TECHNICAL UNIVERSITY OF CRETE

MASTER OF SCIENCE THESIS

White-Basilisk: A Hybrid Model for Code Vulnerability Detection

---

Author:
Ioannis LAMPROU

Thesis Committee:
Prof. Sotiris IOANNIDIS
Prof. Apostolos DOLLAS
Prof. Georgios CHALKIADAKIS

A thesis submitted in fulfillment of the requirements for the diploma of Master of Science in Electrical and Computer Engineering

in the

School of Electrical and Computer Engineering
Parasecurity Group

August 29, 2025

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Master of Science in Electrical and Computer Engineering

White-Basilisk: A Hybrid Model for Code Vulnerability Detection

by Ioannis LAMPROU

The proliferation of software vulnerabilities presents a significant challenge to cybersecurity, necessitating more effective detection methodologies. We introduce White-Basilisk, a novel approach to vulnerability detection that demonstrates superior performance while challenging prevailing assumptions in AI model scaling. Utilizing an innovative architecture that integrates Mamba layers, linear self-attention, and a Mixture of Experts framework, White-Basilisk achieves state-of-the-art results in vulnerability detection tasks with a parameter count of only 200M. The model's capacity to process sequences of unprecedented length enables comprehensive analysis of extensive codebases in a single pass, surpassing the context limitations of current Large Language Models (LLMs). White-Basilisk exhibits robust performance on imbalanced, real-world datasets, while maintaining computational efficiency that facilitates deployment across diverse organizational scales. This research not only establishes new benchmarks in code security but also provides empirical evidence that compact, efficiently designed models can outperform larger counterparts in specialized tasks, potentially redefining optimization strategies in AI development for domain-specific applications.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Master of Science in Electrical and Computer Engineering

White-Basilisk: A Hybrid Model for Code Vulnerability Detection

by Ioannis LAMPROU

Η διάδοση των ευπαθειών λογισμικού αποτελεί σημαντική πρόκληση για την κυβερνοασφάλεια, καθιστώντας αναγκαία την ανάπτυξη αποτελεσματικότερων μεθοδολογιών ανίχνευσης. Παρουσιάζουμε το White-Basilisk, μια καινοτόμο προσέγγιση στην ανίχνευση ευπαθειών που επιδεικνύει ανώτερη απόδοση, αμφισβητώντας παράλληλα τις επικρατούσες υποθέσεις στην κλιμάκωση μοντέλων AI. Χρησιμοποιώντας μια καινοτόμο αρχιτεκτονική που ενσωματώνει Mamba layers, linear self-attention, και ένα πλαίσιο Mixture of Experts, το White-Basilisk επιτυγχάνει κορυφαία αποτελέσματα σε εργασίες ανίχνευσης ευπαθειών με μόλις 200M παραμέτρους. Η ικανότητα του μοντέλου να επεξεργάζεται ακολουθίες πρωτοφανούς μήκους επιτρέπει την ολοκληρωμένη ανάλυση εκτεταμένων βάσεων κώδικα σε ένα πέρασμα, ξεπερνώντας τους περιορισμούς πλαισίου των τρεχόντων Large Language Models (LLMs). Το White-Basilisk επιδεικνύει ισχυρή απόδοση σε μη ισορροπημένα σύνολα δεδομένων του πραγματικού κόσμου, διατηρώντας παράλληλα υπολογιστική αποδοτικότητα που διευκολύνει την ανάπτυξη σε διάφορες οργανωτικές κλίμακες. Αυτή η έρευνα όχι μόνο θέτει νέα σημεία αναφοράς στην ασφάλεια κώδικα, αλλά παρέχει επίσης εμπειρικά στοιχεία ότι τα συμπαγή, αποδοτικά σχεδιασμένα μοντέλα μπορούν να ξεπεράσουν μεγαλύτερα αντίστοιχά τους σε εξειδικευμένες εργασίες, επαναπροσδιορίζοντας πιθανώς τις στρατηγικές βελτιστοποίησης στην ανάπτυξη AI για εφαρμογές συγκεκριμένου τομέα.

Acknowledgements

I would like to express my deepest gratitude to my research supervisor, Professor Sotiris Ioannidis, for his invaluable guidance, unwavering support, and insightful feedback throughout this research journey. His expertise and vision have been instrumental in shaping not only this thesis but also my approach to research in cybersecurity. I am also profoundly grateful to the members of my thesis committee, Professor Apostolos Dollas and Professor Georgios Chalkiadakis, for their constructive criticism, thoughtful recommendations, and the time they dedicated to reviewing my work. Their diverse perspectives and expertise have significantly enhanced the quality of this research. Special thanks to Dr. Alexander Shevtsov, whose mentorship and technical insights proved invaluable during the development of the White-Basilisk model. His willingness to share his knowledge and experience helped me overcome numerous challenges throughout this project. I would also like to acknowledge the Parasecurity Group at the Technical University of Crete for providing an enriching academic environment and the resources necessary for conducting this research. The collaborative spirit and intellectual rigor of the group have been a constant source of inspiration. My appreciation extends to my fellow researchers and friends who offered encouragement, engaged in stimulating discussions, and provided moral support during the more challenging phases of this project. Finally, I am eternally grateful to my family for their unconditional love, understanding, and support throughout my academic pursuits. Their belief in me has been my strongest motivation. This thesis would not have been possible without the contribution of everyone mentioned above.

Contents

List of Figures

List of Tables

Dedicated to my family and friends…

Chapter 1

Introduction

The field of Artificial Intelligence (AI) has experienced significant advancements in recent years, particularly in the domain of Natural Language Processing (NLP). Large Language Models (LLMs) such as GPT, Llama, and Gemini have demonstrated remarkable capabilities across diverse tasks. These models, often comprising hundreds of billions of parameters, reflect the "bigger is better" philosophy in AI development. However, even with the notable success of these massive models, they present substantial challenges. For instance, the computational requirements for training and inference of such models are considerable, resulting in high energy consumption and limited accessibility [1, 2]. As these models continue to expand in size and complexity, it is important to reconsider the sustainability and necessity of this approach for all AI applications.

One domain where the limitations of the "bigger is better" paradigm become particularly evident is in specialized tasks like code vulnerability detection. This critical aspect of cybersecurity requires models that can comprehend complex code structures, identify subtle patterns, and maintain high accuracy – all while ideally being deployable in resource-constrained environments. Code vulnerability detection represents a unique challenge at the intersection of software engineering and cybersecurity. Vulnerabilities often emerge from complex interactions between various components in the codebase, making detection through standard techniques difficult. The rapid evolution of both software development practices and attack methodologies further exacerbates this challenge. As developers embrace novel paradigms such as microservices and AI-driven code generation, the potential attack surface expands in ways that can be challenging to predict.

Traditional approaches to automated vulnerability detection, including static application security testing (SAST) tools, have demonstrated both strengths

and limitations in addressing security challenges. Studies have shown that SAST tools generally achieve lower vulnerability detection rates compared to newer approaches like large language models (LLMs) [3]. This lower detection rate is concerning because it means that SAST tools may miss critical security vulnerabilities, potentially leaving software systems exposed to exploitation and attacks. The inability to identify a significant portion of vulnerabilities undermines the effectiveness of these tools in ensuring robust software security, highlighting the need for more advanced or complementary detection methods.

Recent machine learning (ML)-based approaches have demonstrated promising results in vulnerability detection. However, many exhibit significant limitations in processing extended code sequences and comprehending complex, context-dependent vulnerabilities. These models often struggle with long-range dependencies and fail to capture the nuanced interactions between disparate code components. Moreover, their performance can be inconsistent across diverse vulnerability types and programming paradigms, limiting their efficacy in comprehensive security analyses. In response to these challenges, we present White-Basilisk: a compact but powerful model that challenges the prevailing paradigm in AI design. With approximately 200M parameters – a fraction of the size of many current state-of-the-art models – White-Basilisk demonstrates that thoughtful architecture and targeted training can yield impressive results in code vulnerability detection.

To evaluate White-Basilisk's efficacy, we conducted rigorous experiments across multiple benchmark datasets, including PRIMEVUL [4], BigVul [5], Draper [6], REVEAL [7], and VulDeepecker [8]. Our compact model not only demonstrated competitive performance against larger counterparts but emerged as the front-runner in several key metrics.

By addressing the unique challenges of code vulnerability detection with a resource-efficient approach, White-Basilisk not only explores new possibilities in this critical domain but also raises pertinent questions about the necessity of ever-larger models in AI. In the subsequent sections, we will elucidate the details of White-Basilisk's architecture, training procedures, and performance across multiple benchmarks.

1.1 Objectives and Scope

This thesis aims to demonstrate that efficient, specialized models can outperform larger general-purpose models in the domain of code vulnerability detection. The primary objective is to develop and validate a novel hybrid architecture that combines state space models, linear attention mechanisms, and sparse expert systems while maintaining a parameter count under 200 million.

The scope of this research encompasses the analysis of C and C++ codebases, focusing on detecting security vulnerabilities across various complexity levels and contexts. While the principles developed may apply to other programming languages, we specifically target C and C++ due to their prevalence in security-critical systems and the availability of comprehensive vulnerability datasets in these languages.

Our investigation extends to analyzing sequences of up to 128,000 tokens, enabling the processing of entire codebases in a single pass. This capability is essential for identifying vulnerabilities that manifest through interactions between different parts of a program. The research also explores the environmental impact of AI model training and deployment, considering the broader implications of model efficiency beyond mere performance metrics.

The work focuses primarily on static analysis of source code, rather than dynamic analysis or runtime vulnerability detection. We evaluate our approach against established benchmarks in the field, including PRIMEVUL, BigVul, Draper, REVEAL, and VulDeepecker datasets, providing a comprehensive assessment of the model's capabilities across different vulnerability types and contexts.

1.2 Contributions

This thesis advances the field of code vulnerability detection by challenging fundamental assumptions about model scaling and efficiency. The primary contribution is the development of White-Basilisk, a novel hybrid architecture that achieves state-of-the-art performance in vulnerability detection while using significantly fewer parameters than existing approaches.

White-Basilisk's design philosophy centers on maximizing efficiency without compromising performance. This approach led to several innovations that form the core of our contributions, which can be summarized as follows:

1. **Efficient Architecture:** We propose a novel integration of Mamba layers, linear-complexity Infini-attention, and Mixture of Experts. This combination enables effective processing of long code sequences while maintaining a relatively small parameter count.

2. **Extended Context Length:** White-Basilisk can analyze sequences up to 128,000 tokens during inference. This capability facilitates whole-codebase analysis on a single GPU, potentially uncovering vulnerabilities that span multiple functions or files.

3. **Resource-Efficient Training:** Our model achieves competitive performance using a dataset of just 2 million code samples. This efficiency challenges conventional assumptions about data requirements for specialized AI tasks.

4. **Advanced Training Techniques:** We incorporated Fill-in-the-Middle (FIM) pretraining and Scale-Invariant Fine-Tuning (SIFT) to enhance model robustness and generalization. These techniques contribute to White-Basilisk's ability to perform well across diverse code vulnerability detection tasks.

5. **Comprehensive Benchmarking:** We conduct rigorous experiments to evaluate White-Basilisk's efficacy across multiple benchmark datasets, including PRIMEVUL [4], BigVul [5], Draper [6], REVEAL [7], and VulDeepecker [8]. Our compact model demonstrates competitive performance against larger counterparts, emerging as the front-runner in several key metrics.

Chapter 2

Literature Review and Theoretical Background

2.1 Code Vulnerability Detection

The evolution of code vulnerability detection methodologies can be traced through three distinct phases: traditional approaches, machine learning-based methods, and deep learning techniques. Each phase has contributed significantly to our current understanding of automated vulnerability detection, while also revealing important limitations that drove subsequent innovations.

