UCLouvain

École polytechnique de Louvain

PQMap: Identification and Mapping of Post-Quantum-Relevant Cryptographic Functions in Binaries

An Embedding-Based Semantic Retrieval Tool for Binary Cryptographic Code

Author: **Vinícius DE PAULA**
Supervisor: **Ramin SADRE**
Readers: **Marcos SIMPLÍCIO JÚNIOR, Bastien WIAUX**
Academic year 2025–2026
Master [120] in Computer Science and Engineering

Abstract

The transition to post-quantum cryptography requires identifying where classical public-key cryptography is used in existing software. This task is especially difficult for compiled binaries, where source code may be unavailable and function symbols may be incomplete, stripped, or inconsistent across library versions. This work presents `PQMap`, an embedding-based semantic retrieval tool for mapping post-quantum migration risk in binary cryptographic code.

PQMap analyzes binaries with Ghidra, extracts decompiled pseudo-code for individual functions, and embeds each function using a fine-tuned code embedding model. The model is trained with contrastive function pairs built from trusted canonical identifiers across multiple versions and libraries. PQMap embeds a function's pseudo-code and retrieves semantically similar functions from a labeled reference corpus. The retrieved neighbors are then used to support cryptographic function identification and consequently post-quantum migration analysis.

On a manually reviewed benchmark, PQMap achieved 95.0% Precision@1 and 95.5% mean reciprocal rank over 1,000 retrieval queries. These results show that decompiler-based semantic embeddings can support practical discovery and prioritization of cryptographic functions relevant to post-quantum migration in compiled binaries.

The implementation and supporting scripts are available in the tool's repository [1].

Acknowledgments

I would like to thank my supervisor, Ramin Sadre, for the guidance, feedback, and critical questions that helped shape this work. The discussions throughout the project were especially important in refining the evaluation methodology, clarifying the role of symbol names, and strengthening the experimental design.

I am also grateful to my home university supervisor, Marcos Simplício Júnior, for his support and availability for discussions, constant suggestions and availability, and for serving as a reader for this thesis. Furthermore, I extend my thanks to Bastien Wiaux, who also served as a reader and provided valuable suggestions that directly improved the evaluation of the tool.

Finally, I would like to thank my family and friends for their support and encouragement during the development of this thesis.

Declaration of AI use

During the preparation of this master's thesis, I used generative AI tools to assist with specific language refinement and supplementary coding tasks. For language and phrasing, I used Gemini 3.1 Pro, the Writefull model integrated in Overleaf and Piccolo, strictly to improve the wording, flow, and clarity of the drafts. In addition, I used GPT-5 to assist in generating specific auxiliary scripts. This was limited to writing the Java support file for the Ghidra Headless Analyzer, and the Python files related to building the dataset, the inference and the evaluation steps of the tool.

In contrast, generative AI was explicitly not used for the following: it was not used to write the core text or generate any figures (all of which are of my own authorship); it was not used for research and analysis, including data collection, experimental design, or data analysis; and it was not used to develop the core tool code of this thesis, which consists of my own original authorship and manual modifications to the training components of the project that inspired this research.

Finally, I have manually reviewed, tested, and evaluated all AI-generated code and text suggestions. The AI served merely as a supporting tool that allowed me to focus on the primary objectives of the research. I take full academic responsibility for the accuracy, integrity, and final content of this thesis.

Contents

Chapter 1

Introduction

1.1 Motivation and Research Context

Quantum computing is a rapidly expanding field with both theoretical and experimental advances. A recent bibliometric analysis [2] revealed exponential growth in quantum computing as a research domain, expanding from foundational algorithms to diverse applied topics including quantum cryptography, hybrid algorithms, and hardware development. In that sense, document-level citation analysis of publications from 2020 to 2025 highlights the emergence of new thematic frontiers and demonstrates intensified research activity in the field [2].

In 2023, IBM Quantum has unveiled the first quantum processor with more than 1,000 qubits (specifically, 1,121 qubits) [3, 4], which is still far from sufficient for practical use. The company's focus then changed to developing more reliable chips, reducing the error rate [3]. Similarly, in 2025, Microsoft reported experimental progress toward a topological qubit architecture based on Majorana modes [4, 5], which pivots towards enabling intrinsically protected qubits, whose physical design is intended to make them less sensitive to certain types of noise [6], thus contributing to the reduction of error rates. Despite current limitations in qubit scalability and fault-tolerant error correction, sustained experimental progress and intensified research efforts suggest that quantum computing may become practically viable in the foreseeable future.

Beyond exploiting quantum mechanical principles to address computational problems intractable for classical computers [7], quantum computing also poses significant implications for modern cryptography. In particular, large-scale fault-tolerant quantum computers would be capable of efficiently solving the integer factorization problem using quantum algorithms, thereby compromising widely deployed public-key cryptosystems such as RSA [8]. More broadly, quantum computers will undermine the security of current standard public-key cryptographic

algorithms [9].

Consequently, due to the forthcoming compromise of widely used public-key algorithms and to the technical and logistical challenges of replacing them, initiatives involving the public and private sectors have focused on developing demonstrations and tools that can aid organizations in migrating to quantum-resistant alternatives. In particular, significant attention has been directed toward tools that can pinpoint the usage of public-key cryptographic algorithms [9, 10].

1.2 Problem Statement

In light of the concerns previously outlined, tools that can identify instances of quantum-vulnerable public-key algorithms, widely used throughout an organization, will assist in creating a cryptographic algorithm inventory. This inventory will support the development of a migration plan. Furthermore, although the advent of quantum computing poses significant threats to current public-key algorithms, the development of tools to facilitate the migration to quantum-resistant alternatives is still in its early stages. Despite some alternatives that have been proposed, there is a clear need for more robust automatic tools specifically designed to assist in this crucial transition.

Moreover, binaries for which there might not be a source code, which is often the case for third-party applications, render the capability of binary scanning essential for such automated discovery tools. Hence, motivated by these challenges, this master thesis aims at developing an automated cryptographic algorithm identification tool over binary executables, specifically pivoting towards quantum-vulnerable cryptographic algorithm identification.

1.3 Research Objectives and Contributions

To achieve the aforementioned objectives, this master thesis proposes the design of an ML-based automated identification workflow that leverages transfer learning by utilizing a high-quality code embedding model as a backbone and further adapts to the cryptographic identification problem. This workflow aims to automate the process of identifying quantum-vulnerable cryptographic algorithms, reducing the need for manual intervention and increasing efficiency.

Additionally, in alignment with the requirement for binary scanning capabilities, the proposed tool uses the reverse engineering suite Ghidra [11] to enable the identification of relevant code structures in binaries. The usage of Ghidra Headless Analyzer renders the automation of the decompilation process possible, and the encoding process ensures compatibility with various compilation formats.

Given the vast number of binary executables used by an enterprise, it is crucial that such cryptographic discovery tools aim to reduce the analyst's effort when identifying quantum-vulnerable cryptographic usage in their inventory. The proposed solution will automate repetitive tasks, such as scanning and initial analysis, freeing up analysts to focus on more complex and critical aspects of the security assessment.

Finally, the performance of the tool is evaluated, providing a comprehensive assessment of its effectiveness in real-world scenarios and its ability to scale with increasing data volumes.

In view of the above, this master thesis aims to make the following contributions:

- Introduction of a toolchain designed to ease the cryptographic migration process: An automated toolchain designed to facilitate the shift to secure alternatives by detecting, locating, and mapping instances of quantum-vulnerable algorithms within software executables.
- Validation of the accuracy and efficiency of the tool developed using real-world and synthetic datasets: Rigorous testing and validation of the tool using a variety of datasets to ensure its accuracy, efficiency, and reliability in identifying quantum-vulnerable cryptographic usage.

1.4 Structure of this Report

This report is organized as follows:

- **Section 2** provides an overview of the existing research in the fields of cryptographic discovery and binary code similarity. It includes a detailed examination of recent publications in post-quantum cryptography, systematization of cryptographic identification, and the current state-of-the-art tools used in these domains.
- **Section 3** delves into the core concepts that underpin the development of the proposed tool, introducing reverse engineering and the Ghidra suite, explaining its role and functionalities in the context of binary code analysis. In addition, it covers essential machine learning concepts, including transfer learning and code embeddings.
- **Section 4** outlines the design principles and architectural choices made during the development of the tool. It includes detailed descriptions of the system architecture, the integration of Ghidra Headless and Jina, and the overall design.

- **Section 5** discusses the practical implementation of the tool, covering the technical details of the implementation process, including the algorithms used and the data processing techniques employed.
- **Section 6** presents the experimental setup used to evaluate the tool, including the datasets, the evaluation metrics, and the methodology used. Then, it discusses the results obtained, providing analysis and interpretation of the findings.
- **Section 7** acknowledges the limitations of the current work and discusses potential avenues for future research, highlighting the challenges encountered during the development process.
- Finally, **Section 8** summarizes the key findings and contributions of this thesis.

Chapter 2

Related Work

2.1 Cryptographic Discovery Research

In this section, the related work on the general problem of cryptographic identification is discussed, along with the specific problem addressed in this thesis: the discovery of the usage of cryptographic algorithms that are vulnerable to post-quantum cryptography (PQC).

