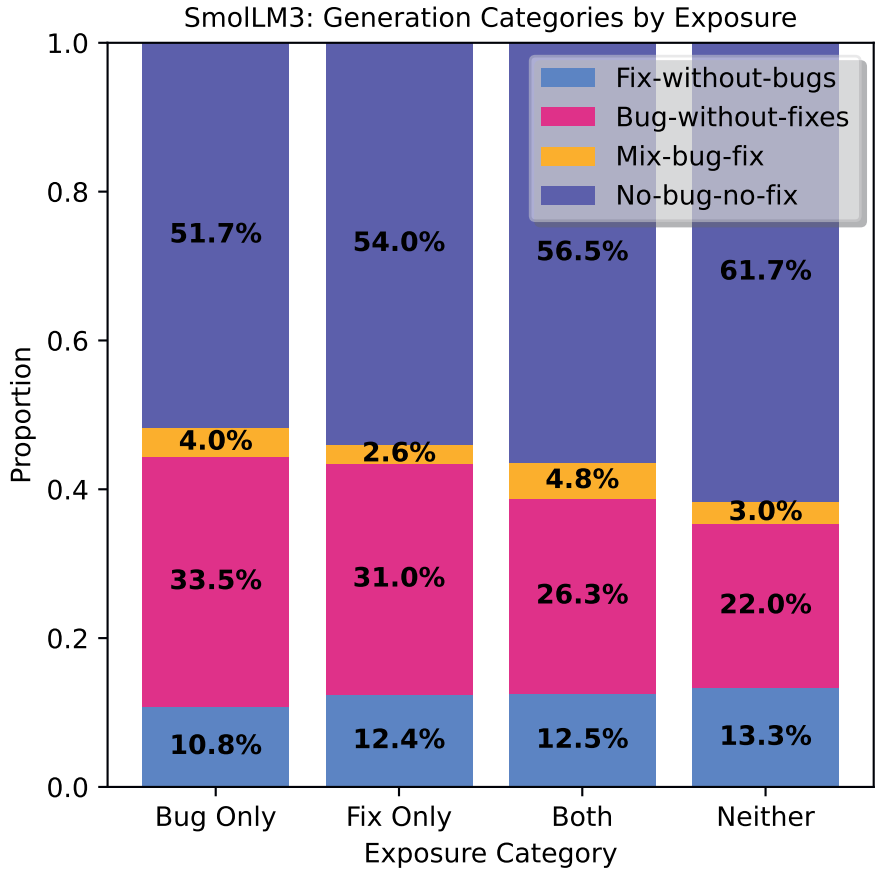


Figure 6.7: Model generations per exposure category (SmolLM3)

ones. Bug-only exposure amplifies bug propagation the most, while neither-seen minimises it; fix-only provides minimal improvement in fix generation.

6.6 Discussion

Our investigation reveals that code language models, despite encountering fewer bugs than fixes in their training data overall, exhibit a pronounced tendency to propagate bugs over fixes, heavily influenced by specific exposure patterns. Across all models, buggy variants were generated at substantially higher rates than fixes in every exposure category, with bug-only exposure exacerbating this bias. Addressing the core question of if "models see, then do": models only adhere to this principle to a certain extent. Exposure to bugs strongly drives increased bug generation and preference, whereas fix exposure only weakly promotes corrections. Metrics like minimum probability highlighted that models can generalise toward fixes in all exposure conditions.

The category-specific susceptibilities observed in RQ2, suggest that certain bug types align more closely with prevalent coding idioms or anti-patterns embedded in the training data. For instance, numeral changes may resemble common refactoring or optimisations that appear frequently in code repositories, making models more prone to propagating buggy variants when they are the only ones seen. This alignment could amplify security risks in LLM-generated code, as models might inadvertently reproduce vulnerable patterns. To mitigate this, targeted fine-tuning on under-represented fix patterns could bolster model robustness.

Another insight emerges from contrasting likelihood-based evaluation with actual generative behaviour. We observed that metrics like minimum token probability consistently favour the fixed code across all exposure conditions, whereas in practice the models often generate the buggy variant at much higher rates, especially under bug-only exposure. This reflects that these methods probe different aspects of the model. Likelihood metrics examine the probability distribution the model has learned, whereas generation metrics reflect the model's actual output under a given decoding scheme. In other words, the model "knows" that the fix should be more probable, but the process of sampling and decoding can still yield the bug as output. Both evaluation approaches can be complementary: likelihood-based metrics can diagnose what the model has learned and its confidence profile, whereas generation-based metrics reveal its practical behaviours and risks.

Bug Lifecycle Our results reveal that fixes appear more frequently in the training corpus than bugs, with 14% of pairs having only the fix exposed, compared to 7% exposing only the bug. This asymmetry may stem from the inherent construction of datasets like SStuBs, where bugs are mined from commit histories and can only be identified retrospectively once they are fixed. Bugs, by nature, have a limited observable lifetime in repositories, they persist until corrected, after which the fixed version dominates subsequent versions and forks. Consequently, training corpora like Stack-v2 are likely to capture more instances of the post-fix code, especially in actively maintained projects where fixes propagate quickly through merges and updates.

Analysis of bug lifetimes reveals that detected bugs in exposure categories like bug-

only pairs persist for substantially longer compared to fixes or neither-seen pairs. Cases where both variants are seen fall in between, suggesting a detection bias toward longer-lived bugs that survive multiple repository iterations before correction. Very short-lived bugs, which are quickly fixed within a few commits, may evade mining in datasets like SStuBs altogether. This could imply that our evaluation under-represents transient errors, where bugs have comparably short lifetimes, potentially skewing exposure-aware assessments toward more entrenched, harder-to-fix patterns that models are more likely to propagate.

The observed preference for bugs in model generations (RQ3), even in scenarios where fixes are more exposed. Fix-only pairs show only marginal increases in fix-matching rates, this highlights a potential imbalance in training data where erroneous patterns are reinforced through sheer volume or recency bias in code repositories. Bugs, often introduced in recent commits before being fixed, may appear in more transient or duplicated contexts across forks and pull requests. In contrast, fixes, while more stable, might be diluted by the nature of LLM training, where high-frequency error patterns overshadow corrections.

Higher bug generation rates may also arise from biases in dataset mining and the inherent 'naturalness' of human-written errors. SStuBs mining favours detectable bugs that were eventually fixed, potentially underrepresenting fixes that resemble common idioms, leading to an evaluation set where bugs appear more generation-prone. Additionally, these bugs were originally authored by programmers, making them feel 'natural' in probabilistic terms; models may favour them as they align with frequent coding slips observed in training. Furthermore, these are coding slips that managed to get past unit-testing, and perhaps code review, into the code base. On the other hand, their eventual discovery and correction indicates they were detected, and fixed.

Models Interestingly, our analysis uncovered minimal between the 3B and 7B and 15B variants of StarCoder2, despite the substantial disparity in model size and capacity. The models exhibit comparable generation rates favouring bugs across exposure categories. Preference metrics like minimum probability further reinforced this uniformity. This lack of differentiation suggests that scaling model parameters alone may not suffice to overcome inherent biases in training data.

The near-identical performance of Mellum-4B and the StarCoder2 models across RQ1–RQ3, in contrast to the distinct patterns in SmolLM3, suggests that, for our evaluation, additional training data has minimal impact. Mellum-4B, while incorporating some closed-source datasets beyond Stack-v2, exhibits preference metrics and generation rates that mirror StarCoder2-7B closely, implying that these proprietary additions are limited in scope or relevance to code tasks. SmolLM3, however, deviates with lower overall match rates and weaker fix preferences, likely due to its significant inclusion of non-code training data; although code is not expected in these portions, incidental inclusions or domain shifts could dilute its code-specific capabilities.

From an AI safety perspective, the tendency of models to regenerate seen bugs, as evidenced by higher bug-matching rates in generations (RQ3), raises significant concerns about downstream harms, such as introducing vulnerabilities into production software. When models memorise and reproduce buggy patterns from training data, they risk propagating errors across multiple projects. This phenomenon aligns with broader discussions

on memorisation risks in LLMs, where unintended data regurgitation can undermine software reliability [30, 33, 178].

While developers typically have little control over the composition of pre-training datasets, our study highlights the specific risks that models may inherit from such data. Understanding these exposure effects is therefore essential, particularly because practitioners often deploy models without direct visibility into or control over their training sources [29, 91, 202].

6.6.1 Threats to Validity

Although we have made efforts to ensure the robustness and reliability of our study, several potential threats to validity should be acknowledged.

Internal Validity One threat is the potential for false positives in membership inference using Data Portraits, as Bloom filters can incorrectly indicate exposure due to hash collisions, with a reported false positive rate of 0.1% [196]. This could lead to misclassification of bug-fix pairs into exposure categories, potentially exaggerating exposure-driven effects. We limited this impact by adhering to the tool's recommended query length threshold (99 tokens) and using the recommended 90% badness score cutoff for binary classification [196].

A potential limitation in our analysis stems from the context provided to models, which may not be sufficiently constraining to unambiguously distinguish buggy from fixed code. We evaluate single-statement changes with the preceding file context, without full project context, meaning that the preceding lines alone might admit multiple valid completions. This could explain instances where generations match neither variant (prevalent in RQ3), as models generate plausible but unrelated code.

Another threat arises from the probabilistic nature of LLM generations in RQ3, where decoding may introduce variability due to model stochasticity. We mitigated this by generating five independent completions per prompt and aggregating statistics ('Any bug-fix' rates), reducing the influence of single-run anomalies.

External Validity We rely on Java-specific bug-fix pairs from the SStuBs dataset, which may not generalise to other programming languages with different syntax or error patterns (such as dynamic languages like Python). This could limit the applicability of category-specific findings in RQ2 to multilingual settings. We limited this impact by selecting SStuBs for its focus on single-statement bugs, which are common across languages, and by using models trained on Stack-v2's diverse corpus (619 languages). However, we accepted the Java focus as a necessary scope limitation, given SStuBs' established use in code LLM evaluations [39, 175, 203], and recommend future extensions to multilingual benchmarks.