Static Analysis Techniques

Traditional approaches to vulnerability detection primarily relied on static analysis and pattern matching techniques. Early work by Chess and McGraw [9] established fundamental principles for static analysis security testing (SAST), introducing methods for identifying potential security flaws without executing the code. These approaches typically employed techniques such as taint analysis, control flow analysis, and symbolic execution to identify potential vulnerabilities. Livshits and Lam [10] advanced this field by introducing specification-based vulnerability detection, where security patterns were explicitly defined and matched against source code.

While these traditional methods laid crucial groundwork, they faced significant limitations. Cheirdari and Karabatis [11] identified that high false positive rates and scalability issues were major barriers to adoption in real-world development environments.

Dynamic Analysis Techniques

In contrast to static analysis methods that examine code without execution, dynamic analysis techniques investigate program behavior during runtime. These approaches offer complementary strengths for vulnerability detection by observing actual execution paths and data flows that may be difficult to predict through static analysis alone. Dynamic analysis has evolved significantly over the past two decades, with several key methodologies emerging as particularly effective for vulnerability detection.

Fuzzing represents one of the most widely adopted dynamic techniques, as demonstrated by Böhme, Pham, and Roychoudhury [12], who introduced coverage-guided fuzzing that substantially improved vulnerability discovery rates. Modern fuzzing frameworks like American Fuzzy Lop (AFL) and libFuzzer have become essential components in security testing pipelines. These tools generate pseudo-random inputs to trigger unexpected program behaviors, Chen and Chen [13] showing their effectiveness in identifying memory corruption vulnerabilities that often evade static detection.

Taint tracking, another pivotal dynamic approach, monitors the flow of untrusted data through program execution. Schwartz, Avgerinos, and Brumley [14] demonstrated that dynamic taint analysis can precisely identify injection vulnerabilities by tracing how user inputs propagate to security-sensitive operations. This approach proves particularly valuable for detecting vulnerabilities that manifest only under specific runtime conditions.

Dynamic symbolic execution, pioneered by Cadar, Dunbar, and Engler [15] with their KLEE system, systematically explores execution paths by solving path constraints to generate inputs reaching potentially vulnerable code sections. While computationally intensive, this technique provides high precision by verifying the exploitability of detected vulnerabilities, as shown by Stephens et al. [16] in their Driller framework that combines fuzzing with symbolic execution.

Runtime instrumentation frameworks such as Valgrind and AddressSanitizer enable detailed memory access monitoring, with Serebryany et al. [17] demonstrating their effectiveness in catching memory safety issues. These tools introduce runtime checks that identify violations of memory safety properties during program execution, providing precise vulnerability information with minimal false positives.

Despite their strengths, dynamic techniques face significant challenges. They typically achieve lower code coverage compared to static approaches, as noted by Rawat et al. [18], who showed that even advanced fuzzing techniques often explore less than 60% of complex codebases. Additionally, dynamic analysis incurs substantial runtime overhead, limiting its applicability in performance-sensitive environments. The effectiveness of these techniques also depends heavily on the quality and diversity of test inputs, creating a fundamental challenge in ensuring comprehensive vulnerability detection.

Modern approaches increasingly employ hybrid methodologies that combine static and dynamic techniques to leverage their complementary strengths. The integration of machine learning with dynamic analysis has further enhanced vulnerability detection capabilities, guiding fuzzing toward potentially vulnerable program regions.

Machine Learning Approaches

The transition to machine learning-based approaches marked a significant advancement in vulnerability detection. Li et al. [19] pioneered the use of clustering algorithms to identify recurring vulnerability patterns, demonstrating that machine learning could effectively capture patterns that were difficult to specify manually. This work was extended by Grieco et al. [20], who introduced feature engineering techniques specifically designed for vulnerability detection.

A significant breakthrough came with Li et al. [8]'s VulDeePecker, which introduced the concept of code gadgets – semantically related code fragments that could be analyzed as a unit. This approach demonstrated superior performance compared to traditional methods, achieving higher detection rates while reducing false positives. However, these approaches still relied heavily on manual feature engineering, limiting their ability to adapt to new vulnerability patterns.

Deep Learning Approaches

The advent of deep learning techniques has revolutionized vulnerability detection. Russell et al. [6] demonstrated the effectiveness of neural networks in analyzing raw code, eliminating the need for manual feature engineering. This work was further advanced by Zhou et al. [21], who introduced Devign,

a graph neural network approach that could capture both syntactic and semantic features of code.

Recent developments have focused on leveraging large language models (LLMs) for vulnerability detection. Hanif and Maffeis [22] introduced VulBERTa, adapting the RoBERTa architecture for code analysis. This was followed by more specialized approaches like Fu and Tantithamthavorn [23]'s LineVul, which achieved precise vulnerability localization at the line level. These models demonstrated unprecedented accuracy in detecting complex vulnerabilities, though often at the cost of significant computational resources.

However, even these advanced approaches face challenges. Chakraborty et al. [7]'s comprehensive study revealed persistent issues with dataset quality and evaluation methodologies. More recently, Ding et al. [4] demonstrated that many current benchmarks significantly overestimate model performance, particularly when evaluating on real-world codebases.

The limitations of existing approaches, particularly in terms of computational efficiency and practical deployability, have motivated the development of more efficient architectures. Recent work by Gu and Dao [24] on state space models and Wang et al. [25] on linear attention mechanisms has opened new possibilities for efficient, scalable vulnerability detection. These developments suggest that the field is moving toward more resource-efficient solutions that maintain or exceed the performance of larger models while reducing computational requirements.

This evolution of vulnerability detection approaches reflects a broader trend in artificial intelligence: the search for more efficient, practical solutions that can be readily deployed in real-world environments. The following sections will explore in detail the specific technological advances that enable these improvements, particularly in the context of large language models, efficiency in AI models, and the integration of various architectural innovations.

2.1.1 Large Language Models in Code Analysis

Large Language Models (LLMs) have emerged as powerful tools for code analysis, introducing new capabilities while also presenting unique challenges in the context of software engineering. The application of LLMs to code analysis represents a significant shift from traditional program analysis techniques, offering both opportunities and limitations that merit careful examination.

Evolution of Code-Specific LLMs

The development of code-specific LLMs began with models like CodeBERT [26], which demonstrated the effectiveness of pre-training on both programming languages and natural language descriptions. This bimodal pre-training approach enabled the model to develop a deeper understanding of code semantics and their relationship to natural language descriptions. Building on this foundation, Guo et al. [27] introduced UniXcoder, which enhanced code understanding through a unified cross-modal framework that could handle multiple programming languages simultaneously.

A significant advancement came with the introduction of CodeX [28], which demonstrated unprecedented capabilities in code generation and analysis. This was followed by more specialized models like StarCoder [29], which was trained specifically on The Stack, a massive dataset of permissively licensed source code. These models showed remarkable ability to understand code context and generate appropriate completions, though their size and computational requirements presented significant deployment challenges.

Architectural Innovations for Code Processing

The unique characteristics of source code necessitated specific architectural innovations in LLMs. Ahmad et al. [30] introduced tree-based attention mechanisms that could better capture the hierarchical structure of code, while Ding et al. [4] developed techniques for handling long-range dependencies specific to code analysis. These innovations addressed fundamental limitations of traditional transformer architectures when applied to code. Unlike natural language, code has explicit structural elements that need to be preserved and understood. Additionally, code analysis often requires understanding dependencies across much longer sequences than typical natural language processing tasks. Furthermore, small changes in code can have dramatic effects on program behavior, requiring models to be highly sensitive to subtle differences in semantic precision.

Applications in Security Analysis

The application of LLMs to security analysis has shown particular promise. Zhou et al. [3] demonstrated that LLMs could identify subtle security vulnerabilities that traditional static analyzers often missed. However, found that

while LLMs showed superior performance in detecting complex vulnerabilities, they also exhibited higher false positive rates compared to traditional tools in certain scenarios. Recent work by Chakraborty et al. [7] highlighted several key challenges in applying LLMs to security analysis. The decision-making process of LLMs remains largely opaque, making it difficult to validate their security assessments. State-of-the-art models often require significant computational resources, limiting their practical deployment. Models can also struggle with code patterns that differ significantly from their training data.

Efficiency Considerations

A critical challenge in applying LLMs to code analysis has been the trade-off between model capability and computational efficiency. Patterson et al. [1] demonstrated that the energy consumption of large code analysis models can be substantial, raising questions about their environmental impact and practical deployability. This has led to research into more efficient architectures, such as the work by Lieber et al. [31] on combining efficient attention mechanisms with state space models.

The trend toward more efficient models has been particularly relevant for code analysis, where the ability to process long sequences and maintain context is crucial. Recent work by Gu and Dao [24] on selective state space sequences has shown promise in reducing computational requirements while maintaining or improving performance on code analysis tasks. These developments suggest a shift toward more focused, efficient architectures specifically designed for code analysis.

Future Directions

The future of LLMs in code analysis appears to be moving toward more specialized, efficient architectures that can maintain the benefits of large models while reducing computational overhead. Recent work by Munkhdalai, Faruqui, and Gopal [32] on infinite context transformers suggests possibilities for handling extremely long code sequences efficiently, while developments in mixture-of-experts architectures [33] offer potential paths to more scalable solutions.

These trends indicate a growing recognition that while LLMs have transformed code analysis, their future development must prioritize efficiency

and practicality alongside raw performance. The challenge lies in maintaining the sophisticated understanding of code demonstrated by large models while making them more accessible and deployable in real-world development environments.

2.1.2 Transformer Architecture

The Transformer architecture [34] has become foundational in modern natural language processing and code analysis. Initially designed for machine translation, its self-attention mechanism and parallel processing capabilities have made it particularly effective for understanding structured sequences like source code.

Core Architecture

The Transformer consists of several key components working in concert:

$$\text{TransformerLayer}(x) = \text{FFN}(\text{LayerNorm}(\text{MultiHead}(x) + x)) \tag{2.1}$$

where FFN represents a position-wise feed-forward network, and MultiHead represents multi-head attention:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \tag{2.2}$$

Each attention head computes:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.3}$$

Positional Encoding

Unlike recurrent architectures, Transformers have no inherent notion of sequence order. Position information is injected through positional encodings:

$$\text{PE}_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}}) \tag{2.4}$$

$$\text{PE}_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}}) \tag{2.5}$$

Computational Challenges

The standard Transformer architecture faces several significant limitations in processing long sequences. The most pressing challenge is its quadratic memory complexity with sequence length, expressed as $O(L^2 \cdot d)$, where L is the sequence length and d is the hidden dimension. This quadratic scaling makes processing long sequences computationally expensive and memory-intensive. Furthermore, the fixed context window size limits the model's ability to capture long-range dependencies, a particularly important consideration for code analysis where relationships can span hundreds or thousands of lines.

Modern Variants

Recent developments have introduced several important improvements to the basic Transformer architecture. Sparse Transformers reduce attention complexity through structured sparsity patterns, allowing for more efficient processing of long sequences. The sparsity mask defines which attention connections to maintain, significantly reducing computational requirements while preserving most of the model's capacity to capture relevant relationships.

Modern implementations also combine local and global attention mechanisms, allowing models to efficiently process both short-range and long-range dependencies. This approach has proven particularly effective for code analysis, where understanding both local context and global structure is crucial. Memory-efficient implementations have further advanced the practical applicability of Transformers through techniques such as gradient checkpointing and mixed precision training.

Impact on Code Analysis

Transformers have fundamentally changed the landscape of code analysis by enabling more sophisticated understanding of code structure and dependencies. Their ability to process multiple programming languages effectively has led to more general and powerful code analysis tools. The architecture's capacity to capture long-range relationships has proven particularly valuable for security analysis, where vulnerabilities often depend on interactions between distant parts of the codebase.