As previously outlined, the rapid advancement of quantum computing poses a significant threat to traditional cryptographic algorithms, making the migration to post-quantum cryptography (PQC) a priority. In this perspective, it is important to understand the PQC context. In addition, this section also considers broad surveys that pivot towards understanding the current landscape of cryptographic identification.

2.1.1 Post-Quantum Cryptography Context

This aforementioned transition was formally presented with the NIST PQC standards, FIPS 203, 204, and 205 [12, 13, 14], which define algorithms for key encapsulation and digital signature, as shown in Table 2.1. Nevertheless, adapting software ecosystems to these new standards is neither a trivial nor immediate step. As outlined by Newhouse et al. [9, 10], organizations must integrate quantum readiness - which is the process of updating cryptographic systems and security protocols to withstand future, more powerful quantum computers - into their broader risk management and compliance architectures.

To facilitate this complex transition, researchers began to develop structured methodologies to manage enterprise-scale migrations. For example, Hasan et al. [15] proposed a comprehensive framework using graph-theoretic techniques to identify and prioritize vulnerable cryptographic assets within complex organizational IT infrastructures. Similarly, Zhang [16] recently defined Quantum-Safe Software

Table 2.1: NIST's quantum-safe standards

Specification	Function	Algorithm
FIPS 203	Key establishment algorithm	ML-KEM (Modulo-Lattice-Based Key-Encapsulation Mechanism Standard)
FIPS 204	Digital signature algorithm	ML-DSA (Modulo-Lattice-Based Digital Signature Standard)
FIPS 205	Digital signature algorithm	SLH-DSA (Stateless Hash-Based Digital Signature Standard)

Engineering as a distinct research direction, arguing that the probabilistic behavior, side-channel sensitivities, and resource constraints of PQC algorithms require an entirely new class of automated detection tools.

Together, these works highlight a critical gap between high-level migration policies and the practical tools required to execute them securely, directly motivating the need for automated cryptographic discovery toolchains.

2.1.2 Systematization of Cryptographic Identification

Although the said enterprise structured methodologies instruct the comprehensive discovery of vulnerable cryptographic primitives, pinpointing them within compiled, stripped binary executables remains a major technical challenge. In this sense, a set of factors such as the compilation process, aggressive optimization flags, and varying instruction set architectures frequently obscure the distinct signatures and control-flow semantics inherent to cryptographic implementations.

To navigate this complexity, researchers have systematically mapped the evolution of cryptographic recognition techniques. For instance, Zhao et al. [17] establish a comprehensive taxonomy of these methodologies, tracing the shift from early static signature matching and magic constant heuristics to computationally intensive dynamic approaches like taint analysis and symbolic execution. Building on this foundational categorization, a recent systematization of knowledge [18] evaluates the practical efficacy of these tools. This systematic study reveals a persistent and critical trade-off in the domain, which is that dynamic analysis provides high precision and resilience against obfuscation but suffers from severe scalability bottlenecks, whereas static pattern matching scales effortlessly but is fragile against modern compiler transformations.

2.2 Binary Code Similarity Research

The challenge of comparing compiled binaries is centered on the strip of high-level semantic markers, such as function names and variables, during the compilation

process. As detailed by Haq and Caballero [19], early approaches to this problem relied heavily on structural and syntatic techniques, comparing raw instruction sequences, execution traces, or Control Flow Graphs (CFGs) to find matches. In contrast, changes in compilers, optimization levels, or target architectures drastically modify the resulting binary structure, making purely structural comparisons prone to high false-negative rates.

To address the limitations of strict graph and/or syntax matching, recent research has pivoted towards representation learning. Instead of manually selecting features, these modern mechanics transform binary functions into high-dimensional continuous vectors, or embeddings. The similarity is then calculated using geometric distance, allowing the system to recognize semantic equivalence even when the underlying assembly instructions differ entirely. A prominent advancement in generating these representations is detailed by Kryvosheieva et al. [20], when introducing jina-code-embeddings. Their approach leverages an autoregressive backbone pre-trained on massive datasets of both text and code, utilizing last-token pooling and contrastive learning to generate highly dense, truncatable embeddings. The efficiency and semantic depth of the Jina model make it an ideal backbone for scalable code retrieval tasks, and it serves as the foundational embedding engine for the tool developed in this master thesis.

Shi et al. systematically examine the practical application and limitations of these modern embedding techniques [21], with a large-scale empirical study that evaluates state-of-the-art Binary Function Similarity Detection (BFSD) tools against real-world settings, such as heavy function inlining and cross-architecture compilation. This survey serves as the primary theoretical baseline for the cryptographic discovery tool developed in this research. Specifically, the study highlights the effectiveness of dejina, a tool based on Jina [22]. Dejina employs a robust training methodology to align representations across different compilation states, demonstrating strong performance in evaluations across tasks such as cross-optimization, cross-compiler, cross-bitness, cross-architecture, and mixed combinations thereof. To achieve accurate and architecture-agnostic cryptographic discovery, the tool developed in this thesis leverages two state-of-the-art concepts: it utilizes the powerful embedding generation of jina-code-embeddings as its base architecture and adapts the specialized contrastive training and alignment mechanics of the dejina tool.

2.3 Overview of Existing Tools in Cryptographic Identification

Automated identification of cryptographic primitives within binary executables is an important capability for multiple tasks, namely malware analysis, vulnerability research, and the construction of cryptographic inventories. Since the compilation process tends to obscure high-level algorithms into complex and often highly optimized assembly routines, researchers have developed automated methodologies to detect these implementations. Section 2.3.1 explores multiple approaches that are based on machine learning. After that, a tool specifically developed with the intent of cryptographic discovery is detailed.

2.3.1 ML-based cryptographic identification tools

The application of Machine Learning to binary analysis has provided powerful mechanisms to classify and locate cryptography without relying on manually crafted signatures. Early approaches in this domain demonstrated the viability of using Artificial Neural Networks (ANNs) to distinguish cryptographic routines from standard functional blocks. For example, the FALKE-MC framework proposed by Aigner [23] introduced a modular approach to locate cryptographic functions in machine code by automatically extracting structural features and constants from sample binaries to train a neural network classifier. Although effective as a foundational technique, it relied primarily on shallow network architectures and specific feature extraction.

Building upon these early concepts, researchers began applying more complex deep learning architectures. As presented by Hill and Bellekens [24], Dynamic Convolutional Neural Networks (DCNNs) have been successfully employed to learn from variable-length control flow diagnostics. Using dynamically generated synthetic datasets, their approach demonstrated that deep learning models could accurately classify specific cryptographic primitives, such as AES or RSA, by analyzing the representations of their execution traces.

More recently, the field has seen approaches that rely on treating assembly code as a language, directly borrowing techniques from Natural Language Processing (NLP). The methodology presented by Jia et al. [25] abandons manual feature selection, using, instead, an instruction-to-vector model to translate assembly instructions into continuous vectors, which are then processed by a K-Max-CNN-Attention network, forming a pipeline that starts with the generation of an embedding that will then be used by their neural network. This allows the model to capture the deep, high-level semantics of the code, achieving high accuracy in detecting cryptography in malwares. Furthermore, their structure pipeline served as base for

the development of the tool proposed in this thesis, with distinct approaches to embedding and machine learning model selection.

This NLP-inspired trajectory has culminated in the integration of Large Language Models (LLMs) for binary analysis. The recent FoC framework proposed by Shang et al. [26] is one of the most recent approaches in this field. Their framework uses an LLM-based generative model to summarize the semantics of cryptographic functions in human-readable natural language. Then, it pairs this generative capability with a code similarity detection model to retrieve similar implementations.

2.3.2 Cryptographic Discovery tools

Despite the notable extensive research and development dedicated to general cryptographic identification tools, efforts explicitly focused on cryptographic discovery, or more specifically pinpointing quantum-vulnerable cryptography to facilitate PQC migration, remain scarce.

Currently, the Quantum-vulnerable Executable Detection (QED) toolchain, proposed by Rattanavipanon et al. [27], stands as the only established framework that directly addresses this specific challenge. Designed to assist in evaluating software executables for PQC migration, QED provides a semi-automated deterministic method to isolate QV assets within binaries. Their tool relies on a targeted, three-phase static analysis approach, providing efficiency without compromising accuracy since it initially performs fast analysis to filter out files that are unlikely to be quantum-vulnerable, allowing complex analysis over a smaller set of binaries.

2.4 Comparison with Related Tools

The development of the proposed cryptographic discovery tool combines insights of general cryptographic identification, binary function similarity detection (BFSD), and the specific requirements of PQC migration. By leveraging jina-code-embeddings to translate decompiled pseudocode into high-dimensional vector representations, this tool addresses several limitations present in the current state-of-the-art. As highlighted in recent systematization studies [17, 18], traditional static analysis relies heavily on signature matching and magic constants, which are easily obscured by aggressive compiler optimizations, while dynamic analysis provides precision, but can have scalability issues due to its resource-intensive nature. Using a static representation via decompiled pseudocode, but analyzing it from the perspective of semantic embeddings, the proposed tool achieves the scalability of static analysis while maintaining resilience against agressive structural modifications.