We only evaluated a small set of open-weight models (StarCoder2 variants, Mellum-4B, SmolLM3), which may not represent closed-source models like GitHub Copilot or larger architectures. Differences in training data or scale could alter exposure effects. We mitigated this by including models of varying sizes (3B to 15B parameters). We did not evaluate the models using the fill-in-the-middle (FIM) objective. SmolLM3 does not support FIM [195], and StarCoder2-15B exhibits poor FIM performance due to known

training errors by its authors [40].

Construct Validity Our binary classification of exposure based on Data Portraits oversimplifies memorisation, ignoring nuances like partial matches or contextual variations in training data. This could misrepresent true exposure, affecting downstream analyses. We accepted some simplification as a trade-off for scalability, given the size of Stack-v2.

Our analysis measures memorisation using literal string matching, as partial or approximate matching lacks a well-defined and reliable criterion for automated data extraction [204]. Because buggy and fixed versions of code often differ by only a small number of tokens, generations that do not exactly match either version are difficult to classify, potentially leading to underestimation of partial memorisation. We apply this strict criterion uniformly across all exposure categories, preserving the validity of relative comparisons.

6.6.2 Future Work

Our exposure-aware evaluation framework opens several avenues for future research. One promising direction is extending the analysis to multilingual and multi-dataset settings to enhance external validity. While our study focused on Java bug-fix pairs from SStuBs, incorporating benchmarks such as Defects4J, QuixBugs or PySStuBS [185, 186, 205], which span languages like Python and C++, could reveal language-specific exposure effects. For instance, dynamic languages might exhibit different bug lifetimes or category susceptibilities (such as type-related errors in RQ3), potentially modulated by syntactic differences. By applying Data Portraits to the full Stack-v2 corpus, future work could quantify cross-language propagation risks, providing a more comprehensive view of how code LLMs handle diverse error patterns.

Improving membership inference techniques represents another key area, and would solve the internal validity threats from false positives and binary classifications in Data Portraits. Advanced methods, such as embedding-based similarity searches or neural membership inference [31, 32, 155, 206], could offer finer-grained exposure estimates, accounting for partial matches or contextual variations in training data. This would enable nuanced stratifications beyond our seen/unseen dichotomy, such as weighting by exposure frequency or recency, and better isolate memorisation from generalisation in preference metrics. Black-box membership inference could allow for the detection of exposure, purely based on model characteristics. However, current model-only membership inference approaches are not reliable enough to assist in our evaluation [158, 181].

Finally, the metrics discussed in RQ1 and RQ2 might offer practical utility for developers by helping rank and filter multiple code completion candidates to select the most reliable option. Furthermore, metrics like the Gini coefficient can function as a form of membership inference, allowing developers to identify, and decide what to do with, code that has been memorised directly from the training data, without access to the training corpus.

6.7 Conclusion

In this chapter, we explored whether code LLMs "see, then do" by propagating bugs from training data, hypothesising that exposure to buggy versus fixed code could drive mem-

orisation and error regurgitation. Through an exposure-aware pipeline combining membership inference, likelihood metrics, and generation analysis on the SStuBs benchmark, we empirically examined this across multiple models.

Our results confirm that models do "see, then do" bugs: while many bug-fix pairs remain unseen and fixes are generally more exposed than bugs, models often prefer and generate buggy variants, especially when exposed to them. Some likelihood preferences lean toward fixes but can reverse under bug exposure, with effects varying by bug type.

Our study underscores the need for exposure-aware evaluation in coding LLMs. The "see, then do" effect reveals that models internalise fixes as preferable in likelihood scores, yet frequently reproduce memorised bugs in generations. This discrepancy between knowledge and outputs has implications for tool reliability, leading us to advocate membership testing and diverse metrics to uncover memorisation biases. By controlling for exposure, researchers can better assess true generalisation, mitigate error propagation, and promote safer software development.

7

Conclusion

In this chapter we conclude the thesis by revisiting the research questions, summarising the cross-cutting findings, and discussing the broader implications. We then reflect on the societal impact of LLMs in software security, and outline directions for future work.

7.1 Revisiting the Research Questions

In this section we revisit and answer the two research questions laid out in Chapter 1.

7.1.1 RQ1: How can LLMs be adapted and rigorously evaluated for realistic software-security tasks?

The first research question concerned the adaptation and evaluation of LLMs for realistic software-security tasks. The results of this thesis show that LLMs can be adapted to such tasks, but only when supported by domain-specific datasets, carefully designed evaluation frameworks, and metrics that reflect the structure of the task.

Chapter 2 demonstrated this in the context of binary reverse engineering. Source-code-pretrained models were adapted to summarise decompiled binary functions by constructing CAPYBARA, a dataset of aligned source, decompiled, and transformed code-summary pairs, and by fine-tuning CodeT5 into BinT5. The results show that models trained on source code can be transferred to decompiled code with strong performance. This finding is important because decompiled code lacks much of the natural channel available in source code, such as meaningful variable names, comments, and high-level structure. The chapter therefore shows that LLMs can support low-level program understanding tasks, provided that the input domain is represented by suitable training and evaluation data.

Chapter 3 extended the analysis from static program understanding to interactive cyber tasks. DeepRed evaluates LLM agents in virtualised Capture The Flag environments, where agents must use terminal tools, interact with target systems, and make progress through multi-step exploitation workflows. The results show that current LLM agents can make progress in realistic cyber settings, but remain far from robust autonomous security agents. This chapter also shows that binary success metrics are insufficient. In complex

cyber tasks, an agent may perform reconnaissance, identify services, recover credentials, or obtain a foothold without fully capturing the final flag. Partial-credit evaluation therefore provides a more informative account of agent capability than solved-or-unsolved metrics alone.

> LLMs can be adapted and evaluated for realistic software-security tasks, but this requires more than applying general-purpose models to security prompts. It requires domain-specific datasets, benchmark environments, evaluation setups and metrics that capture task-specific constraints. Capability claims in software security should be grounded in realistic task design rather than anecdotal demonstrations or simplified benchmarks.

7.1.2 RQ2: What security, privacy, and safety risks arise from deploying code-oriented LLMs?

Chapter 4 showed that LLMs used for programming may provide unsafe or maliciously useful assistance. The harmfulness study developed a taxonomy of harmful software-engineering scenarios, constructed a prompt dataset, and trained an automatic evaluator. The results show that harmlessness varies considerably across models and categories. Some models refuse or provide harmless alternatives, while others comply with harmful prompts or provide warnings while still giving actionable information. This demonstrates that general-purpose alignment does not necessarily guarantee harmlessness in software-engineering contexts.

Chapter 5 showed that code LLMs are vulnerable to memorisation and extraction attacks. Since code corpora may contain secrets, credentials, identifying information, proprietary code, and non-permissively licensed code, memorisation creates privacy, security, and intellectual-property risks. These risks are not hypothetical side effects; they arise directly from the way models are trained on large, often opaque software datasets.

Chapter 6 extended the memorisation argument from privacy to reliability. If models reproduce code from training data, then memorisation can affect not only what private data is leaked, but also what bugs are propagated. The exposure-aware bug-vs-fix evaluation shows that models may generate buggy variants even when fixed variants exist, and that prior exposure can skew behaviour. This creates a software-quality risk: generated code may inherit defects from the training corpus, and conventional benchmark scores may obscure that effect if they do not account for exposure.

> Code-oriented LLMs introduce security, privacy, safety, and reliability risks that are directly connected to their training data, model design, alignment, and deployment context. They may provide harmful technical assistance, leak memorised training data, and reproduce buggy or insecure code patterns. LLMs for code should therefore not be evaluated only by syntactic validity, functional correctness, or benchmark performance, but also by harmfulness, memorisation, exposure, leakage, and the propagation of unreliable code.

7.2 Cross-Cutting Findings

Several findings recur across the thesis. The first is that capability and risk are coupled. In security settings, capability is inherently dual-use. A model that supports reverse engineering can help defenders analyse malware or legacy binaries, but similar techniques can also help attackers understand proprietary software or search for exploitable weaknesses. A model that can reason through cyber tasks can assist authorised testing, but may also lower the barrier to misuse. A model that learns from large software corpora can improve developer productivity, but may also memorise and reproduce sensitive or defective code.

The second cross-cutting finding is that realistic evaluation matters. Claims should be based on tasks that approximate real workflows and preserve meaningful constraints.

The third finding is that artefacts matter. The datasets, benchmarks, prompt sets, execution traces, evaluation scripts, and analysis pipelines developed in this thesis are part of the research contribution because they make capability and risk claims inspectable, repeatable, and extensible.

7.3 Data Sharing and Cumulative Research

A recurring goal of this thesis has been to contribute not only findings, but also reusable research artefacts. The chapters introduce datasets, models, benchmark harnesses, prompt sets, scoring rubrics, evaluation scripts, and analysis pipelines. These artefacts support replication, comparison, and follow-on work. They also make it possible to evaluate new models as the field changes.

This is especially important in LLM research because opaque datasets and evaluation practices limit interpretability. Without shared artefacts, it becomes difficult to determine whether improvements are due to better models, different data, benchmark contamination, evaluation choices, or implementation details. In software security, this problem is amplified because models and benchmarks evolve quickly, and because many claims about capability or risk are sensitive to the exact task setting.

At the same time, data sharing in this area cannot be treated as an unconditional good. Several artefacts in this thesis concern dual-use tasks, or sensitive data. Responsible sharing therefore requires selective openness, documentation, and safeguards. Some artefacts can be released openly; others require redaction, access restrictions, or warnings about intended use. The goal is to preserve scientific utility while reducing foreseeable harm.

Responsible artefact release is therefore part of the thesis contribution. It supports cumulative research while acknowledging the security and privacy constraints of the domain. This returns to one of the goals stated in the introduction: producing and responsibly sharing datasets, benchmarks, and evaluation pipelines that make LLM research in software security more reproducible and cumulative.