The ongoing evolution of Transformer architectures continues to focus on reducing computational requirements while maintaining or improving performance on code analysis tasks. These developments suggest that future improvements will likely come from architectural innovations that better balance efficiency and effectiveness, particularly for the specialized demands of code analysis and vulnerability detection.

2.1.3 Attention Mechanisms

Attention mechanisms have been fundamental to the success of modern language models, though their computational complexity presents significant challenges for processing long sequences. Recent innovations in attention design have focused on achieving linear complexity while maintaining effectiveness.

Traditional Self-Attention

The standard self-attention mechanism [34] computes attention scores through scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.6}$$

where Q, K, and V represent query, key, and value matrices respectively, and d_k is the dimension of the key vectors. While effective, this formulation has quadratic complexity in both time and memory with respect to sequence length:

$$\text{Complexity}_{\text{standard}} = O(L^2 \cdot d) \tag{2.7}$$

where L is the sequence length and d is the hidden dimension.

Linear Attention Variants

To address the quadratic complexity issue, several linear attention variants have been proposed. Wang et al. [25] introduced low-rank approximations of the attention matrix:

$$\text{Attention}_{\text{linear}}(Q, K, V) = \phi(Q)(\phi(K)^T V) \tag{2.8}$$

where $\phi(\cdot)$ represents a linear projection that reduces the sequence dimension. This approach achieves linear complexity:

$$\text{Complexity}_{\text{linear}} = O(L \cdot d) \tag{2.9}$$

Long-Range Attention

For code analysis, handling long-range dependencies is crucial. Dai et al. [35] proposed the Transformer-XL architecture, which introduces relative positional encodings and segment-level recurrence. This allows the model to capture dependencies beyond its immediate context window while maintaining computational efficiency.

Recent work by Munkhdalai, Faruqui, and Gopal [32] on Infini-attention has shown promise in processing extremely long sequences through a novel decomposition of the attention computation:

$$A_{\text{mem}} = \frac{(\text{ELU}(Q) + 1)M^T}{(\text{ELU}(Q) + 1)z^T + \epsilon} \tag{2.10}$$

where M represents a compressive memory and z is a normalization term.

Memory-Efficient Implementations

Practical implementations of attention mechanisms have benefited from optimization techniques. Dao et al. [36] introduced Flash Attention, which optimizes memory access patterns. Their approach incorporates tiled matrix multiplication for better cache utilization, in-place softmax computation, and fused kernel implementations for common operations. These optimizations have made attention mechanisms more practical for deployment while maintaining their effectiveness.

Sparse Attention Patterns

An alternative approach to reducing attention complexity involves using sparse attention patterns. Notable variants include block-sparse attention, which attends only to local neighborhoods; strided attention, which attends to tokens at regular intervals; and dynamic sparse attention, which learns which connections to maintain. These patterns can reduce complexity while preserving most of the benefits of full attention for many tasks.

2.1.4 State Space Models and Mamba

State Space Models (SSMs) have emerged as a promising alternative to attention mechanisms in sequence modeling, with Mamba [24] representing a significant advancement in this domain through its selective state space mechanism.

Foundations of State Space Models

State Space Models in deep learning emerged from classical control theory and signal processing. The continuous-time state space model is defined by:

$$\begin{aligned} \frac{d}{dt}h(t) &= Ah(t) + Bx(t) \\ y(t) &= Ch(t) + Dx(t) \end{aligned} \tag{2.11}$$

where $h(t)$ represents the hidden state, $x(t)$ the input, and $y(t)$ the output. The matrices A, B, C, and D define the system's dynamics.

Mamba Architecture

Mamba [24] introduced several key innovations in the application of state space models to sequence processing:

1. **Selective State Spaces:** Instead of fixed state matrices, Mamba implements input-dependent state spaces where the state matrices are functions of the input:
$$A(x) = \text{diag}(\Delta(x))A_{\text{fixed}} \tag{2.12}$$
where $\Delta(x)$ represents input-dependent selective factors.

2. **Linear Complexity:** The architecture achieves linear time and memory complexity with respect to sequence length:
$$\text{Complexity} = O(L \cdot d) \tag{2.13}$$
where L is the sequence length and d is the state dimension.

3. **Efficient Parameterization:** The model employs a structured state space that reduces parameter count while maintaining expressiveness.

Performance Characteristics

The efficiency of Mamba makes it particularly relevant for processing long sequences. Key characteristics include linear memory scaling with sequence length, efficient hardware utilization, and the ability to process longer sequences compared to attention-based architectures.

These characteristics are expressed through the memory complexity:

$$\text{Memory}_{\text{Mamba}} = O(L) \tag{2.14}$$

Recent Developments

While Mamba itself does not contain attention mechanisms, recent work by Lieber et al. [31] introduced Jamba, which combines Mamba layers with self-attention layers in an interleaved pattern. This hybrid approach aims to leverage the strengths of both architectures: Mamba's efficient sequence processing and attention's ability to capture global dependencies.

The success of these hybrid architectures suggests that while Mamba provides an efficient alternative to attention, the combination of different architectural components may offer the best path forward for complex sequence processing tasks like code analysis.

2.1.5 Mixture of Experts

Mixture of Experts (MoE) has emerged as a powerful approach for building efficient, scalable neural networks by conditionally activating different subnetworks based on input characteristics. This architecture has become particularly relevant for large language models and code analysis tasks where computational efficiency is crucial.

Fundamental Architecture

The basic MoE architecture [37] consists of a set of expert networks and a router that directs inputs to specific experts:

$$y = \sum_{i=1}^{n} g(x)_i E_i(x) \tag{2.15}$$

where $g(x)_i$ represents the router's gate value for expert i, E_i is the i-th expert network, and n is the total number of experts. The router typically implements a sparse gate:

$$g(x) = \text{top-k}(\text{softmax}(W_g x)) \tag{2.16}$$

where top-k selects the k highest values, usually with $k \ll n$.

Scaling with Sparsity

Fedus, Zoph, and Shazeer [33] introduced the Switch Transformer, demonstrating that MoE architectures can effectively scale model capacity while maintaining computational efficiency. Key innovations include load balancing across experts through auxiliary loss terms, efficient routing strategies that reduce communication overhead, and capacity factor tuning to control expert utilization.

The load balancing loss is typically formulated as:

$$\mathcal{L}_{\text{balance}} = \alpha \cdot \text{mean}(P^2) \cdot n \tag{2.17}$$

where P is the router probability distribution and α is a balancing coefficient.

Applications in Code Analysis

MoE architectures have shown particular promise in code analysis tasks due to their ability to specialize different experts for different types of code patterns. Zhou et al. [3] demonstrated that MoE-based models can effectively analyze different programming language constructs, handle varying levels of code complexity, and process multiple programming paradigms efficiently.

Expert Specialization

Recent work has shown that experts in MoE models tend to specialize in specific patterns or tasks. For code analysis, this specialization often aligns with different programming language syntax, various code vulnerability patterns, and distinct code structure types (e.g., loops, conditionals, function definitions).

Efficiency Considerations

MoE models achieve efficiency through conditional computation, where only a subset of parameters is active for any given input. The effective FLOPS per token can be expressed as:

$$\text{FLOPS}_{\text{effective}} = \text{FLOPS}_{\text{base}} \cdot \frac{k}{n} \tag{2.18}$$

where k is the number of active experts and n is the total number of experts.

2.1.6 Scale-Invariant Fine-Tuning

Scale-Invariant Fine-Tuning (SIFT) represents a significant advancement in model training techniques, particularly for improving model robustness and generalization. This approach has become especially relevant for security-critical applications like code vulnerability detection, where model reliability is paramount.

Theoretical Foundation

SIFT builds upon the principles of adversarial training [38] and extends them to create more robust models through scale-invariant perturbations. The core idea is to maintain model performance across different scales of input perturbations:

$$\mathcal{L}_{\text{SIFT}} = \mathbb{E}_{x \sim \mathcal{D}}[\max_{\delta \in \Delta} \mathcal{L}(f_\theta(x + \delta), y)] \tag{2.19}$$

where f_θ represents the model, $\mathcal{D}$ is the data distribution, and Δ defines the set of allowed perturbations.

Implementation Mechanics

The practical implementation of SIFT involves several key components:

1. **Perturbation Layer:** A learnable layer that generates input perturbations:

$$\delta = \alpha \cdot \tanh(W_p x) \tag{2.20}$$

where α controls the perturbation magnitude and W_p are learned parameters.

2. **Scale-Invariant Loss:** The combined loss function incorporating both task-specific and robustness objectives:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{task}}(f_\theta(x), y) + \lambda \mathcal{L}_{\text{SIFT}}(f_\theta(x + \delta), f_\theta(x)) \tag{2.21}$$

where λ balances the two objectives.

Application to Code Analysis

In the context of code vulnerability detection, SIFT provides several key advantages. Models become more resilient to syntactic variations that preserve semantic meaning, demonstrate better performance on previously unseen code patterns, and show enhanced resistance to adversarial attacks that might try to deceive vulnerability detection.

Training Dynamics

The training process with SIFT involves a careful balance of different components:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta (\mathcal{L}_{\text{task}} + \lambda \mathcal{L}_{\text{SIFT}}) \tag{2.22}$$

where η is the learning rate and θ represents the model parameters. The gradients from both the task loss and SIFT loss are combined to update the model parameters.

Impact on Model Performance

SIFT has demonstrated significant improvements in model robustness metrics. These improvements include reduced sensitivity to input perturbations, more consistent performance across different code styles, and better generalization to out-of-distribution samples.

These improvements are particularly valuable in security-critical applications where model reliability is essential.

Chapter 3

Methodology

The development of White-Basilisk was driven by three key challenges in code vulnerability detection: addressing long-range dependencies across code sequences, balancing local and global information processing, and maintaining computational efficiency. This chapter details our architectural approach to these challenges and our methodology for training and evaluation.

3.1 Model Architecture

White-Basilisk's architecture represents a novel hybrid design that combines three main components: Mamba layers for efficient sequence processing, linear-complexity Infini-attention for handling extended contexts, and a Mixture of Experts framework for dynamic adaptation. The synergy between these components enables White-Basilisk to process sequences up to 128,000 tokens during inference, facilitating holistic analysis of entire codebases.

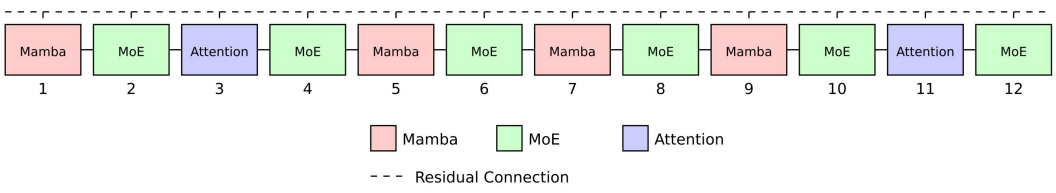

FIGURE 3.1: White-Basilisk Architecture

The layer combination pipeline in our architecture is defined by two main configuration parameters, determined through experimentation:

- Attention layer offset: 2
- Attention layer period: 8

The conjunction of layers is formally described by:

$$\text{Layer}_i = \begin{cases} \text{Attention}(x), & \text{if } (i-2) \bmod 8 = 0 \text{ and } i \geq 2 \\ \text{MoE}(x), & \text{if } i \bmod 2 = 1 \\ \text{Mamba}(x), & \text{otherwise} \end{cases} \tag{3.1}$$

The forward pass through the model can be expressed as:

$$h_i = \text{Layer}_i(h_{i-1}) + h_{i-1} \tag{3.2}$$

where h_i is the hidden state after the i-th layer, and h_0 is the initial input embedding. The final output of the model is obtained by applying layer normalization to the last hidden state:

$$y = \text{LayerNorm}(h_L) \tag{3.3}$$

3.1.1 Mamba Layers

Mamba layers form the backbone of White-Basilisk, chosen for their exceptional efficiency in capturing local dependencies in code sequences. Unlike traditional recurrent neural networks (RNNs) or attention mechanisms, Mamba's selective state-space mechanism allows for linear-time computation with respect to sequence length. This efficiency is crucial for processing long code sequences without excessive computational overhead. The adaptive nature of Mamba layers enables the model to focus computational resources on the most relevant parts of the input, making it highly efficient for detecting subtle patterns that may indicate vulnerabilities.