Moreover, while previous machine learning approaches have successfully identified cryptographic primitives, they often rely on simpler, limited and feature-selection-dependent architectures. For instance, unlike early frameworks that rely on manually extracted structural features for shallow ANNs [23], the proposed tool automates feature extraction through modern representation learning to capture deeper semantic meaning. Similarly, although it shares a conceptual pipeline with tools that convert code to vectors for classification [25], it significantly modernizes the approach: instead of using raw assembly instructions and CNN-Attention networks, this tool processes decompiled pseudocode using an autoregressive, transformer-based backbone. This choice inherently captures more abstract logic and aligns with the robust cross-architecture methodologies demonstrated in recent BFSD literature [21]. Finally, regarding the specific domain of PQC migration, the proposed tool offers a more complete alternative to the existing toolchain, QED [27], which stands as the primary resource explicitly targeting quantum-vulnerable cryptography, using a static analysis approach. Although efficient for initial filtering, purely static methods can be brittle in multiple scenarios, such as when heavy obfuscation, function inlining, or cross-compiler variations are present. The proposed tool differentiates itself by applying a geometric similarity search. By converting target binaries into pseudocode embeddings and computing their distance against a curated dataset of known quantum-vulnerable cryptographic implementations, the system identifies vulnerable assets based on semantic equivalence rather than strict structural or syntax matching.

Table 2.2 summarizes the methodologies and limitations of the discussed tools.

Table 2.2: Comparison of Existing Cryptographic Identification Methodologies with Proposed Tool

Approach / Tool	**Primary Methodology**	**Key Limitation(s)**
Traditional Static Analysis [17, 18]	Signature matching and magic constants.	Fragile against modern compiler optimizations and obfuscation.
Dynamic Analysis [17, 18]	Taint analysis and symbolic execution.	Resource-intensive, leading to severe scalability bottlenecks.
Early Machine Learning [23]	Shallow ANNs using manually crafted structural features.	Reliant on manual feature selection and limited network architectures.
Vectorized ML Pipelines [25]	Instruction-to-vector models on raw assembly with CNN-Attention networks.	Processes raw assembly syntax, limiting cross-architecture robustness.
QED Toolchain [27]	Targeted, three-phase static analysis for PQC discovery.	Brittle against obfuscation, function inlining, and compiler variations.
Proposed Tool	**Decompiled pseudocode embeddings (jina) + contrastive training.**	**Overcomes above limitations by evaluating semantic equivalence rather than strict syntax.**

Chapter 3

Background

This background chapter establishes the foundational knowledge required to contextualize the contributions of this thesis. Section 3.1 introduces the theoretical and practical underpinnings of reverse engineering, including its methodologies, challenges, and relevance to modern software analysis. Section 3.2 narrows the focus to Ghidra, a state-of-the-art reverse engineering framework, detailing its architecture, capabilities, and limitations to justify its selection as the primary tool for this work. Finally, Section 3.3 explores machine learning concepts — such as transfer learning and code embeddings — critical for automating or enhancing reverse engineering tasks. Together, these sections provide the necessary groundwork to understand the problem space, the tools employed, and the approaches proposed in subsequent chapters.

3.1 Reverse Engineering: Foundations, Applications and Limitations

3.1.1 Definitions and Brief Historical Introduction

Historically, the concept of Reverse Engineering originated in the hardware industry [28]. It involved carefully analyzing and even dismantling physical products to deduce their design, understand their operation, with the goal of replicating it or making improvements. As software systems grew in size and complexity, the software engineering community recognized the need for similar techniques to understand, maintain, and evolve their codebases.

Chikofsky and Cross [28] introduced a detailed taxonomy in 1990 to clarify the terminology in the field. In that sense, they defined key terms, specifically defining reverse engineering as *the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of*

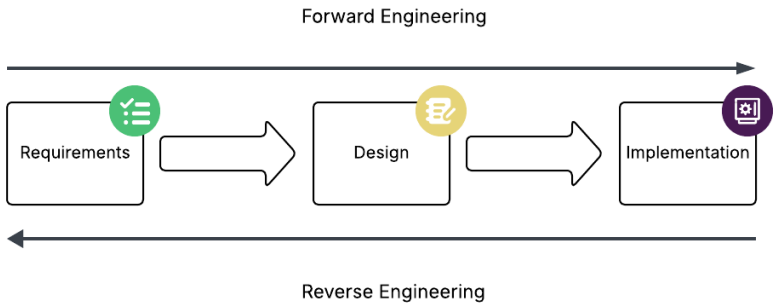

Figure 3.1: Forward Engineering and Reverse Engineering

the system in another form or at a higher level of abstraction.

Crucially, their work established that reverse engineering is strictly a process of analysis, not modification. It is the act of discovering the intention and design of an existing artifact. Hence, reverse engineering simply seeks to move from a lower level of abstraction, such as source code or binaries, to a higher level, such as architectures, diagrams, or the requirements. Therefore, reverse engineering acts in reverse to forward engineering, which is the process of creating a product from concept to design and finally the implementation, as shown in Figure 3.1.

In spite of the fact that the foundational definition remains accurate, both the scope and application of reverse engineering have expanded. Canfora and Di Penta [29] highlighted that the field has evolved well beyond the straightforward extraction of structural information from simpler single-language systems. In that regard, reverse engineering was forced to adapt to new frontiers, characterizing modern systems that are highly dynamic, distributed, and heterogeneous.

Therefore, this evolution highlighted the necessity of changes in design recovery, majorly expanding from static to dynamic techniques and adapting to heterogeneous software systems. Early reverse engineering relied heavily on static analysis, and with the rise of more complex approaches such as object-oriented programming and reflection, static analysis was rendered insufficient on its own, resulting in the importance of dynamic analysis to capture actual system behavior. Moreover, modern software is rarely written in a single programming language. In this respect, they recommended the development of design recovery approaches and tools capable of working with new software architectures that are *extremely dynamic, highly distributed, self-configurable, and heterogeneous* [29].

3.1.2 Practical Applications

In light of the foundations of reverse engineering aforementioned, the most prominent applications of reverse engineering have historically been centered in software engineering and system maintenance. In this context, it is used for design recovery, software maintenance, and migration of legacy systems. In addition, as the software landscape evolved, reverse engineering expanded to address new frontiers, such as understanding web applications and architectures.

Furthermore, reverse engineering has become indispensable in the domain of cybersecurity. For instance, in adversarial environments, analysts rarely have access to source code and, consequently, reverse engineering compiled binaries is crucial for malware analysis, allowing security researchers to analyze malicious payloads and understand attacker behaviors. Additionally, it is highly applied in vulnerability research, where reverse engineering proprietary software or firmware to discover exploitable flaws before they can be leveraged by malicious actors became a common approach.

Within the cybersecurity domain, an advanced and growing application of reverse engineering is in Binary Function Similarity Detection, which involves analyzing stripped, compiled binaries to identify specific functionalities or match unknown code against a database of known components. In this context, one of the specific use cases is the identification of cryptographic algorithms.

3.1.3 Challenges and Limitations

When considering modern software ecosystems and their inherent complexity, reverse engineering generally encounters significant challenges, especially considering static-based reverse engineering tools such as Ghidra, particularly because modern software often relies on dynamic behaviors, rendering static analysis insufficient. Thus, for distributed or even web-based applications, combining static and dynamic analysis is crucial to capture runtime behavior, but at the cost of resource-intensive consumption. Even when structural designs are recovered from source code or binaries, evaluating the quality and structural cohesion of what was extracted remains a challenge [30].

Similarly, obfuscation and other techniques that aim to avoid reverse engineering practices deliberately complicate this extraction process. Finally, the field is also constrained by ethical and legal boundaries. Reverse engineering proprietary software to decipher trade secrets must be done carefully in order to avoid legal consequences [31].

3.2 Ghidra: A Software Reverse Engineering Suite

3.2.1 Tool Introduction and Historical Background

Ghidra is a powerful open source reverse-engineering suite developed by the National Security Agency (NSA) of the United States of America, consolidated as a tool to aid vulnerability uncovering and malicious code analyzing.

Initially disclosed in 2017 via Wikileaks Vault 7, one of the largest leaks of confidential documents related to the Central Intelligence Agency (CIA), Ghidra was officially released during the RSA Conference 2019. After years of internal use, this release can be viewed as a response to the leak. The NSA also made Ghidra's source code available on GitHub in the same year [32].

Then, the context outlined arose suspicions about Ghidra and NSA's intention behind its release, particularly regarding potential hidden backdoors. After an in-depth analysis of the source code by the community, these suspicions were proven unfounded. Consequently, Ghidra is now widely used in the cybersecurity field and is considered a reliable, flexible, and high-performing reverse engineering suite.

Considering the leading software reverse engineering (SRE) tools, it is important to recognize that each framework possesses its own distinct scope, advantages, and limitations. Proprietary solutions, such as Hex-Rays' IDA Pro, have historically dominated the field by offering highly refined capabilities, though these often come with prohibitive licensing costs that limit accessibility. Other open-source frameworks, such as Radare2, provide powerful analytical tools but accommodate to different workflows, often prioritizing command-line resources over graphical interfaces. In this context, Ghidra distinguishes itself as a comprehensive, free, and open-source alternative. Pérez and Tiwari [32] detailed that Ghidra's inherent flexibility, which is demonstrated by its cross-platform compatibility and robust ecosystem for custom scripts and plug-ins, enables practitioners to effectively configure the suite to their specific cybersecurity needs. Ultimately, while analysts may select different tools depending on the precise requirements of a given task, Ghidra's adaptable architecture and cost-free availability have solidified its position as an essential asset in the modern reverse engineering field.