The artefacts released with Chapter 2 illustrate this role well. The chapter introduced the CAPYBARA dataset and the BinT5 models, both of which have gone on to be reused in later research on binary-code understanding. He et al. drew on CAPYBARA to train and evaluate RETROFIT, a model that shares the same backbone as BinT5 [207], while Lu et al. incorporated CAPYBARA as a source of benign binary summaries in the MalS dataset and

adopted BinT5 as a baseline [208]. Beyond these, a number of further studies have used BinT5 as a baseline for binary-code summarisation and related evaluation tasks [9, 209–213].

By contrast, the artefacts accompanying Chapter 5 attracted comparatively little reuse, even though the paper itself was cited more often. Chu et al. repurposed the memorised samples as a retained set for measuring retention after unlearning [214]; Chen and Jiang adapted the data extraction strategies and the targeted data extraction security game [215]; and Jiang et al. reused the same evaluation method as a baseline [216]. This gap likely stems from the fact that the dataset assembled in Chapter 5 was considerably smaller and more model-specific than CAPYBARA [24]. Whereas CAPYBARA comprised thousands of mined and processed samples that other researchers could readily take up, the dataset from Chapter 5 was narrower and far less transferable owing to its problem domain.

7.4 Broader Implications

For software engineering, the main implication is that LLM evaluation should move beyond task performance alone. Accuracy, pass rates, and benchmark scores remain useful, but they are insufficient for systems that may be deployed in high-impact software workflows. Evaluations should also consider harmlessness, data exposure, memorisation, reliability, and the conditions under which models fail. This is especially important for coding assistants, where generated code may be incorporated into production systems with limited inspection.

For cybersecurity, the thesis shows that claims about LLM cyber capability require careful empirical grounding. Demonstrations of individual successes are not enough. LLM agents should be evaluated in controlled, realistic environments, with transparent scoring methods and reproducible artefacts. Partial-credit evaluation is particularly important because current agents often make measurable intermediate progress without completing full tasks. Understanding this partial progress is necessary for tracking capability growth and assessing future risks.

For LLM research, code and security domains provide important testbeds for studying memorisation, alignment, robustness, and dual-use behaviour. These domains expose tensions that may be less visible in general natural-language benchmarks. In code, training-data exposure can have legal, privacy, and reliability consequences. In security, helpfulness and harmfulness are often separated by user intent and deployment context. These properties make software security a demanding but valuable setting for evaluating LLMs.

For research practice, the thesis reinforces the importance of shared artefacts. Reusable datasets and benchmarks enable cumulative progress, but openness must be balanced against privacy and dual-use risks.

7.5 Societal Impact

Software is a foundational artefact. It underlies communication, finance, healthcare, transportation, energy systems, science, public administration, and national security. As AI systems become increasingly integrated into software development and security work-

flows, the behaviour of code-oriented LLMs can affect society at large. Their impact is not limited to developers or researchers. It can influence the security of digital infrastructure, the privacy of individuals, the reliability of software systems, and the distribution of technical capability between defenders and attackers.

The positive societal potential is substantial. LLMs can support software developers and security analysts. They reduce the effort required to understand unfamiliar code, analyse binaries, triage vulnerabilities, and reason about complex systems. This could help organisations maintain legacy software, respond to incidents, and identify weaknesses earlier. In contexts where expertise is scarce, LLM-based assistance may make some forms of analysis more accessible.

However, these benefits are accompanied by significant risks. LLMs can amplify attacker capability by helping users reason through exploitation steps, generate malicious code, or adapt known techniques. They can leak memorised training data, including sensitive or proprietary information. They can generate harmful assistance under the guise of programming help. They can also reduce software quality if their outputs reproduce unreliable or buggy code patterns. These risks matter because software failures and security breaches can have consequences beyond the immediate system in which the code is deployed.

The societal value of LLMs in software security therefore depends on rigorous evaluation, transparent artefacts, responsible sharing, and risk-aware deployment. LLMs should not be adopted in security-sensitive settings solely because they appear capable in demonstrations. Their deployment should be guided by evidence about their limitations, failure modes, exposure risks, and misuse potential. Human oversight remains essential, particularly where outputs may affect critical systems, sensitive data, or offensive cyber capability.

7.6 Limitations and Future Work

The findings of this thesis should be understood as evidence about trajectories rather than final capability limits. LLMs, agent architectures, alignment methods, and evaluation practices are evolving rapidly. Models that are limited today may be surpassed quickly. Conversely, risks that appear manageable under current capabilities may become more severe as models become more autonomous, more widely deployed, or more deeply integrated into development environments.

Future work should therefore continue evaluating newer models and richer agent systems. In the context of cyber agents, future benchmarks should include longer-horizon tasks, more varied environments, and more realistic operational constraints. They should also study how different agent architectures, memory systems, planning strategies, and tool interfaces affect performance. Current agents often fail not because they lack basic tool knowledge, but because they struggle with persistence, state tracking, adaptation, and long-horizon reasoning.

Future work should also improve evaluation methods for harmfulness and safety in software engineering. Software-specific harmfulness remains difficult to define because the same technical information can be benign or harmful depending on context. Better evaluations should consider user intent, deployment setting, code functionality, and the practical usefulness of the generated response. They should also study interventions,

including safer training data, domain-specific alignment, output filtering, and model behaviours under adversarial prompting.

For memorisation and exposure, future work should develop stronger membership-inference and extraction techniques, as well as better mitigation strategies. More precise exposure measurements would help distinguish memorisation from generalisation.

Finally, future work should further connect exposure-aware evaluation to software reliability. If LLM-generated code is shaped by prior exposure to buggy or fixed examples, then evaluation protocols should account for training-data provenance and benchmark contamination. Developers and researchers need methods that can identify when generated code is likely to be memorised, when it is likely to reproduce known defects, and when alternative completions should be preferred.

7.7 Closing Remarks

AI will become increasingly embedded in software and security workflows. Its use will not be confined to isolated coding assistance; it will influence how software is written, reviewed, and secured. This makes it necessary to understand both what AI can do and what risks it introduces.

In this thesis, we have contributed evidence for this dual perspective. We showed that LLMs can be adapted to realistic software-security tasks, but that their capabilities depend on domain-specific data, task design, and evaluation methodology. We also showed that code-oriented LLMs can provide harmful assistance, leak memorised data, and reproduce buggy code. These findings support the central argument of the thesis: progress in applying LLMs to software security requires capability and risk to be evaluated together, under realistic and domain-specific conditions.

Bibliography

References

[1] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Measuring github copilot's impact on productivity. *Communications of the ACM*, 67(3):54–63, 2024.

[2] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8), December 2024. ISSN 1049-331X. doi: 10.1145/3695988. URL https://doi.org/10.1145/3695988.

[3] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: multi-token code completion by jointly learning from structure and naming sequences. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 401–412, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510172. URL https://doi.org/10.1145/3510003.3510172.

[4] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[5] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.

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

[7] Tiago Conceição and Nuno Cruz. Evaluation of the maturity of llms in the cybersecurity domain: T. conceição, n. cruz. *International Journal of Information Security*, 24(5):197, 2025.

[8] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: a benchmark for code reasoning, understanding

and execution. In *Proceedings of the 41st International Conference on Machine Learning*, pages 16568–16621, 2024.

[9] Xiuwei Shang, Guoqiang Chen, Shaoyin Cheng, Benlong Wu, Li Hu, Gangyang Li, Weiming Zhang, and Nenghai Yu. Binmetric: A comprehensive binary code analysis benchmark for large language models. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pages 7715–7723, 2025.

[10] Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, et al. Cve-bench: A benchmark for ai agents' ability to exploit real-world web application vulnerabilities. In *International Conference on Machine Learning*, pages 79850–79867. PMLR, 2025.

[11] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. An observational investigation of reverse engineers' process and mental models. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI EA '19, page 1–6, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359719. doi: 10.1145/3290607.3313040. URL https://doi.org/10.1145/3290607.3313040.

[12] Yuwen Zou, Jia Liu, and Wenjun Fan. Ctfagent: An llm-powered agent for ctf challenge solving. *Journal of Information Security and Applications*, 96:104305, 2026.

[13] Valentin Hartmann, Anshuman Suri, Vincent Bindschaedler, David Evans, Shruti Tople, and Robert West. Sok: Memorization in general-purpose large language models, 2023. URL https://arxiv.org/abs/2310.18362.

[14] Jiaming Ji, Mickel Liu, Josef Dai, Xuehai Pan, Chi Zhang, Ce Bian, Boyuan Chen, Ruiyang Sun, Yizhou Wang, and Yaodong Yang. Beavertails: Towards improved safety alignment of llm via a human-preference dataset. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 24678–24704. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/4dbb61cb68671edc4ca3712d70083b9f-Paper-Datasets_and_Benchmarks.pdf.

[15] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

[16] Maximilian Mozes, Xuanli He, Bennett Kleinberg, and Lewis D Griffin. Use of llms for illicit purposes: Threats, prevention measures, and vulnerabilities. *CoRR*, 2023.

[17] Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. Bugs in large language models generated code: An empirical study. *Empirical Software Engineering*, 30(3):65, 2025.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[19] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, apr 2016. ISSN 0001-0782. doi: 10.1145/2902362. URL https://doi.org/10.1145/2902362.

[20] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685.

[21] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint*, 2022.

[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

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

[24] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 260–271, 2023. doi: 10.1109/SANER56733.2023.00033.

[25] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

[26] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

[27] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

[28] Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 'smolagents': a smol library to build great agentic systems. https://github.com/huggingface/smolagents, 2025.

[29] Ali Al-Kaswan and Maliheh Izadi. The (ab) use of open source code to train large language models. In *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*, pages 9–10. IEEE, 2023.

[30] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650, 2021.

[31] Hongsheng Hu, Zoran Salcic, Lichao Sun, Gillian Dobbie, Philip S Yu, and Xuyun Zhang. Membership inference attacks on machine learning: A survey. *ACM Computing Surveys (CSUR)*, 54(11s):1–37, 2022.