In White-Basilisk, Mamba layers are implemented with:

- State size (d_{state}): 16
- Convolution kernel size (d_{conv}): 4
- Expansion factor: 2

The core computation in a Mamba layer can be summarized as:

$$y = \Delta \odot (Ax + Bu) + Cu \tag{3.4}$$

where Δ, B, and C are input-dependent parameters, A is a fixed parameter, x is the layer input, u is the input projection, and $\odot$ denotes element-wise multiplication. The selective state space mechanism in Mamba allows for efficient processing of long sequences, making it particularly suitable for code analysis tasks.

3.1.2 Linear-complexity Infini-attention: A Novel Adaptation

Our implementation of Infini-attention is a key factor behind White-Basilisk's ability to handle efficiently extremely long code sequences. Traditional attention mechanisms face challenges with long sequences due to their quadratic complexity, making them computationally prohibitive for whole-codebase analysis. By contrast, our linear-complexity adaptation of Infini-attention enables White-Basilisk to process entire codebases with significantly reduced computational requirements. This efficiency is essential for real-world vulnerability detection, allowing our model to consider broad context and identify vulnerabilities that may stretch across multiple functions or files, while maintaining feasible processing times and memory usage.

Specifically, we propose a novel implementation of the original algorithm proposed by [32], allowing for efficient processing of arbitrarily long sequences while maintaining the ability to capture long-range dependencies. The primary differences are:

1. **Accumulation and Linear Complexity:** Unlike the original Infini-attention, which processes segments independently with bounded memory usage, our implementation accumulates outputs across all segments:

$$\text{total}_{\text{mem}} = \sum_{s=1}^{S} A_{\text{mem,s}}, \quad \text{total}_{\text{attn}} = \sum_{s=1}^{S} A_{\text{dot,s}} \tag{3.5}$$

   where S is the total number of segments. This accumulation leads to linear memory growth with sequence length, trading off bounded memory for the ability to process arbitrarily long sequences.

2. **Global Gating Mechanism:** As a consequence of accumulation, our gating mechanism operates globally on the entire accumulated context, rather than segment-by-segment:

$$O = \text{sigmoid}(\beta) \odot \text{total}_{\text{mem}} + (1 - \text{sigmoid}(\beta)) \odot \text{total}_{\text{attn}} \tag{3.6}$$

This allows for a more holistic balancing of local and global information across the entire sequence.

Moreover, our linear-complexity Infini-attention maintains the core concept of combining local attention and a compressive memory, but adapts it for extremely long sequence processing. The memory retrieval and update processes remain similar:

$$A_{\text{mem}} = \frac{(\text{ELU}(Q) + 1)M^T}{(\text{ELU}(Q) + 1)z^T + \epsilon} \tag{3.7}$$

$$M_{\text{new}} = M + (\text{ELU}(K)^T + 1)V, \quad z_{\text{new}} = z + \sum_{i=1}^{L}(\text{ELU}(K_i) + 1) \tag{3.8}$$

where M is the compressive memory, z is the normalization term, and L is the segment length.

In combination with Mamba layers, which process the entire sequence to capture global patterns, our linear-complexity Infini-attention enables White-Basilisk to effectively balance local and global information processing across very long sequences. This synergy allows the model to maintain high performance on tasks requiring understanding of both fine-grained local context and broad, long-range dependencies, all while scaling efficiently to extreme sequence lengths.

3.1.3 Mixture of Experts (MoE) Layers

We incorporate Mixture of Experts (MoE) layers into our model to introduce dynamic adaptability while maintaining computational efficiency. MoE layers allow the model to activate only a subset of parameters for each input, reducing significantly the computational cost compared to fully-dense models of similar capacity. In the context of code vulnerability detection, this efficiency is crucial as it allows our model to handle effectively diverse types of code and potential vulnerabilities without adding to its complexity and number of parameters. The sparsity induced by MoE layers also contributes to faster inference times, a critical factor in real-world deployment scenarios.

The MoE layers in White-Basilisk are configured with 8 Experts and 2 Experts per token. For an input x, the output of an MoE layer is computed as:

$$y = \sum_{i=1}^{k} G(x)_i E_i(x) \tag{3.9}$$

where $G(x)$ is the output of the router (gating function), E_i is the i-th expert, and $k = 2$ is the number of experts per token. The router uses a top-k gating mechanism to select the most relevant experts for each token, allowing the model to dynamically adapt its processing based on the input characteristics.

3.1.4 Classification Head

For the final vulnerability detection task, we designed an efficient yet powerful classification head that goes beyond a simple linear layer. The architecture progressively refines the high-dimensional representations learned by the main model into classification outputs through multiple stages of transformation. This design balances model capacity and computational efficiency while ensuring robust vulnerability detection.

The classification head structure consists of:

1. **Dense Layer 1:** A fully connected layer that projects the hidden state (dimension 512) to the same dimension. This layer uses a GELU activation function and is followed by dropout for regularization.
2. **Dense Layer 2:** Another fully connected layer that reduces the dimension from 512 to 256, again followed by GELU activation and dropout.
3. **Layer Normalization:** Applied to the output of Dense Layer 2 for improved stability and faster convergence.
4. **Output Layer:** A final linear layer that projects from 256 dimensions to the number of classes (2 for binary classification of vulnerable vs. non-vulnerable code).

This progressive dimension reduction helps distill the relevant features for vulnerability detection while maintaining model efficiency. Mathematically, the classification head can be expressed as:

$$\begin{aligned} x_1 &= \text{Dropout}(\text{GELU}(W_1 h + b_1)) \\ x_2 &= \text{Dropout}(\text{GELU}(W_2 x_1 + b_2)) \\ x_3 &= \text{LayerNorm}(x_2) \\ y &= W_3 x_3 + b_3 \end{aligned} \tag{3.10}$$

where $h \in \mathbb{R}^{512}$ is the input from the main model, $W_1 \in \mathbb{R}^{512\times512}$, $W_2 \in \mathbb{R}^{256\times512}$, and $W_3 \in \mathbb{R}^{2\times256}$ are learnable weights, and b_1, b_2, b_3 are biases.

The use of GELU activations and layer normalization aligns with modern best practices in deep learning architecture design, while the multiple transformation stages allow the model to learn more complex patterns than would be possible with a simple linear classifier. This design choice was particularly important given the subtle and complex nature of code vulnerabilities, which often require understanding intricate patterns and relationships in the code.

3.2 Data Collection and Preprocessing

The development and evaluation of source code vulnerability detection models requires a large collection of annotated data samples. In this section, we outline the datasets chosen for this purpose and explain how they were used for both model training and testing purposes. Additionally, we provide a detailed overview of the training methodology used for our model.

3.2.1 Dataset Selection and Composition

The datasets used in our analyses were divided into two categories: model training and benchmarking. For training, we initially selected a carefully curated subset of the StarCoder dataset ([29]), which includes more than 80 programming languages and consists of 305M files in total. For our study, we focused on C and C++ code samples, using 2M code samples during the pre-training phase. To evaluate the pre-trained model, we required well-established benchmarking datasets with publicly available partitions for fine-tuning and testing. This ensures a fair comparison with existing methods without the need to recreate the original models. For this purpose, we selected five publicly available datasets: VulDeePecker ([8]), Draper ([6]), PrimeVul ([4]), REVEAL ([7]), and BigVul ([5]).

3.2.2 Dataset Statistics and Characteristics

The selected datasets exhibit diverse characteristics in terms of size, class distribution, and sequence lengths: These statistics significantly influenced our model design decisions. The substantial class imbalance across all datasets

TABLE 3.1: Dataset Distribution and Vulnerability Statistics

Dataset	Sample Count			Vulnerable (%)			Duplicates (%)		
	Train	Val	Test	Train	Val	Test	Train	Val	Test
Draper	1,019,471	127,476	127,419	6.46	6.47	6.48	0.00	0.00	0.00
PRIMEVUL	184,427	25,430	25,911	3.02	2.75	2.68	0.00	0.00	0.00
BigVul	150,908	18,864	18,864	5.79	5.88	5.59	0.005	0.00	0.00
VulDeepecker	128,118	16,015	16,015	6.08	6.08	6.08	37.72	20.27	20.33
REVEAL	18,187	2,273	2,274	9.90	9.24	10.11	1.17	0.18	0.09

TABLE 3.2: Sequence Length Statistics (Training Split)

Dataset	Min	Max	Mean	Median	95th %
PRIMEVUL	3	296,924	502.44	193.0	1,729.0
VulDeepecker	8	312,940	284.03	142.0	893.0
REVEAL	10	120,684	569.05	226.0	1,929.7
BigVul	6	70,440	343.90	147.0	1,193.0
Draper	10	42,492	320.85	236.0	858.0

(ranging from 3.02% to 10.11% vulnerable samples) motivated our implementation of specialized class weighting and sampling strategies. The extreme range in sequence lengths (from 3 to 312,940 tokens) justified our focus on developing an architecture capable of handling very long sequences efficiently. The varying levels of data duplication (0% to 37.72%) highlighted the importance of robust evaluation metrics and careful interpretation of results, particularly for VulDeepecker. The consistency of class distributions across splits suggests that our evaluation metrics should be reliable indicators of real-world performance. REVEAL's higher proportion of vulnerable samples (10%) compared to other datasets (3-6%) provides an important test case for our model's ability to handle different class balance scenarios.

3.2.3 Class Imbalance Handling

To address class imbalance in code vulnerability detection, we employ a dual approach affecting both data sampling and loss computation:

$$w_c = \frac{N}{2N_c} \tag{3.11}$$

where w_c is the weight for class c and N_c is the number of samples in class c. Using the class weights we implement a Weighted Random Batch Sampler and a loss function, weighted by w_c for each sample.

This approach ensures balanced class representation in mini-batches and adjusts the loss to emphasize underrepresented classes, crucial for effective vulnerability detection in imbalanced datasets.

3.3 Training Pipeline

Traditional LLM training methods are generally designed to enable models to comprehend language and its syntax. This is often accomplished through Causal Language Modeling (CLM), where a model learns to predict the next token given its input. Another common approach is based on the Fill in the Middle (FIM) technique, in which random text portions are masked, and the model must reconstruct the missing content. Some advanced source code LLMs combine both methods to increase model flexibility ([29]). Similarly, in our work, we employ both techniques during model pre-training on the selected 2M code samples. This process requires approximately 600 hours to complete on a single NVIDIA A100 40GB GPU.

SIFT (Scale-Invariant Fine-Tuning)

We implement automated adversarial training using SIFT to improve the model's resilience against adversarial examples. SIFT operates by introducing small perturbations to the input during training, encouraging the model to learn more robust features. In our implementation, we added a PerturbationLayer into the model architecture, which applies learnable perturbations to the input embeddings. The training process was designed to minimize both the task loss and the adversarial loss, the latter being computed as the difference between predictions on clean and perturbed inputs.

This approach confers several advantages:

- Improved model generalization
- Enhanced robustness to minor variations in input
- Better resilience against potential adversarial attacks

Such resilience is crucial in the domain of code vulnerability detection, where the model must maintain consistent performance across diverse code samples and potential adversarial inputs. By adopting this technique, we have

effectively imbued the model with a form of 'adversarial immunity', rendering it more resilient against potential attacks or attempts to deceive its analysis. This enhanced robustness is particularly valuable in security-critical applications, where the reliability and consistency of the model's performance are of paramount importance.