3.2.2 Core Capabilities

Ghidra supports both deep interactive reverse engineering and large-scale automated analysis, and is capable of decompiling binaries of a wide variety of architectures. In this regard, it initially disassembles a given binary using SLEIGH [33], which is a processor specification language and also the associated library and tools for using

such a specification to generate assembly and P-code [34]. Rather than representing source code, P-code acts as an intermediate representation (IR) of the machine code, translating hardware-specific instructions into a standardized, generic set of operations. This architecture-agnostic approach allows Ghidra's decompiler to generate highly readable, C-like pseudocode regardless of the binary's original instruction set.

Alongside its graphical interface, Ghidra's API is accessible and enables deep customization and automation. By using custom Java or Python scripts via this API, analysts can traverse the decompiler's Abstract Syntax Tree (AST) and extract structural features. For instance, this extensibility ensures that the tool can be adapted to identify specific cryptographic signatures, algorithmic structures, or functional artifacts within the code.

To operationalize these capabilities at scale, Ghidra provides a Headless Analyzer, which renders possible the exploitation of robust, automated analysis pipelines. It allows the execution of project operations and custom scripts directly from the command line, eliminating the overhead of the graphical interface. Consequently, this feature is highly valuable for bulk processing, as it eases automated ingestion, auto-analysis, and concurrent decompilation of multiple binaries.

3.3 Machine Learning Concepts

3.3.1 Transfer Learning

Transfer Learning is a technique in machine learning in which knowledge learned from a task is re-used in order to boost performance on a related task, or the same task with a different domain. Thus, this technique allows domains, tasks and distributions used in training and testing to differ. Transfer learning offers major advantages by repurposing pre-trained models, drastically reducing training time, computational costs, and the need for massive datasets. Enhanced performance in new tasks, especially with limited data, is achieved by using learned features. Figure 3.2 shows the learning process for transfer learning.

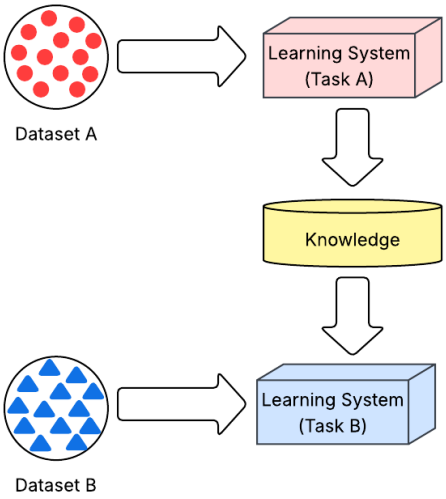

Figure 3.2: Learning Process (Transfer Learning)

Formally, given the source and target domains $\mathcal{D}_S$, $\mathcal{D}_T$ and learning tasks $\mathcal{T}_S$, $\mathcal{T}_T$, the objective of transfer learning is to improve the learning of the target predictive function $f_T(\cdot)$ (i.e., the function, learned from the trained data, that can be used to predict the corresponding label y of a new instance x) in $\mathcal{D}_T$ using the knowledge of $\mathcal{D}_S$ and $\mathcal{T}_S$, where $\mathcal{D}_S \neq \mathcal{D}_T$, or $\mathcal{T}_S \neq \mathcal{T}_T$ [35].

Inductive learning is one of the sub-settings of transfer learning, when the tasks in the source and target domains are different, regardless whether the domain is the same. Furthermore, the presence of labeled data in the target domain is required to induce the objective predictive function.

In that regard, the dejina tool presented by Shi et al. [21], which was built by fine-tuning a jina code embedding model (`jina-embeddings-v2-base-code` [22]), can be seen as using inductive, parameter-based (starting from pretrained Jina weights and afterwards updating parameters using labels specific for Binary Function Similarity Detection) transfer learning with contrastive fine-tuning, since the developed tool uses Jina embeddings as a pretrained encoder, fine-tunes on labeled function similarity pairs and uses a contrastive similarity objective. Inspired by their approach, the tool developed in this master thesis also takes advantage of inductive learning to leverage a high-quality code embedding model as its backbone.

3.3.2 Code Embeddings

Code embeddings are continuous, dense vector representations of source code snippets mapped into a high-dimensional mathematical space. Unlike traditional

discrete representations, such as Abstract Syntax Trees (ASTs), embeddings capture both the syntactic structure and the underlying semantic intent of the code. By projecting the source code into this vector space, machine learning models can compute the semantic distance between different snippets using geometric metrics, such as cosine similarity.

This capability is crucial for automating various complex software engineering tasks, including code retrieval tasks, vulnerability identification, and code clone detection. Transformation of source code into an embedding vector is typically performed by a neural network encoder, which learns to extract structural patterns and semantic meaning from the code during its training phase.

Jina Code Embeddings

The embedding model `jina-code-embeddings` [20] is designed for code retrieval tasks, notably identifying similar code across languages. Moreover, two code embedding models were introduced, differing by the number of parameters: `jina-code-embeddings-0.5b` and `1.5b`, with 494 million and 1.54 billion parameters, respectively, and correspondingly built on `Qwen2.5-Coder-0.5B` and `Qwen2.5-Coder-1.5B` backbones. Both code embedding models employ an autoregressive decoder architecture and generate embeddings via last-token pooling.

Considering the training process for embedding models, the state-of-the-art approach has been to first initialize them from pre-trained backbones, then further pre-train with contrastive learning on curated unsupervised data pairs, i.e. the pairs are created without manually inserted labels but carefully constructed so they are likely to be positive or negative pairs, and finally fine-tune on supervised data for the desired downstream task. Additionally, decoder-only architectures have started outperforming the state-of-the-art bidirectional models [36].

Accordingly, Jina's aforementioned code embeddings leverage existing pre-trained code generation LLMs—specifically, the compact Qwen2.5-Coder autoregressive models—as their architectural backbones. To adapt these generative models for representation tasks, Jina applies last-token pooling to the final hidden layer to generate the embedding vector. The model is then optimized using a contrastive loss function [20]. In that regard, contrastive learning aims to map similar data points closely together in the embedding space while pushing dissimilar points apart. In an unsupervised setting, these pairs are generated heuristically (for example, pairing two variations of the same code) rather than through manual human annotation.

Chapter 4

Design and Architecture

This chapter presents the design and architecture of the developed tool. It first outlines the underlying assumptions in Section 4.1, followed by a comprehensive overview of the system in Section 4.2. Subsequent sections examine the specific strategies that constitute the tool, with Section 4.7 integrating all these elements.

4.1 Approach and Assumptions

For the analysis of a given unknown program, the desired outcome is the automatic detection of all quantum-vulnerable cryptographic usage. Some software applications utilize cryptographic functionalities via API access to cryptographic libraries, such as OpenSSL, which makes verifying the presence of dependencies related to such libraries an immediate approach for cryptographic discovery in these applications. Nevertheless, focusing on identifying such dependencies would not be very effective in all cases, since such security practice is not present in all programs. Similarly, signature matching in static analysis to recognize known cryptographic functions would also result in a limited alternative. In this research, the goal is to present a solution that works to uncover quantum-vulnerable cryptographic algorithms in any binary executable. To attain this objective, the presented approach leverages static representations via decompiled pseudocode and analyzes them from the perspective of semantic embeddings, guided by three small assumptions:

- Absence of Obfuscation: The tool assumes that the target binaries are unobfuscated. This aligns with its main objective, that is, identifying cryptographic assets within an enterprise environment to construct an internal inventory. Consequently, circumventing obfuscation techniques, which are typically associated with malware evasion and, in some cases, in intellectual property protection, is beyond the intended scope.

- No Malware: The system is predicated on the premise that the programs being analyzed are not malware. This assumption simplifies the analysis process, as malware often employs advanced obfuscation and evasion techniques.
- Similarity to Known Cryptographic Patterns: The tool operates on the premise that the quantum-vulnerable cryptographic algorithms used in the programs exhibit patterns similar to known cryptographic functions. This assumption is crucial for the tool's ability to detect cryptographic functions using Binary Function Similarity Detection (BFSD).

4.2 Design Overview

The approach discussed in this thesis focuses on analyzing decompiled pseudocode to detect quantum-vulnerable cryptographic functions. In that sense, the core problem to be solved is: how can cryptographic functions within binary executables be accurately identified, even when aggressive compiler optimizations are present? To address this challenge, a three-phase system was designed, as shown in Figure 4.1, inspired by the K-CNN-Attention network developed by Jia et al. [25] as discussed in Section 2.3.1.