[32] Fatemehsadat Mireshghallah, Kartik Goyal, Archit Uniyal, Taylor Berg-Kirkpatrick, and Reza Shokri. Quantifying privacy risks of masked language models using membership inference attacks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 8332–8347, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.570.

[33] Ali Al-Kaswan, Maliheh Izadi, and Arie Van Deursen. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[34] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, Donggyun Han, and David Lo. Unveiling memorization in code models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[35] Anjan Karmakar, Julian Aron Prenner, Marco D'Ambros, and Romain Robbes. Codex hacks hackerrank: Memorization issues and a framework for code synthesis evaluation. *arXiv preprint arXiv:2212.02684*, 2022.

[36] Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Ben Mann, Nova DasSarma, et al. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*, 2021.

[37] Mohammed Latif Siddiq, Arvin Islam-Gomes, Natalie Sekerak, and Joanna Santos. Large language models for software engineering: A reproducibility crisis. *arXiv preprint arXiv:2512.00651*, 2025.

[38] Stella Biderman, Usvsn Prashanth, Lintang Sutawika, Hailey Schoelkopf, Quentin Anthony, Shivanshu Purohit, and Edward Raff. Emergent and predictable memorization in large language models. *Advances in Neural Information Processing Systems*, 36:28072–28090, 2023.

[39] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th international conference on mining software repositories*, pages 573–577, 2020.

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

[41] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. On the evaluation of neural code summarization. In *Proceedings of the 44th international conference on software engineering*, pages 1597–1608, 2022.

[42] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), nov 2020. doi: 10.1145/3428293. URL https://doi.org/10.1145/3428293.

[43] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Comput. Surv.*, 48(4), May 2016. ISSN 0360-0300. doi: 10.1145/2896499. URL https://doi.org/10.1145/2896499.

[44] Ligeng Chen, Zhongling He, and Bing Mao. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 88–98, 2020. doi: 10.1109/DSN48063.2020.00028.

[45] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[46] Toufique Ahmed, Premkumar Devanbu, and Anand Ashok Sawant. Learning to find usage of library functions in optimized binaries. *IEEE Transactions on Software Engineering*, pages 1–1, 2021. doi: 10.1109/TSE.2021.3106572.

[47] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC conference on computer and communications security*, pages 1631–1645, 2022.

[48] Daniel Lehmann and Michael Pradel. Finding the dwarf: recovering precise types from webassembly binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 410–425, 2022.

[49] Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. Neutron: an attention-based neural decompiler. *Cybersecurity*, 4:5, 03 2021. doi: 10.1186/s42400-021-00070-0.

[50] Chunyan Zhang, Junchao Wang, Qinglei Zhou, Ting Xu, Ke Tang, Hairen Gui, and Fudong Liu. A survey of automatic source code summarization. *Symmetry*, 14(3), 2022. ISSN 2073-8994. URL https://www.mdpi.com/2073-8994/14/3/471.

[51] Toufique Ahmed and Premkumar Devanbu. Multilingual training for software engineering. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1443–1455. IEEE, 2022.

[52] Casey Casalnuovo, Earl T Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. A theory of dual channel constraints. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, pages 25–28, 2020.

[53] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3931–3937, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1394. URL https://aclanthology.org/N19-1394.

[54] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[55] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174, 2008.

[56] M Tim Jones. Optimization in gcc. *Linux journal*, 2005(131):11, 2005.

[57] Sandrine Blazy and Stéphanie Riaud. Measuring the robustness of source program obfuscation: Studying the impact of compiler optimizations on the obfuscation of c programs. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, page 123–126, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450322782. doi: 10.1145/2557547.2557577. URL https://doi.org/10.1145/2557547.2557577.

[58] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 659–676. IEEE, 2021.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6000–6010, 2017.

[60] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0262010976.

[61] Jürgen Schmidhuber, Sepp Hochreiter, et al. Long short-term memory. *Neural Comput*, 9(8):1735–1780, 1997.

[62] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[63] Liu Zhuang, Lin Wayne, Shi Ya, and Zhao Jun. A robustly optimized BERT pre-training approach with post-training. In *Proceedings of the 20th Chinese National Conference on Computational Linguistics*, pages 1218–1227, Huhhot, China, August 2021. Chinese Information Processing Society of China. URL `https://aclanthology.org/2021.ccl-1.108`.

[64] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186, 2019.

[65] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. Fasttext.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016.

[66] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, Athens Greece, October 2019. ACM. ISBN 978-1-4503-6995-4. doi: 10.1145/3359591.3359735. URL `https://dl.acm.org/doi/10.1145/3359591.3359735`.

[67] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL `http://jmlr.org/papers/v21/20-074.html`.

[68] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual*

Meeting of the Association for Computational Linguistics, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040.

[69] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/W04-1013.

[70] Alon Lavie and Michael J Denkowski. The meteor metric for automatic evaluation of machine translation. *Machine translation*, 23(2):105–115, 2009.

[71] Boxing Chen and Colin Cherry. A systematic comparison of smoothing techniques for sentence-level BLEU. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 362–367, Baltimore, Maryland, USA, June 2014. Association for Computational Linguistics. doi: 10.3115/v1/W14-3346. URL https://aclanthology.org/W14-3346.

[72] Hieu Hoang and Philipp Koehn. Design of the moses decoder for statistical machine translation. In *Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 58–65, 2008.

[73] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. In *ICLR*, 2021.

[74] Jim Alves-Foss and Jia Song. Function boundary detection in stripped binaries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 84–96, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376280. doi: 10.1145/3359789.3359825. URL https://doi.org/10.1145/3359789.3359825.

[75] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2091–2100, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/allamanis16.html.

[76] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. *Reassessing Automatic Evaluation Metrics for Code Summarization Tasks*, page 1105–1116. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450385626. URL https://doi.org/10.1145/3468264.3468588.

[77] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 36–47, 2022.

[78] Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. Variable name recovery in decompiled binary code using constrained masked language modeling. *arXiv preprint arXiv:2103.12801*, 2021.

[79] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. Language models for code completion: A practical evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639138. URL https://doi.org/10.1145/3597503.3639138.

[80] Mohammed Latif Siddiq, Joanna Cecilia da Silva Santos, Sajith Devareddy, and Anna Muller. Sallm: Security assessment of generated code, 2024.

[81] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Communications of the ACM*, 68(2):96–105, 2025.

[82] Ismayil Hasanov, Seppo Virtanen, Antti Hakkala, and Jouni Isoaho. Application of large language models in cybersecurity: A systematic literature review. *IEEE access*, 12:176751–176778, 2024.

[83] Miles Brundage, Shahar Avin, Jack Clark, Helen Toner, Peter Eckersley, Ben Garfinkel, Allan Dafoe, Paul Scharre, Thomas Zeitzoff, Bobby Filar, et al. The malicious use of artificial intelligence: Forecasting, prevention, and mitigation. *arXiv preprint arXiv:1802.07228*, 2018.

[84] Alba Thaqi, Arbena Musa, and Blerim Rexha. Leveraging ai for ctf challenge optimization. In *2024 5th International Conference on Communications, Information, Electronic and Energy Systems (CIEES)*, pages 1–5. IEEE, 2024.

[85] Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, Farshad Khorrami, et al. Nyu ctf bench: A scalable open-source benchmark dataset for evaluating llms in offensive security. *Advances in Neural Information Processing Systems*, 37:57472–57498, 2024.

[86] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.

[87] Thiem Nguyen Ba, Binh Nguyen Thanh, and Viet-Trung Tran. Covernexus: Multiagent llm system for automated code coverage enhancement. In *International Symposium on Information and Communication Technology*, pages 472–484. Springer, 2024.

[88] Tanwir Ahmad, Matko Butkovic, and Dragos Truscan. Using reinforcement learning for security testing: A systematic mapping study. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 208–216. IEEE, 2025.

[89] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024.

[90] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37: 50528–50652, 2024.

[91] Ali Al-Kaswan, Sebastian Deatc, Begüm Koç, Arie van Deursen, and Maliheh Izadi. Code red! on the harmfulness of applying off-the-shelf large language models to programming tasks. *Proceedings of the ACM on Software Engineering*, 2(FSE):2477–2499, 2025.

[92] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. {PentestGPT}: Evaluating and harnessing large language models for automated penetration testing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 847–864, 2024.

[93] Talor Abramovich, Meet Udeshi, Minghao Shao, Kilian Lieret, Haoran Xi, Kimberly Milner, Sofija Jancheska, John Yang, Carlos E Jimenez, Farshad Khorrami, Prashanth Krishnamurthy, Brendan Dolan-Gavitt, Muhammad Shafique, Karthik R Narasimhan, Ramesh Karri, and Ofir Press. EnIGMA: Interactive tools substantially assist LM agents in finding security vulnerabilities. In *Forty-second International Conference on Machine Learning*, 2025. URL `https://openreview.net/forum?id=Of3wZhVv1R`.

[94] Theo Combe, Antony Martin, and Roberto Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016.

[95] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024.

[96] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[97] Jianxun Wang and Yixiang Chen. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289. IEEE, 2023.

[98] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg

Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[99] Usman Anwar, Abulhair Saparov, Javier Rando, Daniel Paleka, Miles Turpin, Peter Hase, Ekdeep Singh Lubana, Erik Jenner, Stephen Casper, Oliver Sourbut, et al. Foundational challenges in assuring alignment and safety of large language models. *CoRR*, 2024.

[100] Deep Ganguli, Liane Lovitt, Jackson Kernion, Amanda Askell, Yuntao Bai, Saurav Kadavath, Ben Mann, Ethan Perez, Nicholas Schiefer, Kamal Ndousse, et al. Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned. *CoRR*, 2022.

[101] Tianhao Shen, Renren Jin, Yufei Huang, Chuang Liu, Weilong Dong, Zishan Guo, Xinwei Wu, Yan Liu, and Deyi Xiong. Large language model alignment: A survey. *arXiv preprint arXiv:2309.15025*, 2023.

[102] Oliver Klingefjord, Ryan Lowe, and Joe Edelman. What are human values, and how do we align ai to them? *arXiv preprint arXiv:2404.10636*, 2024.