3.3.1 Hyperparameters

Our training process utilizes the following key hyperparameters:

TABLE 3.3: Model Training Hyperparameters

Parameter	**Pretraining**	**Fine-tuning**
Learning Rate	1.41e-5	5e-6
Batch Size	16	4
Number of Epochs	10	10
Warmup Ratio	0.15	0.15
Optimizer	AdamW	AdamW
Weight Decay	0.01	0.01
FIM and FIM-SPM Rates	0.5	-

3.3.2 Training Procedure

The training process consists of two main phases:

1. **Pretraining Phase**: - Combined CLM and FIM training on 2M code samples - Duration: Approximately 600 hours - Hardware: Single NVIDIA A100 40GB GPU - Objective: Develop general code understanding capabilities

2. **Fine-tuning Phase**: - Dataset-specific training with SIFT - Smaller learning rate and batch size to prevent catastrophic forgetting - Focus on adapting to specific vulnerability detection patterns

This training pipeline ensures that White-Basilisk develops both broad code understanding capabilities and specialized vulnerability detection skills while maintaining robustness against adversarial inputs.

3.4 Evaluation Framework

To evaluate our model's performance comprehensively, we established a rigorous evaluation framework encompassing multiple benchmarks and metrics. All datasets are evaluated on binary classification (0 = Safe, 1 = Vulnerable), using the same data splits as the baseline models to ensure fair comparison. The metrics for models other than White-Basilisk were sourced from their respective publications.

3.4.1 Evaluation Metrics

Given the significant class imbalance observed in the datasets (with vulnerable samples comprising only 3-10% of the data) and the critical importance of detecting vulnerabilities, we selected F1 score as our primary evaluation metric. The F1 score is particularly crucial for vulnerability detection tasks for several reasons. It provides a balance between precision and recall, as in security applications, both false positives (incorrectly flagging safe code as vulnerable) and false negatives (missing actual vulnerabilities) can have serious consequences. The F1 score offers a balanced measure of both aspects. It also demonstrates robustness to class imbalance, unlike accuracy, which can be misleading in imbalanced datasets. F1 score remains meaningful even with highly skewed class distributions, which is particularly important given that our datasets contain as few as 3.02% vulnerable samples. The metric has practical utility as well, since in real-world deployment scenarios, security teams need to balance the thoroughness of vulnerability detection with resource constraints. A high F1 score indicates that the model can effectively identify vulnerabilities without overwhelming developers with false alarms. Additionally, we considered the Vulnerability Detection Score (VD-S), a novel metric introduced by [4]. VD-S specifically evaluates a model's ability to minimize false negatives while maintaining a very low false positive rate. This metric is particularly significant because it focuses on the critical aspect of not missing vulnerabilities (false negatives), enforces a strict requirement on false positives, ensuring practical usability, provides a more stringent evaluation of model performance in security-critical contexts, and better reflects real-world deployment requirements where false positives must be minimized to maintain developer productivity.

Our evaluation metrics include:

- **Accuracy**: Measures the proportion of correct predictions among all predictions:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.12}$$

- **Precision**: Measures the proportion of correct positive predictions among all positive predictions:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.13}$$

- **Recall**: Measures the proportion of actual positives correctly identified:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.14}$$

- **F1 Score**: The harmonic mean of precision and recall:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.15}$$

- **Vulnerability Detection Score (VD-S)**: Evaluates the False Negative Rate at a specific False Positive Rate (FPR) threshold:

$$\text{VD-S} = \frac{FN}{FN + TP} \text{ at } FPR \leq 0.005 \tag{3.16}$$

3.4.2 Benchmark Datasets

We evaluated White-Basilisk across five widely-used datasets in code vulnerability detection:

1. **PRIMEVUL** ([4]): A comprehensive dataset focusing on real-world vulnerabilities
2. **BigVul** ([5]): Large-scale dataset with diverse vulnerability types
3. **Draper** ([6]): Contains carefully curated vulnerability instances
4. **REVEAL** ([7]): Focuses on multi-class vulnerability detection
5. **VulDeepecker** ([8]): Classic dataset with established benchmarks

3.4.3 Baseline Models

For comparative analysis, we included several state-of-the-art models:

- CodeT5 [39]: 60M parameters, multi-lingual code model
- CodeBERT [26]: 125M parameters, bimodal pre-training
- UnixCoder [27]: 125M parameters, cross-modal architecture
- StarCoder2 [29]: 7B parameters
- CodeGen2.5 [40]: 7B parameters
- VulBERTa variants [22]: MLP and CNN versions
- LineVul [23]: Based on CodeBERT

3.4.4 Evaluation Strategy

Our evaluation strategy follows these key principles. We maintain consistent use of original dataset splits to ensure fair comparison with baseline models. Our approach includes evaluation across multiple metrics to provide comprehensive performance assessment. We conduct separate analysis of performance on different CWE (Common Weakness Enumeration) categories. Our strategy considers both overall accuracy and vulnerability-specific metrics. Finally, we assess model performance across varying sequence lengths. This comprehensive evaluation framework enables us to assess White-Basilisk's performance not only in terms of raw accuracy but also in terms of its practical utility for real-world vulnerability detection tasks.

Chapter 4

Experimental Results

4.1 Experimental Results: Small Model, Big Impact

To evaluate our model's performance, we conducted extensive experiments across five widely-used datasets in code vulnerability detection: PRIMEVUL, BigVul, Draper, REVEAL, and VulDeepecker. All datasets are evaluated on **binary classification** (0 = Safe, 1 = Vulnerable). To ensure a fair comparison, we used the same data splits as the baseline models. The metrics for models other than White-Basilisk were sourced from their respective publications. Across all datasets, White-Basilisk consistently demonstrated superior performance, exceeding that of larger and more resource-intensive models.

Given the class imbalance observed in the datasets and the significance of the minority class (vulnerable samples), we opted for F1 score as our primary evaluation metric. A high F1 score reflects the model's ability to identify vulnerabilities accurately while minimizing false positive/negative cases, thus achieving the critical balance needed in real-world security applications. Additionally, we considered a novel Vulnerability Detection Score (VD-S), introduced by [4], which evaluates the False Negative Rate of a detector (1-Recall).

On PRIMEVUL, it achieved an F1 Score of 29.07% and VD-S of 72.39, significantly surpassing larger models. For the Draper dataset, White-Basilisk set a new standard with an F1 Score of 60.69%. On REVEAL, it scored an F1 of 49.34% and accuracy of 89.88%, outperforming the next best model. VulDeepecker saw White-Basilisk achieve an F1 Score of 93.88% and the highest precision at 97.20%. Most impressively, on BigVul, White-Basilisk dramatically outperformed all competitors across all metrics with an F1 Score of 94.90%, accuracy of 99.42%, and VD-S of 3.98.

Our experiments reveal that White-Basilisk, with its modest 200 million parameters, not only competes with but often surpasses the performance of much larger models across diverse code vulnerability detection datasets. Tables 4.1, 4.2 summarizes these remarkable results, showcasing White-Basilisk's prowess in this critical domain.

4.1.1 Performance Metrics: Precision in Vulnerability Detection

In the realm of code vulnerability detection, two metrics stand out as particularly crucial:

- **F1 Score**: This balanced measure of precision and recall is vital in vulnerability detection. A high F1 score indicates the model's ability to identify vulnerabilities accurately while minimizing false alarms, striking the delicate balance needed in real-world security applications.
- **Vulnerability Detection Score (VD-S)**: This metric was introduced by the PrimeVul Dataset and evaluates the False Negative Rate of a detector. A lower VD-S signifies fewer missed vulnerabilities, which is critical in preventing potential security breaches.

4.1.2 Efficient Design, Superior Results: White-Basilisk's Paradigm

The superior performance of White-Basilisk is the result of a cutting-edge combination of architectural innovations and advanced training techniques. The model's architecture integrates Mamba layers, linear-complexity Infini-attention, and a Mixture of Experts framework, allowing it to efficiently process extended code sequences while simultaneously capturing both local and global dependencies. This design enables our model to process sequences of up to 128,000 tokens during inference, all with a single NVIDIA A100 40GB GPU.

This extended context window represents a significant advancement in code analysis capabilities, enabling a comprehensive examination of entire codebases or extensive code files in a single computational pass. This holistic approach enables the detection of long-range dependencies and contextual nuances that are frequently overlooked by models with more limited context lengths. As a result, our model excels at identifying complex vulnerabilities, particularly those related to inter-functional or cross-file data flow.

White-Basilisk's improved performance is due to its advanced training approach, incorporating various sophisticated techniques. A hybrid pretraining strategy, combining CLM and FIM pretraining enhances the model's comprehension of code structure and context. Moreover, the implementation of SIFT increases the model's adversarial robustness, while specialized methodologies for addressing class imbalance optimise learning from heterogeneous datasets.

The efficacy of this approach is empirically validated by White-Basilisk's performance across several benchmarks. Empty cells indicate that the metric was not reported in the original study. Different metrics are reported for each dataset based on the original studies. F1 score is consistently reported across all models and datasets. On the PRIMEVUL dataset, it achieved an F1 Score of 29.07% and a Vulnerability Detection Score (VD-S) of 72.39, significantly outperforming models with larger parameter counts. Its performance on the BigVul dataset was particularly noteworthy, with an F1 Score of 94.90%, accuracy of 99.42%, and VD-S of 3.98, surpassing all competing models across all evaluated metrics. On the Draper dataset, White-Basilisk established a new benchmark with an F1 Score of 60.69%. For the REVEAL dataset, it attained an F1 Score of 49.34% and accuracy of 89.88%, exceeding the performance of the next highest-performing model. When evaluated on VulDeepecker, White-Basilisk demonstrated exceptional precision, achieving an F1 Score of 93.88% and the highest precision at 97.20%.

TABLE 4.1: BigVul and PRIMEVUL Evaluation Results

Model	PRIMEVUL			BigVul		
	Acc (%)	F1 (%)	VD-S ↓	Acc (%)	F1 (%)	VD-S ↓
White-Basilisk	96.30	**29.07**	**72.39**	**99.42**	**94.90**	**3.98**
CodeT5	96.67	19.70	89.93	95.67	64.93	77.30
CodeBERT	<u>96.87</u>	20.86	88.78	95.57	62.88	81.77
UnixCoder	96.86	21.43	89.21	96.46	65.46	62.30
UnixCoder w/ balancing	95.99	<u>26.28</u>	<u>88.49</u>	-	-	-
StarCoder2	**97.02**	18.05	89.64	96.20	68.26	69.14
CodeGen2.5	96.65	19.61	91.51	<u>96.57</u>	67.30	61.73
LineVul	-	-	-	-	<u>91.00</u>	<u>14.00</u>

TABLE 4.2: Draper, REVEAL, and VulDeepecker Evaluation Results

Model	Draper	REVEAL		VulDeepecker	
	F1 (%)	Acc (%)	F1 (%)	F1 (%)	Prec (%)
White-Basilisk	**60.69**	**89.88**	**49.34**	**93.88**	**97.20**
Russell et al. [6]	56.60	-	-	-	-
VulBERTa-MLP	43.34	84.48	45.27	93.03	95.76
VulBERTa-CNN	57.92	79.73	42.59	90.86	95.26
Baseline-BiLSTM	46.84	77.13	39.11	66.97	52.58
Baseline-TextCNN	49.40	73.22	37.41	75.80	63.48
REVEAL	-	84.37	41.25	-	-
VulDeepecker	-	-	-	92.90	91.90

4.2 Discussion: Rethinking AI Efficiency

The remarkable performance of White-Basilisk, achieved with only 200M parameters, challenges fundamental assumptions in AI development and offers insights into potential new directions for the field. This efficiency prompts a critical reexamination of the relationship between model size, performance, and computational resources in AI.