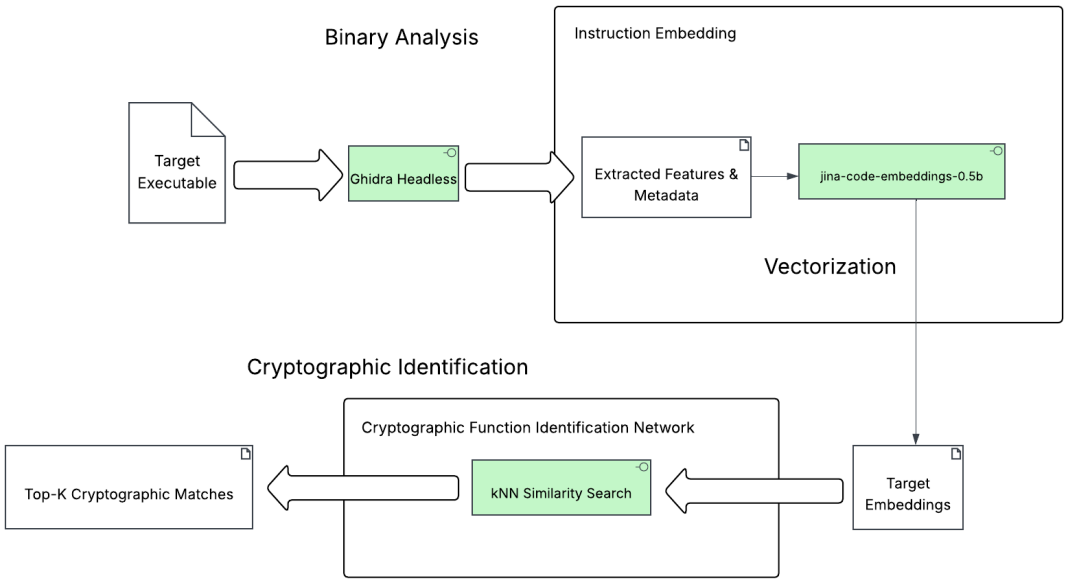

Figure 4.1: System Overview

The first phase is responsible for Binary Analysis, using Ghidra's Headless Analyzer to operate at scale and enable the development of an automated analysis

pipeline. It starts by ingesting a raw executable file and running it through Ghidra's analysis engine, which first disassembles the machine code into assembly instructions and subsequently decompiles it into high-level pseudocode. From these readable formats, the pipeline generates a dataset of the extracted features for each function. This dataset includes the decompiled pseudocode alongside structural metadata, such as the number of basic blocks, memory addresses, compiler, architecture, and opcode hashes.

Then, the second phase handles the vectorization of the extracted pseudocode, generating the static mathematical representations needed to identify quantum-vulnerable cryptographic algorithms. To prepare the data for the machine learning engine, the pipeline feeds the decompiled code into a Code Embedding Model. This model translates the semantic meaning and structural logic of the code into high-dimensional target vectors, creating a standardized format optimized for comparison.

Finally, the third phase aims to pinpoint which functions in the target executable are to be considered as quantum-vulnerable cryptographic routines. It uses a Similarity Search approach to compare the generated embeddings against a database of known cryptographic functions. Thereafter, it uses an algorithm to calculate the mathematical distance between the vectors, outputing a list of the Top-K Most Similar Functions, giving the analyst a highly accurate starting point for identifying where the vulerable crypto is present in the binary.

The design of this pipeline prioritizes modularity to support iterative testing and component evaluation during the early stages of research. By completely decoupling the initial binary analysis from the machine learning layers, the architecture allows for the substitution of individual components, from the primary decompiler and disassembler to the semantic embedding models and distance metrics. In addition, isolating the feature extraction process reduces the complexity of the detection phase, since the identification network operates on standardized mathematical vectors rather than parsing raw machine code or accounting for low-level structural variations.

4.3 Binary Analysis and Feature Extraction

The Binary Analysis phase of the system is responsible for transforming a raw binary executable into a structured representation, to be consumed by the later machine learning stages. This layer converts low-level machine code into a function-oriented data representation that preserves the logic and structural properties of the program, decomposing the executable into individual functions, allowing each routine to be analyzed as a separate candidate for cryptographic identification.

To support this process in a scalable manner, the design relies on Ghidra's

Headless Analyzer, to automatically inspect each input executable, and then recover the program structure, disassemble machine instructions, and decompile detected routines into high-level pseudocode.

Moreover, the main output of this phase is a representation of the analysis data at function-level. Each entry in this representation corresponds to one recovered function from the target executable, as shown in Listing 4.1, and contains multiple categories of information, namely the decompiled code, which provides a higher-level view of the function's behavior and serves as the primary semantic input to the embedding stage, and the function addresses and basic block information, which capture how the routine is organized internally. These features are complemented by contextual metadata, including properties such as architecture and compiler-related information. Finally, the opcode hash is also included as metadata to allow further comparisons to retrieve positive pairs in the similarity search.

Listing 4.1: Segment of the data representation for a single function


```
"00101020": {
    "address": "00101020",
    "start_effective_addr": "0x101020",
    "decompiled_code": "\nvoid FUN_00101020(void)\n\n{\n  (*(code *)
        PTR_00103fb8)();\n  return;\n}\n\n",
    "project": "/path/to/sample_noncrypto",
    "bb_num": 1,
    "ptr_size_bits": 64,
    "library": "openssl",
    "size": 12,
    "end_effective_addr": "0x10102b",
    "name": "FUN_00101020",
    "hashopcodes": "11d0e6d5e77ed5c234c062ed21b5adc8232b40a28471a160ba80aca
       ceded1fe9",
    "arch": "x86:LE:64:default",
    "compiler": "gcc"
},
```


Conversely, decompilation may abstract away implementation details that remain useful for distinguishing function families, especially in optimized binaries. By retaining selected structural and instruction-based descriptors alongside the pseudocode, the pipeline aims to improve robustness against compiler optimizations. A further design consideration in this phase is the standardization of extracted features into a consistent intermediate schema. Because binaries may differ significantly in architecture, compiler settings, and optimization level, the system must represent recovered functions in a normalized format before passing them to the next layer. This ensures that the vectorization stage receives comparable inputs regardless of the original binary's properties.

In this way, it is possible to summarize the objective of the introductory phase as generating a reliable and informative intermediate representation of each recovered

routine. This standardized data representation then becomes the input to the embedding layer, where the extracted pseudocode and associated attributes can be translated into mathematical vectors for code similarity detection.

4.4 Vectorization and Semantic Representation

The second phase of the pipeline is responsible for converting the outputs of the previous phase into a numerical representation that can be compared efficiently in the final detection stage. The extraction and organization of function-level artifacts from the target executable render possible the mapping of each recovered routine to a semantic vector space. In this regard, functions that exhibit similar logic or cryptographic behavior can be represented as nearby points in the vector space, allowing comparison through mathematical distance rather than direct structural matching.

The input to this phase is the decompiled pseudocode associated with each recovered function. This representation expresses program behavior at a higher level of abstraction while still preserving the logic of the underlying binary routine, providing an effective intermediate form.

To generate a semantic representation, the pipeline integrates a Code Embedding Model (`jina-code-embeddings-0.5b`) as a core component of the vectorization layer. In this sense, it encodes each decompiled function into a fixed-length high-dimensional vector that captures relevant semantic characteristics in a standardized mathematical form, as shown in Figure 4.2. This approach takes into account the fact that the recovered pseudocode may vary considerably in size, formatting, and compiler-influenced structure, which would render direct comparison unreliable at scale. Therefore, by translating each function into a uniform vector representation, the system enables semantically meaningful comparisons across routines that may differ syntactically, but implement related operations.

A central design choice of this phase is that vectorization operates at the function level. Thus, by treating each extracted function as an independent semantic unit, the system generates different embeddings for each one. This approach maintains the precise granularity needed to identify specific cryptographic routines within larger executables. Additionally, it enables the similarity search step to rank candidate functions individually rather than evaluating the binary as a whole. Consequently, the output of this phase comprises a discrete set of embeddings, mapped directly to their respective function identifiers.

Hence, the vectorization layer acts as a critical intermediary between symbolic code recovery and similarity-based detection. Its primary objective is not the immediate classification of a function as cryptographic, but rather the generation of a stable semantic representation that facilitates this determination in the subse-

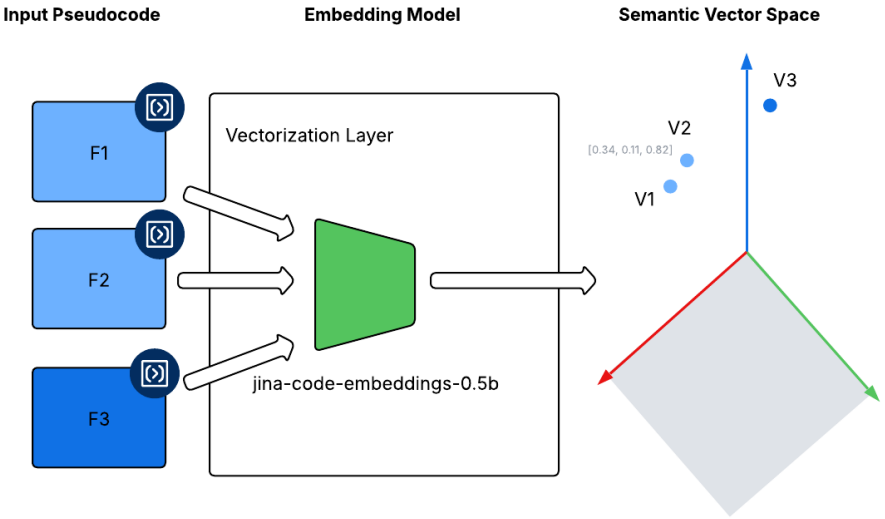

Figure 4.2: Representation of the Vectorization phase (3D space for simplification)

quent phase. Ultimately, these embeddings establish the searchable mathematical foundation upon which the final retrieval and ranking algorithms operate.