[103] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL `https://www.science.org/doi/abs/10.1126/science.abq1158`.

[104] Paul W. McBurney and Collin McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, 2016. doi: 10.1109/TSE.2015.2465386.

[105] Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, et al. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724*, 2023.

[106] Hakan Inan, Kartikeya Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, Qing Hu, Brian Fuller, Davide Testuggine, et al.

Llama guard: Llm-based input-output safeguard for human-ai conversations. *arXiv preprint arXiv:2312.06674*, 2023.

[107] Yuxia Wang, Haonan Li, Xudong Han, Preslav Nakov, and Timothy Baldwin. Do-not-answer: Evaluating safeguards in LLMs. In Yvette Graham and Matthew Purver, editors, *Findings of the Association for Computational Linguistics: EACL 2024*, pages 896–911, St. Julian's, Malta, March 2024. Association for Computational Linguistics. URL https://aclanthology.org/2024.findings-eacl.61.

[108] Simone Tedeschi, Felix Friedrich, Patrick Schramowski, Kristian Kersting, Roberto Navigli, Huu Nguyen, and Bo Li. Alert: A comprehensive benchmark for assessing large language models' safety through red teaming. *CoRR*, 2024.

[109] Miles Brundage, Shahar Avin, Jasmine Wang, Haydn Belfield, Gretchen Krueger, Gillian Hadfield, Heidy Khlaaf, Jingying Yang, Helen Toner, Ruth Fong, et al. Toward trustworthy ai development: mechanisms for supporting verifiable claims. *arXiv preprint arXiv:2004.07213*, 2020.

[110] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

[111] Jan Leike. Distinguishing three alignment taxes, Dec 2022. URL https://aligned.substack.com/p/three-alignment-taxes.

[112] Rafik Hamza and Hilmil Pradana. A survey of intellectual property rights protection in big data applications. *Algorithms*, 15(11), 2022. ISSN 1999-4893. doi: 10.3390/a15110418. URL https://www.mdpi.com/1999-4893/15/11/418.

[113] Irina Suslina and Yana Fetisova. Managing software as intellectual property: protection and commercialization. *Procedia Computer Science*, 213:144–148, 2022. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2022.11.049. URL https://www.sciencedirect.com/science/article/pii/S1877050922017409. 2022 Annual International Conference on Brain-Inspired Cognitive Architectures for Artificial Intelligence: The 13th Annual Meeting of the BICA Society.

[114] Rabia Tahir. A study on malware and malware detection techniques. *International Journal of Education and Management Engineering*, 8:20–30, 03 2018. doi: 10.5815/ijeme.2018.02.03.

[115] Jyoti Landage and M. P. Wankhade. Malware and malware detection techniques: A survey. *International Journal of Engineering Research and Technology (IJERT)*, 2 (12):2278–0181, 2013.

[116] Maanak Gupta, Charankumar Akiri, Kshitiz Aryal, Eli Parker, and Lopamudra Praharaj. From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy. *IEEE Access*, 11:80218–80245, 2023. doi: 10.1109/ACCESS.2023.3300381.

[117] Yan Liu, Xiaokang Chen, Yan Gao, Zhe Su, Fengji Zhang, Daoguang Zan, Jian-Guang Lou, Pin-Yu Chen, and Tsung-Yi Ho. Uncovering and quantifying social biases in code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 2368–2380. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/071a637d41ea290ac4360818a8323f33-Paper-Conference.pdf.

[118] Nikolaos Pitropakis, Emmanouil Panaousis, Thanassis Giannetsos, Eleftherios Anastasiadis, and George Loukas. A taxonomy and survey of attacks against machine learning. *Computer Science Review*, 34:100199, 2019.

[119] Neeraj Varshney, Pavel Dolin, Agastya Seth, and Chitta Baral. The art of defending: A systematic evaluation and analysis of LLM defense strategies on safety and over-defensiveness. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13111–13128, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.776. URL https://aclanthology.org/2024.findings-acl.776/.

[120] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[121] Andrew F. Hayes and Klaus Krippendorff. Answering the call for a standard reliability measure for coding data. *Communication Methods and Measures*, 1(1): 77–89, 2007. doi: 10.1080/19312450709336664. URL https://doi.org/10.1080/19312450709336664.

[122] Klaus Krippendorff. Estimating the reliability, systematic error and random error of interval data. *Educational and psychological measurement*, 30(1):61–70, 1970.

[123] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-sklearn 2.0: Hands-free automl via meta-learning. *Journal of Machine Learning Research*, 23(261):1–61, 2022.

[124] Toufique Ahmed, Premkumar Devanbu, Christoph Treude, and Michael Pradel. Can llms replace manual annotation of software engineering artifacts? In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 526–538. IEEE, 2025.

[125] Ryan Teknium, Jeffrey Quesnelle, and Chen Guang. Hermes 3 technical report. *arXiv preprint arXiv:2408.11857*, 2024.

[126] Teknium. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants, 2023. URL https://huggingface.co/datasets/teknium/OpenHermes-2.5.

[127] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In

Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, pages 2623–2631, 2019.

[128] Daphne Theodorakopoulos, Frederic Stahl, and Marius Lindauer. Hyperparameter importance analysis for multi-objective automl. In *ECAI 2024*, pages 1100–1107. IOS Press, 2024.

[129] Lewis Tunstall, Nils Reimers, Unso Eun Seo Jo, Luke Bates, Daniel Korat, Moshe Wasserblat, and Oren Pereg. Efficient few-shot learning without prompts. In *NeurIPS ENLSP Workshop*, 2022.

[130] Mohammad Atari, Mona J Xue, Peter S Park, Damián E Blasi, and Joseph Henrich. Which humans?, Sep 2023. URL osf.io/preprints/psyarxiv/5b26t.

[131] Rebecca L Johnson, Giada Pistilli, Natalia Menédez-González, Leslye Denisse Dias Duran, Enrico Panai, Julija Kalpokiene, and Donald Jay Bertulfo. The ghost in the machine has an american accent: value conflict in gpt-3. *arXiv preprint arXiv:2203.07785*, 2022.

[132] Jan Betley, Daniel Chee Hian Tan, Niels Warncke, Anna Sztyber-Betley, Xuchan Bao, Martín Soto, Nathan Labenz, and Owain Evans. Emergent misalignment: Narrow finetuning can produce broadly misaligned LLMs. In *ICLR 2025 Workshop on Foundation Models in the Wild*, 2025. URL https://openreview.net/forum?id=bwx3VNjLzX.

[133] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*, 2022.

[134] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022.

[135] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35: 24824–24837, 2022.

[136] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[137] Nicholas Carlini, Steve Chien, Milad Nasr, Shuang Song, Andreas Terzis, and Florian Tramer. Membership inference attacks from first principles. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1897–1914. IEEE, 2022.

[138] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations*, 2022.

[139] Nicholas Carlini, Matthew Jagielski, Chiyuan Zhang, Nicolas Papernot, Andreas Terzis, and Florian Tramer. The privacy onion effect: Memorization is relative. *Advances in Neural Information Processing Systems*, 35:13263–13276, 2022.

[140] Peter Henderson, Xuechen Li, Dan Jurafsky, Tatsunori Hashimoto, Mark A Lemley, and Percy Liang. Foundation models and fair use. *Journal of Machine Learning Research*, 24(400):1–79, 2023.

[141] Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. Coprotector: Protect open-source code against unauthorized training usage with data poisoning. In *Proceedings of the ACM Web Conference 2022*, pages 652–660, 2022.

[142] Kushal Tirumala, Aram Markosyan, Luke Zettlemoyer, and Armen Aghajanyan. Memorization without overfitting: Analyzing the training dynamics of large language models. *Advances in Neural Information Processing Systems*, 35:38274–38290, 2022.

[143] Vitaly Feldman. Does learning require memorization? a short tale about a long tail. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 954–959, 2020.

[144] Zitao Chen and Karthik Pattabiraman. Overconfidence is a dangerous thing: Mitigating membership inference attacks by enforcing less confident prediction. In *Network and Distributed System Security (NDSS) Symposium*, 2024.

[145] Nils Lukas, Ahmed Salem, Robert Sim, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. Analyzing leakage of personally identifiable information in language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 346–363. IEEE Computer Society, 2023.

[146] Shotaro Ishihara. Training data extraction from pre-trained language models: A survey, 2023.

[147] Madiha Zahrah Choksi and David Goedicke. Whose text is it anyway? exploring bigcode, intellectual property, and ethics, 2023. URL `https://arxiv.org/abs/2304.02839`.

[148] Nicolas Carlini, Jamie Hayes, Milad Nasr, Matthew Jagielski, Vikash Sehwag, Florian Tramer, Borja Balle, Daphne Ippolito, and Eric Wallace. Extracting training data from diffusion models. In *32nd USENIX security symposium (USENIX Security 23)*, pages 5253–5270, 2023.

[149] Daphne Ippolito, Florian Tramer, Milad Nasr, Chiyuan Zhang, Matthew Jagielski, Katherine Lee, Christopher Choquette Choo, and Nicholas Carlini. Preventing generation of verbatim memorization in language models gives a false sense of pri-

vacy. In *Proceedings of the 16th International Natural Language Generation Conference*, Prague, Czechia, September 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.inlg-main.3. URL https://aclanthology.org/2023.inlg-main.3.

[150] Setu Kumar Basak, Lorenzo Neil, Bradley Reaves, and Laurie Williams. Secretbench: A dataset of software secrets. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 347–351. IEEE, 2023.

[151] Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. Targeted attack on gpt-neo for the satml language model data extraction challenge. *ArXiv*, abs/2302.07735, 2023.

[152] Alina Oprea and Apostol Vassilev. Adversarial machine learning: A taxonomy and terminology of attacks and mitigations. Technical report, National Institute of Standards and Technology, 2023.

[153] Christopher A. Choquette-Choo, Florian Tramer, Nicholas Carlini, and Nicolas Papernot. Label-only membership inference attacks. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 1964–1974. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/choquette-choo21a.html.