White-Basilisk's success suggests a more nuanced relationship between model size and performance than previously assumed. While larger models like GPT have demonstrated impressive capabilities across a wide range of tasks, our results show that for specialized tasks like code vulnerability detection, carefully designed smaller models can achieve comparable or superior performance. This indicates that the relationship between model size and performance may be task-dependent, with a point of diminishing returns, beyond which additional parameters do not necessarily translate to improved performance.

The success of White-Basilisk's hybrid architecture, combining Mamba layers, linear-complexity Infini-attention, and a Mixture of Experts framework, highlights the potential of architectural innovation as an alternative to simple scaling. This approach allows for more efficient use of parameters, potentially offering a way to break through the computational barriers that currently limit the scaling of AI models. Our findings suggest that future advancements in AI may come not just from increasing model size, but from novel architectures that more efficiently leverage available parameters.

The environmental implications of AI model development are brought into sharp focus by our results. Based on available information about pretraining

procedures, we estimated the approximate CO2 emissions during training for White-Basilisk and several competitor models using the Machine Learning Impact calculator presented in [41] (Table 4.3). The stark contrast in CO2 emissions between White-Basilisk and larger models (85.5 kg vs. 29,622.83 kg for StarCoder) underscores the environmental impact of AI development choices. This massive difference suggests that the AI community needs to seriously consider the environmental costs of model development and deployment. Our results demonstrate that it's possible to achieve state-of-the-art performance with a fraction of the environmental impact of larger models, opening up new possibilities for sustainable AI development.

However, it's important to note that the total environmental impact of an AI model depends not just on its training, but also on its inference costs over its lifetime of use. The long-term environmental implications of deploying many specialized models versus fewer general-purpose models. This consideration adds another layer of complexity to the efficiency-performance trade-off in AI development.

The success of White-Basilisk also suggests that our current metrics for evaluating AI models may be insufficient. While performance on benchmark tasks remains important, our results indicate that we should also consider metrics related to efficiency, scalability, and environmental impact. Developing a more holistic set of evaluation criteria could drive the field towards more sustainable and efficient AI development practices.

TABLE 4.3: Comparison of CO2 Emissions

	White-Basilisk	CodeBERT	StarCoder	UnixCoder	CodeT5	Gasoline Car (per Year)
CO2 (kg)	**85.5**	2,240	29,622.83	2,048	1,136	4,600

4.3 Ablation Study: Combined Dataset Training

To further evaluate White-Basilisk's performance and investigate the impact of training data composition, we conducted an ablation study using a combined training approach across all datasets. This experiment involved concatenating the training splits from all five datasets (REVEAL, Draper, VulDeepecker, BigVul, and PRIMEVUL) into a single unified training set. During each training epoch, the model was evaluated on the concatenated testing splits from all datasets to monitor for potential overfitting. Final performance metrics were obtained by evaluating the trained model separately on each

dataset's designated validation split, ensuring fair comparison with previous results.

4.3.1 Experimental Setup

The model was trained using the same hyperparameters as described in Section 6.2, but with the following data configuration:

- **Training Data:** Combined training splits from all five datasets into a single training set
- **Testing:** Concatenated testing splits from all datasets, used for monitoring training progress
- **Validation:** Individual Validation splits for each dataset, evaluated separately to assess dataset-specific performance

This unified training approach resulted in a significantly larger and more diverse training set, allowing us to investigate how the model performs when exposed to a broader range of vulnerability patterns and coding styles simultaneously.

4.3.2 Results and Analysis

The results of this combined training approach are presented in Table 4.4.

TABLE 4.4: Combined Training Results Across All Datasets

Dataset	Precision	Recall	F1	Accuracy
REVEAL	0.416	0.551	0.470	0.888
Draper	0.568	0.532	0.549	0.948
VulDeepecker	0.939	0.912	0.925	0.989
BigVul	0.936	0.940	0.938	0.993
PRIMEVUL	0.268	0.265	0.233	0.952
Combined Test	0.643	0.585	0.613	0.956

The model maintains strong performance across most datasets, with particularly robust results on VulDeepecker (F1: 0.925) and BigVul (F1: 0.938), suggesting effective transfer learning across different vulnerability detection tasks. Performance varies significantly across datasets, from an F1 score of 0.938 on BigVul to 0.233 on PRIMEVUL, indicating that some vulnerability patterns may be more challenging to learn in a combined setting. The model maintains high accuracy across all datasets (0.888-0.993), demonstrating robust overall classification performance even with the increased complexity of

the combined training task. The model generally maintains a good balance between precision and recall, with some datasets showing nearly identical values (e.g., BigVul: 0.936/0.940), suggesting stable learning of vulnerability patterns.

4.3.3 Model Robustness Analysis

A particularly noteworthy aspect of these results is White-Basilisk's ability to maintain stable performance across multiple diverse datasets without experiencing catastrophic forgetting or overfitting. This is especially significant given the model's relatively compact size of 200M parameters. Several factors contribute to this robustness. The model shows signs of positive transfer learning, where knowledge gained from one dataset appears to benefit the detection of vulnerabilities in others. This is evidenced by the maintenance of high accuracy scores across all datasets despite their varying characteristics. Additionally, the model maintains performance across datasets of different sizes and complexity levels, from the smaller REVEAL dataset to the larger PRIMEVUL dataset. This stability suggests that the model's learning capacity is well-matched to the task complexity.

4.3.4 Comparison with Individual Training

When compared to the individual training results presented in Section 5, the combined training approach shows some interesting trade-offs. For some datasets (VulDeepecker, BigVul), the performance remains close to individual training results, suggesting that the model can effectively learn and maintain dataset-specific patterns even in a combined setting. Performance on more challenging datasets like PRIMEVUL shows some degradation, indicating that the increased diversity of the training data may make it harder for the model to capture some of the more nuanced vulnerability patterns specific to certain datasets. The overall combined test metrics (F1: 0.613, Accuracy: 0.956) demonstrate that White-Basilisk can effectively learn from multiple datasets simultaneously while maintaining reasonable performance across all of them.

This ability to maintain stable performance across diverse datasets without overfitting or experiencing catastrophic forgetting is particularly notable for a model of this size. It suggests that White-Basilisk's architecture strikes an effective balance between model capacity and efficiency, enabling robust

multi-task learning without requiring the massive parameter counts typically associated with such capabilities. This finding has important implications for the development of efficient, multi-purpose vulnerability detection systems that can be deployed in resource-constrained environments while maintaining high performance across a range of vulnerability types.

4.4 Ablation Study: Attention Mechanisms and Long-Range Vulnerability Detection

To thoroughly evaluate White-Basilisk's performance and validate our architectural choices, we conducted a comprehensive ablation study focusing on two key aspects: (1) the effectiveness of our linear-complexity Infini-attention mechanism compared to standard self-attention, and (2) the model's performance across varying sequence lengths. This analysis is particularly important given the prevalence of vulnerabilities that span multiple functions or files, requiring models to maintain effectiveness over long code sequences.

4.4.1 Experimental Setup

Model: We used the White-Basilisk checkpoint that was trained on the combined datasets from 4.3.
We categorized sequences into four length bins for analysis:

- Bin 1: 0-16,384 tokens (standard context length)
- Bin 2: 16,384-32,768 tokens (extended context)
- Bin 3: 32,768-65,536 tokens (long context)
- Bin 4: 65,536-131,072 tokens (very long context)

For each bin, we monitored both memory consumption and model performance across all datasets, comparing our Infini-attention implementation against standard self-attention (eager implementation).

4.4.2 Memory Efficiency Analysis

Table 4.5 presents the peak memory consumption across different sequence lengths and attention implementations.

TABLE 4.5: Peak Memory Consumption (MB) by Sequence Length and Attention Type

Length Bin	Infini-attention		Standard Attention	
	Peak	Reserved	Peak	Reserved
Bin 1 (0-16K)	1,338	1,442	32,409	39,696
Bin 2 (16K-32K)	1,654	1,956	OOM	OOM
Bin 3 (32K-65K)	2,213	2,638	OOM	OOM
Bin 4 (65K-131K)	3,322	3,848	OOM	OOM

OOM: Out of Memory on NVIDIA A100 40GB GPU

The results demonstrate the significant memory advantages of our Infini-attention approach. Infini-attention shows near-linear memory scaling, increasing from 1.3GB to 3.3GB across bins. Standard attention requires 24x more memory for Bin 1 and fails entirely on longer sequences. While standard attention becomes infeasible beyond 16K tokens, Infini-attention successfully processes sequences up to 131K tokens with modest memory requirements.

4.4.3 Performance Analysis Across Sequence Lengths

Table 4.6 presents a comprehensive analysis of performance across different sequence lengths and datasets.

TABLE 4.6: Performance and Distribution Analysis Across Sequence Lengths

		Sample Distribution			Infini-attention			Eager-attention		
Dataset	Bin	Total	Non-Vuln	Vuln	F1	Precision	Accuracy	F1	Precision	Accuracy
BigVul	Bin 1	18,853	17,747	1,106	0.943	0.946	0.993	0.937	0.967	0.993
	Bin 2	7	5	2	0.800	0.667	0.857	-	-	-
	Bin 3	4	3	1	1.000	1.000	1.000	-	-	-
VulDeePecker	Bin 1	12,764	11,814	950	0.925	0.939	0.989	0.932	0.968	0.990
	Bin 2	2	2	0	-	-	1.000	-	-	-
	Bin 3	2	2	0	-	-	1.000	-	-	-
	Bin 4	1	1	0	-	-	1.000	-	-	-
PRIMEVUL	Bin 1	25,411	24,715	696	0.233	0.208	0.952	0.240	0.237	0.958
	Bin 2	18	15	3	0.250	0.200	0.667	-	-	-
	Bin 3	1	1	0	-	-	1.000	-	-	-
REVEAL	Bin 1	2,269	2,062	207	0.474	0.416	0.888	0.468	0.445	0.898
Draper	Bin 1	12,769	11,819	950	0.568	0.609	0.948	0.511	0.646	0.948

Several remarkable findings emerge from this analysis. White-Basilisk demonstrates remarkable effectiveness in detecting vulnerabilities in longer sequences (over 16K tokens), despite their relative rarity. In BigVul, the model achieves perfect detection (F1=1.000) for sequences in Bin 3. For Bin 2 sequences, it maintains strong performance (F1=0.800) despite the increased complexity. This effectiveness on longer sequences is particularly noteworthy given the increased difficulty of maintaining coherent attention over such distances.

The infini-attention variant achieves comparable performance to full self-attention, with nearly identical metrics across major datasets (e.g., BigVul: 0.937 vs 0.943 F1). It maintains high precision while reducing computational complexity and demonstrates that linear attention is a viable alternative for vulnerability detection. In Bin 1, where most vulnerabilities occur, the model shows exceptional performance, with BigVul reaching F1=0.943, Accuracy=0.993 and VulDeePecker achieving F1=0.925, Accuracy=0.989. The model maintains effectiveness despite significant class imbalance, successfully detecting vulnerabilities even when they comprise only 2.74% of samples (PRIMEVUL) and maintaining balanced precision-recall trade-offs across length bins. Performance variations across datasets reveal interesting patterns. Stronger performance on BigVul and VulDeePecker suggests better handling of certain vulnerability types, while lower scores on PRIMEVUL indicate the challenge of detecting more subtle or complex vulnerabilities.

4.4.4 Comparative Analysis with Standard Attention

When comparing Infini-attention with standard attention (where possible), we observe. Infini-attention achieves comparable or better performance while using significantly less memory. Unlike standard attention, Infini-attention can process the full range of sequence lengths present in real codebases. The ability to handle longer sequences without performance degradation makes White-Basilisk suitable for analyzing entire codebases in a single pass.