4.5 Cryptographic Identification

The third phase of the pipeline is responsible for identifying which functions within the target executable are most likely to correspond to quantum-vulnerable cryptographic routines. After the previous stage converts each recovered function into a semantic vector, this phase performs similarity-based retrieval by comparing those vectors against a reference collection of embeddings derived from known cryptographic implementations (Figure 4.3). In this way, the task is formulated as a nearest-neighbor search in a shared embedding space.

Each function embedding generated from the target binary is queried against a curated dataset, which stores vector representations of previously analyzed cryptographic functions together with their associated labels and metadata. Subsequently, the identification phase ranks the reference candidates according to their mathematical proximity to the query vector, returning the Top-K most similar functions.

Therefore, the output of this step is a ranked list of candidate cryptographic matches for each analyzed function, providing a similarity list that highlights where vulnerable cryptographic behavior is most likely to be present.

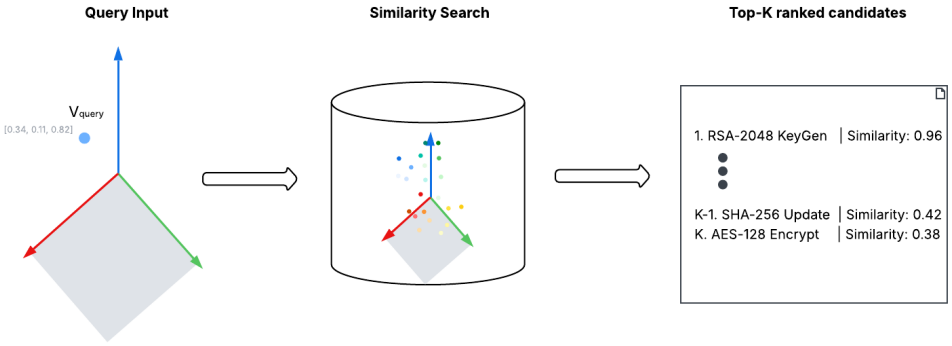

Figure 4.3: Representation of the Identification phase (3D space for simplification)

4.6 Dataset Architecture

The tool uses a reference dataset of known functions as the basis for both training and retrieval. Each analyzed binary is processed with Ghidra, and every recovered function is stored as a JSON record containing its address, name when available, decompiled pseudo-code, size, basic-block count, source library, and source binary. These exported pseudo-code files are the raw corpus from which the training pairs and inference corpus are derived.

The dataset is organized into separate layers. The canonical label map, stored in a JSON file, maps functions to stable identifiers such as `library:function_name`. These identifiers are used to create positive and negative training pairs. For retrieval, the same functions are stored individually in a function-level corpus, with tokenized pseudo-code linked to metadata such as library, address, source file, and canonical identifier.

After inference, each function is represented by a fixed-size embedding vector. A query function is embedded in the same way and compared against the stored reference vectors to retrieve nearest neighbors. The query itself does not need a label; labels are used only to train the model, evaluate results, and interpret the retrieved neighbors.

4.7 Architecture Summary

The architecture of the tool is designed to detect quantum-vulnerable cryptographic functions within binary executables through a three-phase pipeline. The first phase, Binary Analysis, leverages Ghidra's Headless Analyzer to disassemble and decompile raw executables into high-level pseudocode, generating a dataset

of extracted features for each function. The second phase (Vectorization and Semantic Representation) converts the decompiled pseudocode into high-dimensional vectors using a Code Embedding Model, enabling semantic comparisons. Finally, the Cryptographic Identification phase performs similarity searches to identify functions likely to be quantum-vulnerable cryptographic routines by comparing the generated vectors against a reference dataset of known cryptographic functions. This modular architecture supports iterative testing and component evaluation, ensuring robustness and scalability in detecting cryptographic vulnerabilities.

Chapter 5

Implementation

5.1 System Overview

As previously outlined, the developed tool in this thesis aims to detect cryptographic implementations in executable binaries that are vulnerable in the advent of quantum computing. To achieve this objective, it relies on an automated analysis pipeline, which is publicly available on Github [1]. Section 5.2 initially details the development environment. Then, the implementation of the pipeline's stages is described in Sections 5.3, 5.4, and 5.5. Section 5.6 uncovers the embedding model training. The remaining sections unveil additional relevant discussions on the implementation of the tool, with Sections 5.8 and 5.9 exploring performance considerations and encountered challenges.

5.2 Development Environment and Toolchain

This section outlines the chosen toolchain, spanning from the base operating system and hardware specifications to the specific programming languages, reverse engineering frameworks, and machine learning libraries utilized. Explicitly defining this environment not only ensures the reproducibility of the experiments, but also provides the necessary context for the performance metrics discussed in Section 5.8.

The evaluation of the automated analysis pipeline was conducted using Google Colaboratory (Colab). The platform provides a cloud-based Jupyter Notebook environment hosted on virtual machines running Ubuntu Linux. Hardware-accelerated instances allowed accelerating the machine learning stages of the pipeline. Specifically, the experiments were executed using an NVIDIA L4 GPU with approximately 22 GB of VRAM and 53 GB system RAM, as shown in Table 5.1.

The development environment for this project was designed to support two

Table 5.1: Hardware setup

Component	**Specification**
Platform	Linux-6.6.113+-x86_64-with-glibc2.35
GPU	NVIDIA L4
VRAM	22.03 GB
System Memory	52.96 GB

closely connected workflows: large-scale binary analysis and machine learning-based function identification. As a result, the toolchain relied on more than one programming ecosystem, with Python serving as the primary development language, being used to orchestrate the dataset generation, preprocess extracted pseudocode and metadata, manage vectorization experiments, and support the training and evaluation pipeline. In addition, since Ghidra is built on Java and exposes its analysis functionality through that ecosystem, a compatible script written in Java was developed for automated disassembly, decompilation, and feature extraction.

5.3 Binary Analysis and Feature Extraction Implementation

The implementation of the first phase of the pipeline focuses on automating the transformation of binary executables into function-level records. To attain this objective, the system relies on Ghidra as the reverse engineering suite, using its headless execution mode to enable automated processing of binaries at scale.

In that regard, the Headless Analyzer is used to load each target binary, recover function boundaries, disassemble instructions, and generate decompiled pseudocode. The use of headless mode is relevant for dataset construction, since it renders processing binaries in a consistent manner and incorporates reverse engineering directly into the machine learning pipeline as a possible approach.

Feature extraction is performed through Ghidra's Java API [37], which provides access to the recovered program representation. The developed script traverses the discovered functions and exports the information required by later stages of the system. For each recovered function, the pipeline extracts the decompiled pseudocode, together with structural and contextual attributes: function size, name, start and end addresses, number of basic blocks, pointer size, architecture, and compiler information. These fields are included to preserve information that is not fully captured by the decompiled text alone: size and basic-block count describe the function's structural complexity, addresses identify the function within the analyzed binary, and pointer size, architecture, and compiler specification provide

the execution and compilation context needed when comparing functions across binaries. The script also computes an opcode-based fingerprint by iterating over the instructions in the function body, collecting their mnemonic strings in address order, concatenating them with spaces, and applying SHA-256 to the resulting sequence. Additional fields, such as the originating project or library, are also included when available, in order to support the analysis.

The extracted data are then represented in a JSON-based intermediate format, with each function represented as an individual record, as shown in Listing 4.1. This format allows for accommodating heterogeneous attributes, including free-form pseudocode and textual metadata, while remaining straightforward to consume in later stages of the pipeline. Thus, instead of producing a summary for the entire executable, the system generates a collection of independent records corresponding to individual routines, which allows the embedding stage to operate directly on the decompiled code of each function, while retaining the metadata needed to map any later similarity match back to a specific location in the original binary.

Furthermore, the pipeline relies on Ghidra to handle practical binary analysis challenges such as missing symbols or variations introduced by compiler optimizations. Moreover, even when human-readable function names are not available, the system can still assign internal identifiers and extract pseudocode for recovered routines. Thus, the quality of extracted artifacts remains dependent on the success of the underlying decompilation and function recovery process.

In summary, the output of this phase is a structured JSON dataset containing function-level pseudocode and associated metadata for the target binary. This dataset serves as the direct input to the vectorization stage.

5.4 Vectorization Layer

The vectorization layer converts the function-level output of binary analysis into semantic embeddings suitable for retrieval, which occurs as shown in Figure 5.1 and described hereafter. In the codebase, this responsibility is formalized through the RepresentationBuilder interface, which defines how to consume the analysis output and produce the representation result. The concrete implementation used in this project is the PseudoEmbeddingBuilder, whose purpose is to transform Ghidra-exported pseudocode into embedding files that can be consumed by the later evaluation stage. In that regard, the input to this layer is the JSON artifact generated during binary analysis.

The implementation performs a lightweight preprocessing step before embedding. Specifically, this stage uses tree-sitter with the C++ grammar to parse the decompiled pseudocode and remove selected syntactic elements, namely comments and variable declarations. Since array definitions are represented as variable

declarations in this grammar, local array declarations are also removed; however, subsequent uses of arrays, such as indexing and assignments, are preserved. The purpose of this step is to reduce decompiler-generated boilerplate and type/storage artifacts while retaining the portions of the pseudocode that most directly reflect operational behavior. After this filtering stage, empty lines are removed and the remaining code is preserved as the textual input to the embedding model.