[154] Jasper Tan, Daniel LeJeune, Blake Mason, Hamid Javadi, and Richard G. Baraniuk. A blessing of dimensionality in membership inference through regularization. In Francisco Ruiz, Jennifer Dy, and Jan-Willem van de Meent, editors, *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, volume 206 of *Proceedings of Machine Learning Research*, pages 10968–10993. PMLR, 25–27 Apr 2023. URL https://proceedings.mlr.press/v206/tan23b.html.

[155] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2017.

[156] Benjamin Hilprecht, Martin Härterich, and Daniel Bernau. Monte carlo and reconstruction membership inference attacks against generative models. *Proc. Priv. Enhancing Technol.*, 2019(4):232–249, 2019.

[157] Sorami Hisamoto, Matt Post, and Kevin Duh. Membership inference attacks on sequence-to-sequence models: Is my data in your machine translation system? *Transactions of the Association for Computational Linguistics*, 8:49–63, 2020.

[158] Jan Dubiński, Antoni Kowalczuk, Stanisław Pawlak, Przemyslaw Rokita, Tomasz Trzciński, and Paweł Morawiecki. Towards more realistic membership inference attacks on large diffusion models. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 4860–4869, 2024.

[159] Myung Gyo Oh, Leo Hyun Park, Jaeuk Kim, Jaewoo Park, and Taekyoung Kwon. Membership inference attacks with token-level deduplication on korean language models. *IEEE Access*, 11:10207–10217, 2023.

[160] Jie Huang, Hanyin Shao, and Kevin Chen-Chuan Chang. Are large pre-trained language models leaking your personal information? In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 2038–2047, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.findings-emnlp.148.

[161] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[162] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow, march 2021. *URL https://doi. org/10.5281/zenodo*, 5297715(5):3, 2021.

[163] Hannah Brown, Katherine Lee, Fatemehsadat Mireshghallah, Reza Shokri, and Florian Tramèr. What does it mean for a language model to preserve privacy? In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*, pages 2280–2292, 2022.

[164] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

[165] Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. Pythia: A suite for analyzing large language models across training and scaling, 2023.

[166] Swapnil Sharma, Nikita Anand, and Kranthi Kiran G. V. Stochastic code generation, 2023.

[167] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. CERT: Continual pre-training on sketches for library-oriented code generation. In *The 2022 International Joint Conference on Artificial Intelligence*, 2022.

[168] Nikhil Kandpal, Eric Wallace, and Colin Raffel. Deduplicating training data mitigates privacy risks in language models. In *International Conference on Machine Learning*, pages 10697–10707. PMLR, 2022.

[169] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445, 2022.

[170] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig

Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[171] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

[172] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based Software Engineering Agents. *CoRR*, January 2024. URL https://openreview.net/forum?id=WYK8E00dm2.

[173] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639187. URL https://doi.org/10.1145/3597503.3639187.

[174] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, 50(9):2254–2268, 2024. doi: 10.1109/TSE.2024.3428972.

[175] Kevin Jesse, Toufique Ahmed, Premkumar T Devanbu, and Emily Morgan. Large language models and simple, stupid bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 563–575. IEEE, 2023.

[176] Ahmad Mohsin, Helge Janicke, Adrian Wood, Iqbal H Sarker, Leandros Maglaras, and Naeem Janjua. Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms. *arXiv preprint arXiv:2406.12513*, 2024.

[177] Li Zhong and Zilong Wang. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 21841–21849, 2024.

[178] Mahmoud Jahanshahi and Audris Mockus. Cracks in the stack: Hidden vulnerabilities and licensing risks in llm pre-training datasets. In *2025 IEEE/ACM International*

Workshop on Large Language Models for Code (LLM4Code), pages 104–111. IEEE, 2025.

[179] Avi Schwarzschild, Zhili Feng, Pratyush Maini, Zachary Lipton, and J Zico Kolter. Rethinking llm memorization through the lens of adversarial compression. *Advances in Neural Information Processing Systems*, 37:56244–56267, 2024.

[180] Jiaolong Kong, Xiaofei Xie, and Shangqing Liu. Demystifying memorization in llm-based program repair via a general hypothesis testing framework. *Proceedings of the ACM on Software Engineering*, 2(FSE):2712–2734, 2025.

[181] Matthieu Meeus, Igor Shilov, Shubham Jain, Manuel Faysse, Marek Rei, and Yves-Alexandre de Montjoye. Sok: Membership inference attacks on llms are rushing nowhere (and how to fix it). In *2025 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pages 385–401. IEEE, 2025.

[182] Yao Wan, Guanghua Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Pan Zhou, Hai Jin, and Lichao Sun. Does your neural code completion model use my code? a membership inference approach. *ACM Transactions on Software Engineering and Methodology*, 35(3):1–24, 2026.

[183] Jonathan Katzy, Razvan Popescu, Arie Van Deursen, and Maliheh Izadi. An exploratory investigation into code license infringements in large language model training datasets. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 74–85, 2024.

[184] Jonathan Katzy, Razvan Mihai Popescu, Arie van Deursen, and Maliheh Izadi. The heap: A contamination-free multilingual code dataset for evaluating large language models. In *Proceedings of the 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering*, 2025. URL `https://arxiv.org/abs/2501.09653`.

[185] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[186] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56, 2017.

[187] Shenglai Zeng, Yaxin Li, Jie Ren, Yiding Liu, Han Xu, Pengfei He, Yue Xing, Shuaiqiang Wang, Jiliang Tang, and Dawei Yin. Exploring memorization in fine-tuned language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3917–3948, 2024.

[188] David Reid, Mahmoud Jahanshahi, and Audris Mockus. The extent of orphan vulnerabilities from code reuse in open source software. In *Proceedings of the 44th international conference on software engineering*, pages 2104–2115, 2022.

[189] Fatemehsadat Mireshghallah, Archit Uniyal, Tianhao Wang, David Evans, and Taylor Berg-Kirkpatrick. Memorization in nlp fine-tuning methods. In *First Workshop on Pre-training: Perspectives, Pitfalls, and Paths Forward at ICML 2022*, 2022.

[190] Xingjun Ma, Yifeng Gao, Yixu Wang, Ruofan Wang, Xin Wang, Ye Sun, Yifan Ding, Hengyuan Xu, Yunhao Chen, Yunhan Zhao, et al. Safety at scale: A comprehensive survey of large model safety. *CoRR*, 2025.

[191] Dario Di Palma, Felice Antonio Merra, Maurizio Sfilio, Vito Walter Anelli, Fedelucio Narducci, and Tommaso Di Noia. Do llms memorize recommendation datasets? a preliminary study on movielens-1m. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2582–2586, 2025.

[192] Fabio Salerno, Ali Al-Kaswan, and Maliheh Izadi. How much do code language models remember? an investigation on data extraction attacks before and after fine-tuning. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 465–477. IEEE, 2025.

[193] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Communications of the ACM*, 61(10):29–31, 2018.

[194] Nikita Pavlichenko, Iurii Nazarov, Ivan Dolgov, Ekaterina Garanina, Karol Lasocki, Julia Reshetnikova, Sergei Boitsov, Ivan Bondyrev, Dariia Karaeva, Maksim Sheptyakov, Dmitry Ustalov, Artem Mukhin, Semyon Proshev, Nikita Abramov, Olga Kolomyttseva, Kseniia Lysaniuk, Ilia Zavidnyi, Anton Semenkin, Vladislav Tankov, and Uladzislau Sazanovich. Mellum-4b-base, 2025.

[195] Elie Bakouch, Loubna Ben Allal, Anton Lozhkov, Nouamane Tazi, Lewis Tunstall, Carlos Miguel Patiño, Edward Beeching, Aymeric Roucher, Aksel Joonas Reedi, Quentin Gallouédec, Kashif Rasul, Nathan Habib, Clémentine Fourrier, Hynek Kydlicek, Guilherme Penedo, Hugo Larcher, Mathieu Morlon, Vaibhav Srivastav, Joshua Lochner, Xuan-Son Nguyen, Colin Raffel, Leandro von Werra, and Thomas Wolf. SmolLM3: smol, multilingual, long-context reasoner. `https://huggingface.co/blog/smollm3`, 2025.

[196] Marc Marone and Benjamin Van Durme. Data portraits: Recording foundation model training data. *Advances in Neural Information Processing Systems*, 36:15121–15135, 2023.

[197] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[198] Ola Shorinwa, Zhiting Mei, Justin Lidard, Allen Z. Ren, and Anirudha Majumdar. A Survey on Uncertainty Quantification of Large Language Models: Taxonomy, Open Research Challenges, and Future Directions. *ACM Comput. Surv.*, June 2025. ISSN 0360-0300. doi: 10.1145/3744238.

[199] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Prem Devanbu, and Toufique Ahmed. Calibration and correctness of language models for code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 540–552. IEEE, 2025.

[200] Jinwei Xu, He Zhang, Yanjing Yang, Lanxin Yang, Zeru Cheng, Jun Lyu, Bohan Liu, Xin Zhou, Alberto Bacchelli, Yin Kia Chiam, et al. One size does not fit all: Investigating efficacy of perplexity in detecting llm-generated code. *ACM Transactions on Software Engineering and Methodology*, 2024.

[201] Maciej Besta, Julia Barth, Eric Schreiber, Ales Kubicek, Afonso Catarino, Robert Gerstenberger, Piotr Nyczyk, Patrick Iff, Yueling Li, Sam Houliston, Tomasz Sternal, Marcin Copik, Grzegorz Kwaśniewski, Jürgen Müller, Łukasz Flis, Hannes Eberhard, Hubert Niewiadomski, and Torsten Hoefler. Reasoning Language Models: A Blueprint, January 2025.

[202] Kun Wang, Guibin Zhang, Zhenhong Zhou, Jiahao Wu, Miao Yu, Shiqian Zhao, Chenlong Yin, Jinhu Fu, Yibo Yan, Hanjun Luo, et al. A comprehensive survey in llm (-agent) full stack safety: Data, training and deployment. *arXiv preprint arXiv:2504.15585*, 2025.