4.4.5 Implications for Vulnerability Detection

This ablation study yields several important insights. The success of Infini-attention in maintaining high performance across sequence lengths validates our architectural choices. The model's ability to handle sequences up to 131K tokens while maintaining accuracy demonstrates its practical utility for real-world applications. Strong performance on longer sequences suggests effective capture of long-range dependencies, crucial for detecting vulnerabilities that span multiple functions or files. The memory efficiency of our approach makes it feasible to deploy White-Basilisk on standard hardware, even for processing very long sequences.

These findings confirm that White-Basilisk successfully addresses the key challenges in vulnerability detection: maintaining high accuracy across varying sequence lengths while remaining computationally efficient. The model's

particular strength in handling long sequences, combined with its consistent performance on more common shorter sequences, makes it a practical and effective tool for real-world code security applications.

4.5 Ablation Study: CWE-Specific Performance Analysis

To provide deeper insights into White-Basilisk's vulnerability detection capabilities, we conducted a comprehensive analysis of its performance across different Common Weakness Enumeration (CWE) categories. This analysis focuses on the BigVul, Vuldeepecker and Draper dataset, which provide detailed CWE-level metrics, allowing us to understand the model's strengths and limitations across various vulnerability types.

4.5.1 Experimental Setup

- **Model**: We used the White-Basilisk checkpoint that was trained on the combined datasets from 4.3.
- **Evaluation Splits**: Validation split of each dataset

Dataset-Specific Performance Patterns

Draper Dataset Performance In the Draper dataset (Table 4.8), we observe a consistent pattern of high precision (1.000) across all CWE categories, but with varying recall rates. CWE-119 (Buffer Overflow) achieves the highest recall (0.597) and F1 score (0.748). CWE-120 (Buffer Copy without Checking Size) shows similar performance (recall=0.592, F1=0.744). CWE-469 and CWE-476 demonstrate progressively lower recall (0.563 and 0.522 respectively). This pattern suggests that while the model is highly precise in its predictions, it exhibits some conservatism in vulnerability detection, particularly for less frequent vulnerability types.

VulDeePecker Dataset Analysis The VulDeePecker results (Table 4.9) show more balanced precision-recall characteristics. CWE-119 demonstrates near-perfect balance (precision=0.939, recall=0.940). CWE-399 (Resource Management Errors) shows lower but consistent performance (precision=0.776, recall=0.785). The balanced metrics suggest more robust learning of these vulnerability patterns, possibly due to better representation in the training data.

BigVul Dataset Insights The BigVul dataset (Table 4.10) provides the most comprehensive view of White-Basilisk's capabilities across 50+ CWE categories. Several significant patterns emerge.

Regarding perfect detection cases, 22 CWE categories achieve perfect scores (F1=1.000). These include critical vulnerabilities such as CWE-787 (Out-of-bounds Write) and CWE-310 (Cryptographic Issues); access control issues including CWE-732 (Permission Assignment) and CWE-284 (Access Control); and various severity levels ranging from CWE-59 (Link Following) to CWE-617 (Reachable Assertion). It is notable that perfect detection spans both frequent (over 200 samples) and rare (under 50 samples) categories.

For high-volume vulnerability performance, CWE-119 (2,746 samples) shows excellent performance (F1=0.969), CWE-264 (1,240 samples) demonstrates strong results (F1=0.925), and CWE-20 (1,977 samples) exhibits robust detection (F1=0.933).

Performance degradation patterns are observed in certain vulnerability types. Resource-related vulnerabilities show more variable performance, with CWE-404 (Resource Shutdown) having the lowest F1 score (0.571) and CWE-772 (Missing Release) demonstrating lower precision (0.750). Format-string vulnerabilities (CWE-134) show precision-recall imbalance (0.500/1.000).

The impact of sample size reveals that large sample categories (over 1000 samples) show consistently strong but not perfect performance. Medium-sized categories (100-1000 samples) demonstrate more variable results. Small categories (under 100 samples) often show perfect or near-perfect scores, suggesting potential overfitting.

Cross-Dataset Performance Analysis

The model's behavior across datasets reveals important patterns. Regarding CWE-119 consistency, as the only vulnerability type present across all three datasets, it shows interesting variation: BigVul with F1=0.969 (balanced precision-recall), VulDeePecker with F1=0.940 (balanced precision-recall), and Draper with F1=0.748 (high precision, lower recall). This variation suggests dataset-specific characteristics affect detection performance. For scale effects, larger datasets (BigVul) generally show more balanced precision-recall trade-offs compared to smaller datasets.

Implications and Insights

These results yield several important insights for vulnerability detection. Regarding architecture effectiveness, White-Basilisk's strong performance across numerous CWE categories validates its hybrid architecture design for vulnerability detection.

Detection patterns indicate that memory-related vulnerabilities consistently show strong detection rates, resource management vulnerabilities present more challenges, and access control vulnerabilities demonstrate surprisingly robust detection.

Practical implications of these findings include that high precision across most categories suggests low false positive rates, variable recall in some categories indicates potential for missed vulnerabilities, and perfect detection in rare categories warrants further investigation for potential overfitting.

These findings demonstrate White-Basilisk's strong general capability while highlighting specific areas for potential improvement. The comprehensive nature of these results, particularly in the BigVul dataset, provides strong evidence for the model's practical utility in real-world vulnerability detection scenarios.

CWE	Description	BigVul		Draper		VulDeePecker	
		Samples	F1	Samples	F1	Samples	F1
CWE-119	Buffer Overflow: Classic buffer overflow vulnerabilities	2,746	0.969	2,419	0.748	10,419	0.940
CWE-399	Resource Management Errors: Failures in managing system resources	1,435	0.923	-	-	5,596	0.780
CWE-20	Input Validation: Improper input validation	1,977	0.933	-	-	-	-
CWE-264	Access Control: Permissions, privileges, and access controls	1,240	0.925	-	-	-	-
CWE-120	Buffer Copy: Buffer copy without checking size of input	-	-	4,750	0.744	-	-
CWE-476	NULL Pointer Dereference	501	0.971	1,208	0.686	-	-
CWE-416	Use After Free: Using memory after it has been freed	963	0.958	-	-	-	-
CWE-200	Information Exposure: Exposure of sensitive information	883	0.944	-	-	-	-

TABLE 4.7: Comparison of Most Frequent CWEs Across Datasets

CWE	Precision	Recall	F1	Accuracy	Total	Pos. Ratio	Neg. Ratio
CWE-119	1.000	0.597	0.748	0.597	2,419	1.000	0.000
CWE-120	1.000	0.592	0.744	0.592	4,750	1.000	0.000
CWE-469	1.000	0.563	0.721	0.563	252	1.000	0.000
CWE-476	1.000	0.522	0.686	0.522	1,208	1.000	0.000
CWE-other	1.000	0.472	0.642	0.472	3,579	1.000	0.000
Overall	0.609	0.532	0.568	0.948	127,476	0.065	0.935

TABLE 4.8: Draper Dataset Metrics for All CWEs

CWE	Precision	Recall	F1	Accuracy	Total	Pos. Ratio	Neg. Ratio
CWE-119	0.939	0.940	0.940	0.991	10,419	0.077	0.923
CWE-399	0.776	0.785	0.780	0.986	5,596	0.031	0.969
Overall	0.910	0.913	0.911	0.989	16,015	0.061	0.939

TABLE 4.9: VulDeePecker Dataset Metrics for All CWEs

CWE	Precision	Recall	F1	Accuracy	Total	Pos. %	Neg. %
CWE-787	1.000	1.000	1.000	1.000	291	6.19	93.81
CWE-119	0.978	0.960	0.969	0.995	2746	8.27	91.73
CWE-125	0.984	0.984	0.984	0.997	794	7.68	92.32
CWE-264	0.925	0.925	0.925	0.994	1240	4.27	95.73
CWE-416	0.944	0.971	0.958	0.997	963	3.63	96.37
CWE-476	0.944	1.000	0.971	0.998	501	3.39	96.61
CWE-200	0.913	0.977	0.944	0.994	883	4.87	95.13
CWE-189	0.921	0.946	0.933	0.993	695	5.32	94.68
CWE-732	1.000	1.000	1.000	1.000	143	3.50	96.50
CWE-311	1.000	0.667	0.800	0.963	27	11.11	88.89
CWE-772	0.750	1.000	0.857	0.983	116	5.17	94.83
CWE-399	0.957	0.892	0.923	0.992	1435	5.16	94.84
CWE-20	0.905	0.963	0.933	0.992	1977	5.51	94.49
CWE-190	0.960	0.889	0.923	0.989	378	7.14	92.86
CWE-59	1.000	1.000	1.000	1.000	96	2.08	97.92
CWE-362	0.939	0.861	0.899	0.988	592	6.08	93.92
CWE-400	1.000	0.800	0.889	0.993	136	3.68	96.32
CWE-310	1.000	1.000	1.000	1.000	148	4.05	95.95
CWE-754	1.000	1.000	1.000	1.000	32	3.13	96.87
CWE-835	1.000	0.750	0.857	0.988	86	4.65	95.35
CWE-284	0.944	1.000	0.971	0.996	232	7.33	92.67
CWE-358	1.000	1.000	1.000	1.000	15	33.33	66.67
CWE-388	1.000	1.000	1.000	1.000	36	8.33	91.67
CWE-22	1.000	1.000	1.000	1.000	74	4.05	95.95
CWE-704	1.000	1.000	1.000	1.000	80	1.25	98.75
CWE-254	1.000	0.933	0.966	0.997	302	4.97	95.03
CWE-415	1.000	1.000	1.000	1.000	102	13.73	86.27
CWE-369	1.000	1.000	1.000	1.000	78	6.41	93.59
CWE-79	1.000	1.000	1.000	1.000	84	3.57	96.43
CWE-404	0.500	0.667	0.571	0.963	81	3.70	96.30
CWE-134	0.500	1.000	0.667	0.983	60	1.67	98.33
CWE-346	1.000	1.000	1.000	1.000	6	16.67	83.33
CWE-17	1.000	1.000	1.000	1.000	72	4.17	95.83
CWE-77	1.000	1.000	1.000	1.000	22	9.09	90.91
CWE-269	1.000	0.667	0.800	0.988	86	3.49	96.51
CWE-611	1.000	1.000	1.000	1.000	46	6.52	93.48
CWE-19	1.000	0.714	0.833	0.973	73	9.59	90.41
CWE-617	1.000	1.000	1.000	1.000	99	4.04	95.96
CWE-494	1.000	1.000	1.000	1.000	7	14.29	85.71
CWE-287	1.000	1.000	1.000	1.000	59	6.78	93.22
CWE-834	1.000	1.000	1.000	1.000	40	5.00	95.00
CWE-665	1.000	1.000	1.000	1.000	8	37.50	62.50
CWE-674	1.000	1.000	1.000	1.000	4	25.00	75.00
CWE-668	1.000	1.000	1.000	1.000	7	14.29	85.71
CWE-918	1.000	1.000	1.000	1.000	5	20.00	80.00
CWE-682	0.750	1.000	0.857	0.900	10	30.00	70.00
CWE-191	1.000	1.000	1.000	1.000	10	10.00	90.00
CWE-18	1.000	1.000	1.000	1.000	5	80.00	20.00
CWE-16	1.000	1.000	1.000	1.000	7	14.29	85.71
CWE-824	1.000	1.000	1.000	1.000	1	100.00	0.00
Overall	0.946	0.940	0.943	0.993	18864	5.88	94.12