Embedding generation is then carried out, loading a pretrained transformer-based code model and tokenizer. The model is executed in inference mode and processes the cleaned pseudocode in batches. Because functions vary substantially in length, tokenization is performed with truncation and padding to a fixed maximum sequence length of 512 tokens. The model output is reduced to a single vector per function through mean pooling over the token embeddings. The resulting vector is then L2-normalized, i.e., divided by its Euclidean norm, so that all function embeddings have unit length. This normalization removes magnitude differences between vectors and makes cosine-based comparison depend only on their direction in the embedding space. As a result, each recovered routine is mapped to a fixed-size unit-length embedding that can be compared directly using metrics such as cosine-based similarity in the final retrieval stage. The output of the vectorization layer consists of serialized embedding files.

5.5 Cryptographic Identification Module

In this sense, each function embedding extracted from a target binary is compared with a reference corpus of embeddings previously generated from the project's curated dataset of known cryptographic libraries and binaries, and the most similar candidates are returned as potential cryptographic matches.

Thus, this functionality is implemented through a k-nearest-neighbors search over serialized embedding files. The module initially loads the embedding files produced by the vectorization stage and converts them into a matrix representation for efficient comparison. Before retrieval, all vectors are normalized to unit length, which allows similarity to be computed through the dot product of normalized vectors, equivalent to cosine similarity. The search procedure then computes pairwise similarities between query embeddings and indexed reference embeddings, ranks the results in descending order, and returns the Top-K nearest neighbors for each query function.

Integration with the embedding module is achieved through the serialized output produced during the representation stage. The identification module consumes the pickle files containing function-level embeddings and preserves the mapping between each vector and its original function identifier. In addition, the module can recover and display the associated decompiled code, when the identifiers correspond

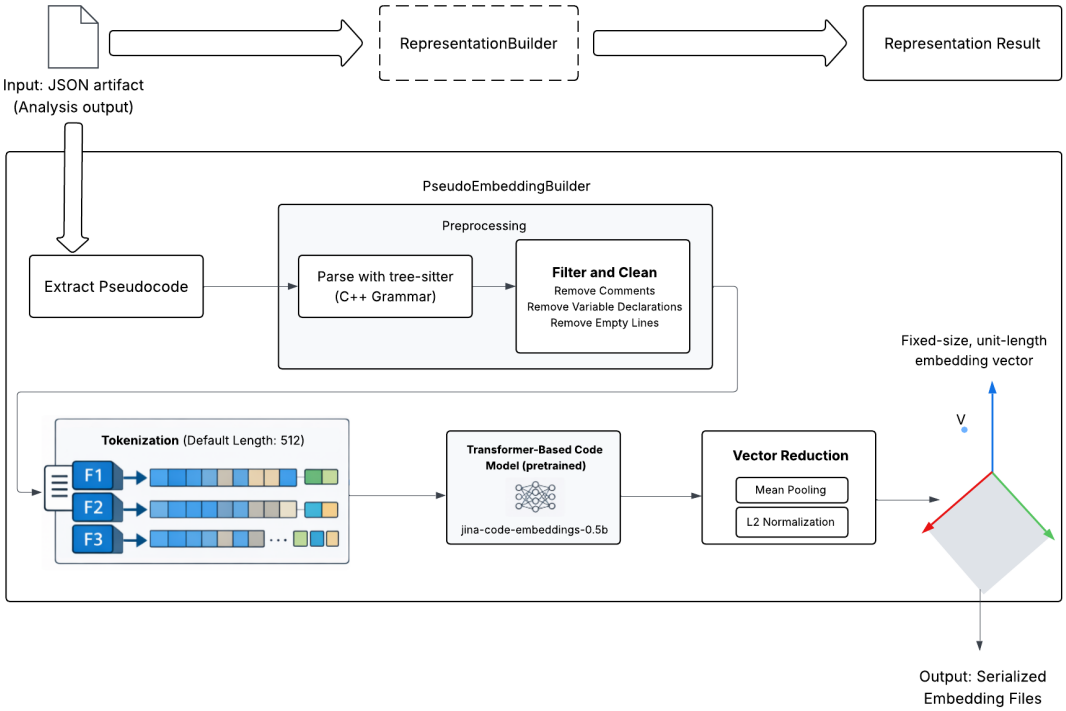

Figure 5.1: Vectorization Layer Workflow

to entries in the pseudocode JSON files, making the retrieved results easier to interpret.

5.6 Embedding Model Training and Inference

In addition to the binary analysis, vectorization, and retrieval components, the system includes a dedicated training module for adapting the embedding model to the target domain of decompiled pseudocode. The purpose of this stage is to improve the quality of the semantic representation used in the vectorization and retrieval stages of the pipeline. In this perspective, the training module serves as an intermediate optimization layer between raw decompiled code and similarity-based cryptographic identification.

The training pipeline is implemented using PyTorch. A pretrained Jina code embedding transformer is first loaded as the base representation model, after which it is optimized on paired function samples. The training data are organized as function pairs together with similarity scores, allowing the model to learn an embedding space in which semantically related routines are mapped closer together than unrelated ones. This formulation is consistent with the later use of nearest-neighbor retrieval, considering that the objective of training is to improve relative distances between function embeddings rather than to predict discrete labels directly.

At the implementation level, each training batch contains two tokenized function inputs and an associated similarity label. The model encodes both inputs, and the resulting token-level outputs are reduced by mean pooling and then L2-normalized to produce one embedding per function. Cosine similarity is then computed between the paired embeddings, and the training objective is applied over these similarity scores. The CoSENT loss function [38] is used for this purpose, since it optimizes the relative ordering of similarity scores across training pairs.

The training loop further incorporates several practical mechanisms to stabilize optimization. The model parameters are updated using the AdamW optimizer, while the learning rate is scheduled through cosine annealing with an additional linear warmup phase during the early steps of training, taking as base the training part of the dejina tool presented by Shi et al. [21].

A separate inference routine is provided to generate embeddings from a trained checkpoint once fine-tuning has been completed. In this sense, the model, in evaluation mode, is used to encode tokenized function inputs without gradient computation. The resulting normalized embeddings are then serialized for use by the retrieval module.

In summary, the training stage determines how well the embeddings capture similarities between cryptographic routines, playing a pivotal role in the effectiveness

of the final identification process.

5.7 Dataset Construction

The dataset was constructed from third-party library binaries and additional cryptographic library packages. Inspired by the organization used by the BinAres dataset [21], the initial binary corpus was organized under the `third_lib` directory and contained multiple versions of common libraries. The main training corpus used in this work focused on Linux x86-64 ELF binaries. To account for compiler and optimization variability, additional source-built binaries were generated with GCC and Clang under multiple optimization levels, namely `-O0`, `-O1`, `-O2`, `-O3`, and `-Os`.

To improve coverage of cryptographic and post-quantum cryptographic functionality, additional binaries were acquired or built for libraries such as OpenSSL, libgcrypt, liboqs, mbedTLS, nettle, libsodium, and wolfSSL. These were complemented with non-cryptographic libraries such as zlib, sqlite, cJSON, libpng, libxml2, and libexpat, which served as negative examples. A separate AArch64 subset was later generated from selected cryptographic libraries and used only for cross-ISA inference evaluation, not for training. Additional dataset construction details are presented in the Appendix.

Each binary was analyzed with Ghidra's headless analyzer. The custom export script extracted one JSON record per discovered function, including its address, function name, size, basic block count, opcode hash, and decompiled pseudo-code. The final source-options export used in this stage contained 139 pseudo-code JSON files. These exports were used in two ways. For contrastive pair construction, the pipeline applied strict filtering: very small functions, functions with fewer than two basic blocks, runtime stubs, excessive duplicate opcode-hash groups, and functions without a usable canonical identifier were removed. After this filtering, 47,598 functions remained available for pair construction. From these functions, the training pipeline generated 16,000 training pairs and 4,000 evaluation pairs, using a 1:2 positive-to-negative ratio. In addition to this training corpus, the separate AArch64 subset contained 56 pseudo-code JSON files.

The labeling strategy was based on canonical function identity. Each function was assigned a canonical identifier, typically of the form `library:function_name`. Functions with the same canonical identifier across different binaries, versions, or distributions were treated as positive pairs, and functions with different canonical identifiers were treated as negative pairs.

As previously outlined, positive pairs were formed from functions sharing the same canonical identifier. Negative pairs, on the other hand, were sampled from different canonical identifiers, with a mixture of random negatives and hard

negatives. Hard negatives were selected from functions with similar size and basic-block structure. Functions smaller than 20 instructions or with fewer than two basic blocks were filtered out, and duplicate functions were also capped.

The final pair dataset used the following split:

Split	Total Pairs	Positive Pairs	Negative Pairs
Training	16,000	5,333	10,667
Evaluation	4,000	1,333	2,667

Table 5.2: Pair dataset split used for contrastive training.

The split was performed at the canonical identifier level, so that function identities used for evaluation were separated from those used for training.