[203] Cedric Richter and Heike Wehrheim. Tssb-3m: Mining single statement bugs at massive scale. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 418–422, 2022.

[204] Jamie Hayes, Marika Swanberg, Harsh Chaudhari, Itay Yona, Ilia Shumailov, Milad Nasr, Christopher A Choquette-Choo, Katherine Lee, and A Feder Cooper. Measuring memorization in language models via probabilistic extraction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 9266–9291, 2025.

[205] Arthur V Kamienski, Luisa Palechor, Cor-Paul Bezemer, and Abram Hindle. Pysstubs: Characterizing single-statement bugs in popular open-source python projects. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 520–524. IEEE, 2021.

[206] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, 2019.

[207] Yiling He, Junchi Lei, Hongyu She, Shuo Shao, Xinran Zheng, Yiping Liu, Zhan Qin, and Lorenzo Cavallaro. Retrofit: Continual learning with bounded forgetting for security applications. *arXiv preprint arXiv:2511.11439*, 2025.

[208] Haolang Lu, Hongrui Peng, Guoshun Nan, Jiaoyang Cui, Cheng Wang, Weifei Jin, Songtao Wang, Shengli Pan, and Xiaofeng Tao. Malsight: Exploring malicious

source code and benign pseudocode for iterative binary malware summarization. *IEEE Transactions on Information Forensics and Security*, 2025.

[209] Kangchen Zhu, Zhiliang Tian, Shangwen Wang, Weiguo Chen, Zixuan Dong, Mingyue Leng, and Xiaoguang Mao. Misum: multi-modality heterogeneous code graph learning for multi-intent binary code summarization. *Proceedings of the ACM on Software Engineering*, 2(FSE):1339–1362, 2025.

[210] Jiaqi Xiong, Guoqiang Chen, Kejiang Chen, Han Gao, Shaoyin Cheng, and Weiming Zhang. Hext5: Unified pre-training for stripped binary code information inference. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 774–786. IEEE, 2023.

[211] Xiuwei Shang, Zhenkan Fu, Shaoyin Cheng, Guoqiang Chen, Gangyang Li, Li Hu, Weiming Zhang, and Nenghai Yu. An empirical study on the effectiveness of large language models for binary code understanding. *Empirical Software Engineering*, 31 (1):1–38, 2026.

[212] Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601*, 2023.

[213] Xiuwei Shang, Guoqiang Chen, Shaoyin Cheng, Shikai Guo, Yanming Zhang, Weiming Zhang, and Nenghai Yu. Foc: Figure out the cryptographic functions in stripped binaries with llms. *ACM Transactions on Software Engineering and Methodology*, 35(1):1–38, 2025.

[214] Zhaoyang Chu, Yao Wan, Zhikun Zhang, Di Wang, Zhou Yang, Hongyu Zhang, Pan Zhou, Xuanhua Shi, Hai Jin, and David Lo. Scrub it out! erasing sensitive memorization in code language models via machine unlearning. *arXiv preprint arXiv:2509.13755*, 2025.

[215] Zhi Chen and Lingxiao Jiang. Promise and peril of collaborative code generation models: Balancing effectiveness and memorization. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 493–505, 2024.

[216] Yuan Jiang, Zehao Li, Shan Huang, Christoph Treude, Xiaohong Su, and Tiantian Wang. Effective code membership inference for code completion models via adversarial prompts. *arXiv preprint arXiv:2511.15107*, 2025.

Curriculum Vitæ

Ali Al-Kaswan

29/01/1999 Born in Lelystad, The Netherlands

Education

9/22–9/26 Ph.D. Candidate, Software Engineering Research Group,
Delft University of Technology, The Netherlands,
Promotor: Prof. Dr. Arie van Deursen,
Supervisor: Dr. Maliheh Izadi.

8/20–8/22 M.Sc. Computer Science and Engineering,
Delft University of Technology, The Netherlands,
Thesis: *Empirical Assessment on the Limits of Binary Code Summarization with Transformer Models*.

8/17–8/20 B.Sc. Computer Science and Engineering,
Delft University of Technology, The Netherlands,
Thesis: *Educational Game about Datacenters for Kids*.

Work Experience

9/26–present Postdoctoral Researcher,
Delft University of Technology, The Netherlands,
Supervisor: Prof. Dr. Andy Zaidman.

4/22–7/25 Teaching Assistant,
Delft University of Technology, The Netherlands,
Course: *Analytics and Machine Learning for Software Engineering*.

11/21–9/22 Junior Incident Responder,
Eye Security, The Hague, The Netherlands.

7/20–8/22 MOOC Moderator, *Elements of AI*,
Delft University of Technology, The Netherlands.

6/19–12/23 Teaching Assistant,
Delft University of Technology, The Netherlands,

Courses: *Software Architecture*, *Concepts of Programming Languages*, *Software Engineering Methods*, and *Computational Intelligence*.

6/19–3/20 Student Mentor,
Delft University of Technology, The Netherlands.

List of Publications

The More You Tune, The More They Spill? An Empirical Study of Data Leakage from Code Models Under Fine-tuning, Quantization, and Parameter-Efficient Adaptation (EMSE'26). Fabio Salerno, **Ali Al-Kaswan**, Maliheh Izadi. In *Empirical Software Engineering*, 2026.

Do Agents Dream of Root Shells? Partial-Credit Evaluation of LLM Agents in Capture The Flag Challenges (AIWare'26). **Ali Al-Kaswan**, Maksim Plotnikov, Maxim Hájek, Roland Vízner, Arie van Deursen, Maliheh Izadi. In *AIWare 2026*.

AST-PAC: AST-guided Membership Inference for Code (FSE-IVR'26). Roham Koohestani, **Ali Al-Kaswan**, Jonathan Katzy, Maliheh Izadi. In *FSE International Workshop on Intelligent and Verifiable Repository Mining 2026*.

Model See, Model Do? Exposure-Aware Evaluation of Bug-vs-Fix Preference in Code LLMs (MSR'26). **Ali Al-Kaswan**, Claudio Spiess, Prem Devanbu, Arie van Deursen, Maliheh Izadi. In *2026 IEEE/ACM International Conference on Mining Software Repositories (MSR)*.

The Poisoned Chalice of LLM Evaluation Report (FSE Companion'26). Jonathan Katzy, **Ali Al-Kaswan**, Razvan Mihai Popescu, Zhou Yang. In *Companion Proceedings of the 34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26)*.

Code Red! On the Harmfulness of Applying Off-the-shelf Large Language Models to Programming Tasks (FSE'25). **Ali Al-Kaswan**, Sebastian Deatc, Begüm Koç, Arie van Deursen, Maliheh Izadi. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE 2025)*.

The NLBSE'25 Tool Competition. **Ali Al-Kaswan**, Giuseppe Colavito, Nataliia Stulova, Pooja Rani. In *2025 IEEE/ACM International Workshop on Natural Language-Based Software Engineering (NLBSE)*.

How Much Do Code Language Models Remember? An Investigation on Data Extraction Attacks Before and After Fine-Tuning. Fabio Salerno, **Ali Al-Kaswan**, Maliheh Izadi. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*.

Traces of Memorisation in Large Language Models for Code (ICSE'24). **Ali Al-Kaswan**, Maliheh Izadi, Arie van Deursen. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE 2024)*.

Towards Safe, Secure, and Usable LLMs4Code (ICSE DocSym'24). **Ali Al-Kaswan**. In *Doctoral Symposium of the IEEE/ACM International Conference on Software Engineering (ICSE Doctoral Symposium 2024)*.

The NLBSE'24 Tool Competition. Rafael Kallis, Giuseppe Colavito, **Ali Al-Kaswan**, Luca Pascarella, Oscar Chaparro, Pooja Rani. In *Proceedings of the Third ACM/IEEE International Workshop on NL-based Software Engineering*.

The (ab)use of Open Source Code to Train Large Language Models (NLBSE'23). **Ali Al-Kaswan**, Maliheh Izadi, Arie van Deursen. In *Proceedings of the ACM/IEEE International Workshop on Natural Language-Based Software Engineering (NLBSE 2023)*.

🏆 **STACC: Code Comment Classification using SentenceTransformers** (NLBSE'23). **Ali Al-Kaswan**, Maliheh Izadi, Arie van Deursen. In *Proceedings of the ACM/IEEE International Workshop on Natural Language-Based Software Engineering (NLBSE 2023)*.

🏆 **Targeted Attack on GPT-Neo for the SATML Language Model Data Extraction Challenge** (2023). **Ali Al-Kaswan**, Maliheh Izadi, Arie van Deursen.

🗎 **Extending Source Code Pre-Trained Language Models to Summarise Decompiled Binaries** (SANER'23). **Ali Al-Kaswan**, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, Arie van Deursen. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

🗎 Included in this thesis.
🏆 Won a best paper, tool demonstration, or proposal award.