TABLE 4.10: Complete BigVul Metrics for All CWEs with Class Balance

CWE	Accuracy	Total	Pos. %	Neg. %
CWE-601	0.875	8	0.00	100.00
CWE-347	1.000	3	0.00	100.00
CWE-426	1.000	3	0.00	100.00
CWE-361	1.000	4	0.00	100.00
CWE-285	1.000	25	0.00	100.00
CWE-290	1.000	8	0.00	100.00
CWE-94	0.875	8	12.50	87.50
CWE-281	1.000	5	0.00	100.00
CWE-706	1.000	1	0.00	100.00
CWE-862	1.000	2	0.00	100.00
CWE-693	1.000	7	0.00	100.00
CWE-295	1.000	7	0.00	100.00
CWE-1021	1.000	5	0.00	100.00
CWE-255	1.000	4	0.00	100.00
CWE-129	1.000	7	0.00	100.00
CWE-120	1.000	8	0.00	100.00
CWE-352	1.000	4	0.00	100.00
CWE-327	1.000	1	0.00	100.00
CWE-909	1.000	7	0.00	100.00
CWE-74	1.000	1	0.00	100.00
CWE-330	1.000	3	0.00	100.00
CWE-90	1.000	4	0.00	100.00
CWE-770	1.000	6	0.00	100.00
CWE-172	1.000	1	0.00	100.00
CWE-354	1.000	1	0.00	100.00
CWE-502	1.000	2	0.00	100.00
CWE-755	1.000	2	0.00	100.00
CWE-664	1.000	2	0.00	100.00

NOTE: Some metrics not applicable due to single-class predictions

TABLE 4.11: Single Class CWE Results

Chapter 5

Conclusions and Future Work

5.1 Limitations and Future Work

While White-Basilisk shows promising results in code vulnerability detection, it is important to acknowledge its current limitations and outline future directions. The main limitation of White-Basilisk is its focus on C and C++ codebases. The model's ability to generalize across a broader range of programming languages, especially those with different syntaxes or paradigms, warrants further exploration. This constraint, combined with potential biases in our training and evaluation datasets, may limit the model's generalizability to diverse real-world codebases. To address this, future work will involve expanding the model's training to include a wider variety of programming languages and curating more representative datasets that reflect a broader spectrum of code samples and vulnerability types.

Also, despite strong performance metrics, White-Basilisk is not infallible. False positives and false negatives, particularly when detecting novel or zero-day vulnerabilities, remain an ongoing challenge. Additionally, while the model is capable of processing long sequences, its true understanding of complex, long-range dependencies in code still needs further investigation. Our future research will focus on reducing error rates, especially in high-stakes scenarios, and enhancing the model's ability to analyze convoluted code structures spanning multiple functions or files.

Another area for improvement is the model's explainability. Currently, White-Basilisk's decision-making process is not sufficiently transparent. Improving the model's ability to provide clear, actionable explanations for detected vulnerabilities is essential for building trust and delivering meaningful insights to developers. Future work will explore methods to offer context-aware understanding of the potential impact of vulnerabilities, as well as suggested

fixes, ultimately aiming to evolve White-Basilisk into a comprehensive code analysis assistant rather than a mere detection tool.

Also, while more efficient than many larger models, White-Basilisk still requires significant computational resources, particularly when processing very long sequences. This may limit its accessibility for smaller organizations or individual developers. Our ongoing research will retain its focus on optimizing further the model architecture to maintain or improve its long-context processing capabilities, while reducing computational demands.

White-Basilisk's performance on a relatively small training dataset (2M samples) is impressive, but raises questions about potential limitations in its knowledge base compared to models trained on much larger datasets. To address this, we plan to scale White-Basilisk to approximately 1 billion parameters. While still modest in comparison to larger language models, this increase in parameter count aims to significantly boost performance while continuing to challenge the notion that only the largest models can deliver state-of-the-art results.

Another area requiring further investigation is improving the model's robustness against adversarial attacks, specifically designed for code analysis models. Despite our use of SIFT, further testing and development of more advanced adversarial training techniques tailored for code vulnerability detection are necessary, for ensuring reliability in hostile real-world environments.

Beyond code vulnerability detection, there is potential for White-Basilisk's architecture and training approach in broader AI applications. Future research will investigate its efficacy in various NLP tasks, exploring whether its computational efficiency and long-context capabilities can offer more resource-efficient alternatives to existing LLMs.

5.2 Conclusion

White-Basilisk represents a significant advancement in the domain of code vulnerability detection, offering a novel solution to the persistent challenge of context handling in Transformer-based Large Language Models (LLMs). With its capacity to process sequences up to 128,000 tokens, White-Basilisk introduces unprecedented possibilities for comprehensive code analysis, potentially revolutionizing approaches to software security.

The model's extended context window addresses a fundamental limitation of many current LLMs, which often struggle with long-range dependencies and global code structure understanding. By enabling the analysis of entire codebases in a single pass, White-Basilisk can capture complex interdependencies and identify vulnerabilities that span multiple functions or files, a capability that has long eluded traditional approaches.

While the context-handling capabilities of White-Basilisk are its standout feature, it's worth noting that these achievements have been realized with a relatively compact model of 200M parameters. This efficiency demonstrates that advances in AI are not solely dependent on increasing model size, but can also stem from innovative architecture design and training methodologies.

The implications of White-Basilisk's approach extend beyond code vulnerability detection. The ability to handle extended contexts efficiently could prove valuable in numerous domains where long-range understanding is crucial, such as document analysis, complex system modeling, or long-form text generation. Moreover, the model's efficiency opens up possibilities for deployment in resource-constrained environments, potentially bringing advanced AI capabilities to a broader range of applications and users.

In conclusion, White-Basilisk represents a significant step forward in addressing the context limitations of current LLMs, while also demonstrating that such advances need not come at the cost of excessive model size or computational requirements. As we continue to refine and expand upon this approach, we anticipate exciting developments in the field of AI, particularly in tasks that require deep understanding of extended contexts. The potential implications of this research are substantial, and we look forward to seeing how these ideas evolve and find application in diverse areas of AI and beyond.

References

[1] David Patterson et al. “Carbon emissions and large neural network training”. In: *arXiv preprint arXiv:2104.10350* (2021).

[2] Ahmad Faiz et al. “Llmcarbon: Modeling the end-to-end carbon footprint of large language models”. In: *arXiv preprint arXiv:2309.14393* (2023).

[3] Xin Zhou et al. “Comparison of Static Application Security Testing Tools and Large Language Models for Repo-level Vulnerability Detection”. In: *arXiv preprint arXiv:2407.16235* (2024).

[4] Yangruibo Ding et al. “Vulnerability detection with code language models: How far are we?” In: *arXiv preprint arXiv:2403.18624* (2024).

[5] Jiahao Fan et al. “AC/C++ code vulnerability dataset with code changes and CVE summaries”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 508–512.

[6] Rebecca Russell et al. “Automated vulnerability detection in source code using deep representation learning”. In: *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE. 2018, pp. 757–762.

[7] Saikat Chakraborty et al. “Deep learning based vulnerability detection: Are we there yet?” In: *IEEE Transactions on Software Engineering* 48.9 (2021), pp. 3280–3296.

[8] Zhen Li et al. “Vuldeepecker: A deep learning-based system for vulnerability detection”. In: *arXiv preprint arXiv:1801.01681* (2018).

[9] Brian Chess and Gary McGraw. “Static analysis for security”. In: *IEEE security & privacy* 2.6 (2004), pp. 76–79.

[10] V Benjamin Livshits and Monica S Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis.” In: *USENIX security symposium*. Vol. 14. 2005, pp. 18–18.

[11] Foteini Cheirdari and George Karabatis. “Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools”. In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 4782–4788. DOI: 10.1109/BigData.2018.8622456.

[12] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1032–1043. ISBN: 9781450341394. DOI: 10.1145/2976749.2978428. URL: https://doi.org/10.1145/2976749.2978428.

[13] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046.

[14] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)". In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[16] Nick Stephens et al. "Driller: Augmenting fuzzing through selective symbolic execution." In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[17] Konstantin Serebryany et al. "AddressSanitizer: a fast address sanity checker". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 28.

[18] Sanjay Rawat et al. "Vuzzer: Application-aware evolutionary fuzzing". In: *2017 Network and Distributed System Security (NDSS) Symposium:[Proceedings]*. Internet Society. 2017, pp. 1–14.

[19] Zhen Li et al. "VulPecker: an automated vulnerability detection system based on code similarity analysis". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACSAC '16. Los Angeles, California, USA: Association for Computing Machinery, 2016, pp. 201–213. ISBN: 9781450347716. DOI: 10.1145/2991079.2991102. URL: https://doi.org/10.1145/2991079.2991102.

[20] Gustavo Grieco et al. "Toward Large-Scale Vulnerability Discovery using Machine Learning". In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. CODASPY '16. New Orleans, Louisiana, USA: Association for Computing Machinery, 2016, pp. 85–

96. ISBN: 9781450339353. DOI: 10.1145/2857705.2857720. URL: https://doi.org/10.1145/2857705.2857720.

[21] Yaqin Zhou et al. "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks". In: *Advances in neural information processing systems* 32 (2019).

[22] Hazim Hanif and Sergio Maffeis. "Vulberta: Simplified source code pre-training for vulnerability detection". In: *2022 International joint conference on neural networks (IJCNN)*. IEEE. 2022, pp. 1–8.

[23] Michael Fu and Chakkrit Tantithamthavorn. "Linevul: A transformer-based line-level vulnerability prediction". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 608–620.

[24] Albert Gu and Tri Dao. "Mamba: Linear-time sequence modeling with selective state spaces". In: *arXiv preprint arXiv:2312.00752* (2023).

[25] Sinong Wang et al. "Linformer: Self-attention with linear complexity". In: *arXiv preprint arXiv:2006.04768* (2020).

[26] Zhangyin Feng et al. "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155* (2020).

[27] Daya Guo et al. "UniXcoder: Unified Cross-Modal Pre-training for Code Representation". In: *arXiv preprint arXiv:2203.03850* (2022).

[28] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[29] Raymond Li et al. "Starcoder: may the source be with you!" In: *arXiv preprint arXiv:2305.06161* (2023).

[30] Wasi Uddin Ahmad et al. "Unified pre-training for program understanding and generation". In: *arXiv preprint arXiv:2103.06333* (2021).

[31] Opher Lieber et al. "Jamba: A hybrid transformer-mamba language model". In: *arXiv preprint arXiv:2403.19887* (2024).

[32] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. "Leave no context behind: Efficient infinite context transformers with infini-attention". In: *arXiv preprint arXiv:2404.07143* (2024).

[33] William Fedus, Barret Zoph, and Noam Shazeer. "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity". In: *Journal of Machine Learning Research* 23.120 (2022), pp. 1–39.

[34] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[35] Zihang Dai et al. "Transformer-xl: Attentive language models beyond a fixed-length context". In: *arXiv preprint arXiv:1901.02860* (2019).

[36] Tri Dao et al. “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2022.

[37] Noam Shazeer et al. “Outrageously large neural networks: The sparsely-gated mixture-of-experts layer”. In: *arXiv preprint arXiv:1701.06538* (2017).

[38] Haoming Jiang et al. “SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models through Principled Regularized Optimization”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics* (2020). DOI: 10.18653/v1/2020.acl-main.197. URL: http://dx.doi.org/10.18653/v1/2020.acl-main.197.

[39] Yue Wang et al. “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation”. In: *EMNLP*. Association for Computational Linguistics, 2021, pp. 8696–8708.

[40] Erik Nijkamp et al. “CodeGen2: Lessons for Training LLMs on Programming and Natural Languages”. In: *arXiv preprint* (2023).

[41] Alexandre Lacoste et al. “Quantifying the Carbon Emissions of Machine Learning”. In: *arXiv preprint arXiv:1910.09700* (2019).