Titles in the IPA Dissertation Series since 2023

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems*. Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN*. Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification*. Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification*. Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software*. Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis*. Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond*. Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic*. Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction*. Faculty of Science, Mathematics and Computer Science, RU. 2024-07

O. Bunte. *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software*. Faculty of Mathematics and Computer Science, TU/e. 2024-08

R.J.A. Erkens. *Automaton-based Techniques for Optimized Term Rewriting*. Faculty of Mathematics and Computer Science, TU/e. 2024-09

J.J.M. Martens. *The Complexity of Bisimilarity by Partition Refinement*. Faculty of Mathematics and Computer Science, TU/e. 2024-10

L.J. Edixhoven. *Expressive Specification and Verification of Choreographies*. Faculty of Science, OU. 2024-11

J.W.N. Paulus. *On the Expressivity of Typed Concurrent Calculi*. Faculty of Science and Engineering, RUG. 2024-12

J. Denkers. *Domain-Specific Languages for Digital Printing Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-13

L.H. Applis. *Tool-Driven Quality Assurance for Functional Programming and Machine Learning*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-14

P. Karkhanis. *Driving the Future: Facilitating C-ITS Service Deployment for Connected and Smart Roadways*. Faculty of Mathematics and Computer Science, TU/e. 2024-15

N.W. Cassee. *Sentiment in Software Engineering*. Faculty of Mathematics and Computer Science, TU/e. 2024-16

H. van Antwerpen. *Declarative Name Binding for Type System Specifications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-01

I.N. Mulder. *Proof Automation for Fine-Grained Concurrent Separation Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2025-02

T.S. Badings. *Robust Verification of Stochastic Systems: Guarantees in the Presence of Uncertainty*. Faculty of Science, Mathematics and Computer Science, RU. 2025-03

A.M. Mir. *Machine Learning-assisted Software Analysis*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-04

L.T. Vinkhuijzen. *Data Structures for Quantum Circuit Verification and How To Compare Them*. Faculty of Mathematics and Natural Sciences, UL. 2025-05

D. van der Wal. *What is the Point? Single-Input-Change Testing a EULYNX Controller*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-06

A. Rosset. *Uniform Monad Presentations and Graph Quasitoposes*. Faculty of Sciences, Department of Computer Science, VU. 2025-07

L. Guo. *Higher-Order Termination with Logical Constraints*. Faculty of Science, Mathematics and Computer Science, RU. 2025-08

D.A. Manrique Negrin. *A Model Orchestra in Digital Twins: A Model-Driven Approach to Integration and Orchestration*. Faculty of Mathematics and Computer Science, TU/e. 2025-09

C.A. Esterhuyse. *Specification-Centric Multi-Agent Systems*. Faculty of Science, UvA. 2025-10

H.M. Muctadir. *Consistency Matters: Building Consistent Digital Twin Virtual Entities*. Faculty of Mathematics and Computer Science, TU/e. 2025-11

A. Stramaglia. *Model Checking Machine-Control Applications*. Faculty of Mathematics and Computer Science, TU/e. 2025-12

M. Saeedi Nikoo. *Supporting business process management: clone detection and recommendation techniques*. Faculty of Mathematics and Computer Science, TU/e. 2025-13

R.B. Rubbens. *Bridging the Implementation Gap: Advances in Model-Based Concurrent Program Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-14

F.I. van der Berg. *DMC Model Checker: Delta-Driven Variable-Length Next-State Generation via Recursive Compression*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-15

D.G. Sprokholt. *Correct Translation between Weak Memory Model Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-16

B. Kohlen. *Verified & Fast Probabilistic Verification: Stepwise Refinement of Probabilistic Model Checking into an IEEE 754 Floating-Point Implementation*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2026-01

D. Frolich. *Exploratory Language Development*. Faculty of Science, UvA. 2026-02

L.B. van den Haak. *Verifying Optimised Parallel Code*. Faculty of Mathematics and Computer Science, TU/e. 2026-03

A. Papotti. *Human Acceptance of Machine Generated Code Security Analysis*. Faculty of Sciences, Department of Computer Science, VU. 2026-04

N. Doorn. *O bug, where art thou? Improving software testing education*. Faculty of Science, OU. 2026-05

C.R. van der Rest. *Reusable Programming Language Components*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2026-06

T.T.P. Franken. *A Parallel Programming Perspective – Designing AuDaLa and AuDaLogic*. Faculty of Mathematics and Computer Science, TU/e. 2026-07

T.J. Smeding. *Reverse-mode Automatic Differentiation for Functional Array Languages*. Faculty of Science, UU. 2026-08

S.A. Rukmono. *Towards Automated Software Architecture Explanation: An AI-enabled method and industrial evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2026-09

R.T.C. Turkenburg. *Comparing and Constructing (Co)Inductive Predicates*. Faculty of Science, Mathematics and Computer Science, RU. 2026-10

P.B.H. Tasche. *Deductive Verification Techniques for Embedded Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2026-11

W.B. Mbaka. *An Empirical Investigation Into Human Judgments of Security Threats*. Faculty of Sciences, Department of Computer Science, VU. 2026-12

S. Srinivasan. *Taming Uncertainty, Enabling Predictability: Systematic Timing Analysis of Real-Time Systems Across Computation and Communication*. Faculty of Mathematics and Computer Science, TU/e. 2026-13

S.O. Brand. *Symbolic Algorithms for the Analysis of Quantitative Systems*. Faculty of Mathematics and Natural Sciences, UL. 2026-14

D. Engel. *Understanding Binaries, Correctly: Decidability and Formal Verification in Binary Analysis*. Faculty of Science, OU. 2026-15

K. Hagens. *Rewriting Induction for Higher-Order Constrained Rewriting*. Faculty of Science, Mathematics and Computer Science, RU. 2026-16

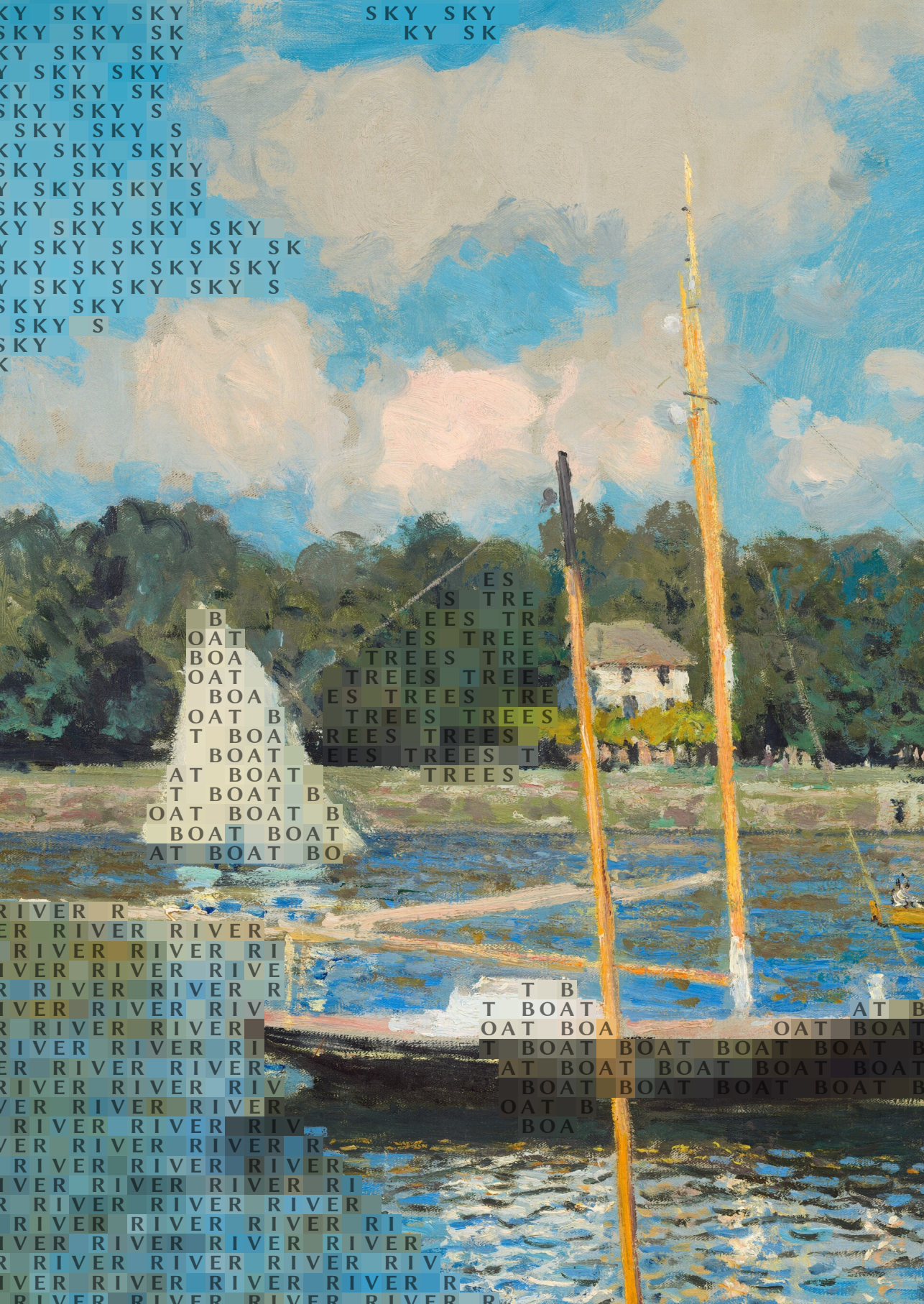
S. Khatami. *Understanding Software Quality Assurance in Open-Source Communities: Awareness, Adoption, and Tool Uti-*

lization. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2026-17

R. Soltani. *Defense Against the Dark Arts: Formal Semantics and Quantitative Analysis of Attacks, Failures, and Defenses*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2026-18

D.A.A. Pelsmaeker. *Language-Parametric Editor Services from Language Specifications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2026-19

A. Al-Kaswan. *A Dual Perspective on Large Language Models for Code and Security*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2026-20


SKY SKY
KY SK
ES
S TRE
EES TR
ES TREE
TREES TRE
TREES TREE
ES TREES TRE
TREES TREES
REES TREES
EES TREES T
TREES
B
OAT
BOA
OAT
BOA
OAT B
T BOA
BOAT
AT BOAT
T BOAT B
OAT BOAT B
BOAT BOAT
AT BOAT BO
RIVER R
ER RIVER RIVER
RIVER RIVER RI
IVER RIVER RIVE
R RIVER RIVER R
IVER RIVER RIV
R RIVER RIVER
RIVER RIVER RI
ER RIVER RIVER
RIVER RIVER RIV
VER RIVER RIVER
RIVER RIVER RIV
VER RIVER RIVER R
RIVER RIVER RIVER
IVER RIVER RIVER RI
R RIVER RIVER RIVER
RIVER RIVER RIVER RI
IVER RIVER RIVER RIVER
R RIVER RIVER RIVER RIV
IVER RIVER RIVER RIVER R
T B
T BOAT
OAT BOA
T BOAT BOAT BOAT BOAT
AT BOAT BOAT BOAT BOAT
BOAT BOAT BOAT BOAT
OAT B
BOA
AT
OAT BOAT
BOAT BOAT BOAT