Several automation scripts were developed to make the dataset construction reproducible. More details on the reproduction of the dataset construction are presented in the tool repository.

In addition to the pair-training labels, category labels were created for retrieval evaluation. The categories used were `classical_pq_vulnerable`, `pqc_kem`, `pqc_signature`, and `noncrypto`. A rule-based seed label file was generated from function names and library metadata, and a smaller manually reviewed label file was later produced to support a higher-confidence evaluation set.

5.8 Optimization and Performance Considerations

Several optimizations were necessary to make the pipeline practical on a corpus of more than one hundred thousand functions. The first optimization was filtering: very small functions and functions with fewer than two basic blocks were removed before pair generation, since these functions often correspond to trivial wrappers, import stubs, or even compiler-generated code.

Additionally, pair generation was also bounded: without limits, functions with many versions could generate a large number of positive pairs. To control this, the number of positive pairs per canonical identifier was capped, and the final dataset size was limited to 16,000 training pairs and 4,000 evaluation pairs. Negative sampling used a fixed negative-to-positive ratio of 2:1.

For the inference corpus, function-level examples were tokenized once and saved as a dataset. Memory usage was managed by limiting sequence length, batching examples, and writing intermediate artifacts to disk. The maximum sequence length was set to 512 tokens for pair construction and inference - the length of 1024

was also tested, and the comparison results are shown in Section 6. As previously mentioned, training was performed on Google Colab with an NVIDIA L4 GPU.

There is a trade-off between accuracy and efficiency throughout the system. Keeping more functions improves coverage but increases noise and computational cost. To mitigate this, we filter out unusually small functions. This improves overall dataset quality because small functions often lack distinct semantic meaning and can confuse the model with highly generic patterns, though this filtering risks discarding some legitimate wrapper functions. Furthermore, the training process incorporates hard negatives. These are mismatched examples that are deceptively similar to correct matches, which forces the model to learn fine-grained, discriminative features rather than relying on superficial similarities. While this yields a more robust and accurate model, it significantly increases preprocessing complexity, as identifying and generating these challenging examples requires advanced mining strategies rather than simple random sampling. Similarly, using decompiled pseudo-code provides richer semantic input than raw symbol names, but it depends on Ghidra quality and can be affected by decompiler artifacts.

Pair construction was fast once pseudo-code JSON files were available. The expensive stages were Ghidra analysis, model training, and full-corpus embedding inference.

5.9 Implementation Challenges

The first major implementation challenge was binary heterogeneity. The dataset contains libraries from different versions, distributions, and build configurations. Some binaries preserve useful symbols, while others contain generic names with no usefulness. This makes labeling and canonical identity assignment difficult. To reduce this problem, the system uses trusted symbol maps where available and avoids relying on addresses as stable identifiers.

The raw binary dataset also contained noisy artifacts. For example, some files inside the pre-existing BinAres-style directory tree were zero-byte files rather than valid ELF binaries. One such case was found under the nginx subset, where `file` reported the input as `empty`. These files appear to be dataset artifacts or incomplete entries rather than usable binaries. The export scripts therefore check file type and skip unsuitable inputs. Even valid binaries can produce noisy decompiler output, especially for tiny functions, stubs, runtime helpers, and compiler-generated code. Filtering by function size and basic-block count was introduced to reduce these cases.

Labeling presented another challenge. A function name containing terms such as RSA, DH, ECDSA, or ML-KEM is strong evidence, but not always sufficient. Some functions are core cryptographic implementations, while others are lifecycle

helpers or generic wrappers. To address this, the project distinguishes between broad rule-based labels and a smaller manually reviewed evaluation set. The manual benchmark contained 1,000 reviewed queries, including 262 classical PQ-vulnerable functions, 223 PQC KEM functions, 373 PQC signature functions, and 142 non-cryptographic functions. Ambiguous functions can be marked as `skip` in the manual benchmark rather than forced into a noisy category.

Chapter 6

Experimental Evaluation and Results

6.1 Experimental Setup and Ground Truth

The experimental evaluation was designed to answer two main questions:

- **(Q1)** Does the fine-tuned embedding model retrieve semantically related functions across different binaries, versions, and libraries?
- **(Q2)** Can the same retrieval mechanism support the identification of functions that are relevant to post-quantum migration, especially classical public-key cryptographic functions that are vulnerable to quantum attacks?

The evaluation was performed on a function-level inference corpus constructed from the exported pseudo-code files. The main source-options inference corpus contained 291,124 functions extracted from 139 pseudo-code JSON exports. The corpus covered classical cryptographic libraries and post-quantum cryptographic libraries, including OpenSSL, libgcrypt, liboqs, mbedTLS, nettle, libsodium, and wolfSSL, together with non-cryptographic libraries such as sqlite, zlib, cJSON, libpng, libxml2, and libexpat. As previously mentioned, a separate AArch64 inference corpus was later constructed for cross-ISA evaluation and was kept out of the training data.

The evaluation follows two complementary parts. The first part is a standard pair evaluation. During pair construction, canonical function identifiers were split into separate training and evaluation sets. The final pair dataset contained 16,000 training pairs and 4,000 evaluation pairs. Because the split was performed by canonical identifier, function identities used for evaluation were not reused as training identities. This evaluation checks whether the contrastive model learned to distinguish matching and non-matching function pairs.

The second part is a corpus-level retrieval evaluation. After training, every function in the inference corpus was embedded using the fine-tuned model. A labeled function was then used as a query, and the remaining corpus was ranked by embedding similarity. This setting is closer to the intended use case: given a function found in a binary, the system should retrieve semantically related functions and help identify whether the function belongs to a relevant cryptographic category.

Two forms of ground truth were used. The first was based on canonical function identity, mapping exported functions to trusted canonical identifiers, typically derived from the library name and exported symbol name. Canonical identifiers were used for constructing positive and negative training pairs and for exact function-match evaluation across versions and distributions.

The second form of ground truth was based on broader vulnerability-oriented categories. A category label map was created to classify functions into `noncrypto`, `classical_pq_vulnerable`, `pqc_kem`, and `pqc_signature`. The initial category labels were generated automatically using library-aware symbol-name rules. In practice, this means that function names were matched against algorithm keywords and naming conventions associated with each category. For example, symbols containing terms such as `RSA`, `DSA`, `DH`, `ECDH`, `ECDSA`, `Ed25519`, or `Curve25519` were treated as candidates for the `classical_pq_vulnerable` category, while liboqs symbols containing names such as `ML-KEM`, `Kyber`, or `OQS_KEM` were mapped to `pqc_kem`, and symbols containing `ML-DSA`, `Dilithium`, `SPHINCS`, or `OQS_SIG` were mapped to `pqc_signature`. Functions from non-cryptographic libraries such as zlib, sqlite, cJSON, libpng, libxml2, and libexpat were labeled as `noncrypto`.

Because symbol names alone can be noisy, a manually reviewed subset was also created as a higher-confidence benchmark. In this manual review set, ambiguous helper functions could be marked as `skip`, while accepted functions were assigned a final manually reviewed category.

This setup separates training from evaluation in two ways. First, the pairwise training and evaluation datasets are split by canonical identifier, preventing the same canonical function identity from appearing in both training and pair evaluation. Second, the corpus-level retrieval evaluation uses a separate function-level corpus and category labels to test whether the trained embeddings are useful for search and vulnerability-oriented identification. This allows the experiments to evaluate not only pair similarity, but also the practical retrieval behavior required by the proposed tool.

6.2 Evaluation Metrics

The system was evaluated using metrics from information retrieval and category-based identification. Since the model produces embeddings, evaluation is performed

by ranking candidate functions according to similarity to a query function. The top-ranked results are then compared against the ground truth labels.

For semantic retrieval, the primary evaluation metrics are HitRate@K and Mean Reciprocal Rank (MRR). HitRate@K measures whether at least one correct result appears within the first K retrieved candidates. In this context, a correct result is defined as a true positive match according to the dataset's ground truth. In this work, K values of 1, 5, and 10 are used. HitRate@1 is the strictest setting, requiring the absolute nearest neighbor in the vector space to be the ground-truth match. HitRate@5 and HitRate@10 measure whether at least one correct result appears within a small set of candidates.

MRR measures how early the first correct result appears in the ranking. For each query, the reciprocal rank is computed as $1/r$, where r is the rank position of the first correct result. The final MRR is, then, the mean over all evaluated queries. This metric is useful because it rewards systems that place correct matches near the top of the result list.

For vulnerability-oriented identification, Precision@K is used to measure the proportion of the top K retrieved functions that belong to the correct category. While HitRate@K evaluates whether at least one relevant function is successfully retrieved, Precision@K assesses the overall relevance of the retrieved set. This is important for the intended use case: if the tool is used to prioritize functions for inspection, the top results should contain as few irrelevant functions as possible.

False positives are also considered in the vulnerability-identification setting. They are especially important because a practical migration-assessment tool should reduce analysts workload.

Together, these metrics evaluate both sides of the system. HitRate@K and MRR measure whether semantically related functions can be found in the embedding space, while Precision@K and false-positive analysis measure whether the retrieved functions are useful for identifying post-quantum migration-relevant code.

6.3 Phase 1: Semantic Similarity and Retrieval Evaluation

The first evaluation phase measures whether the fine-tuned embedding model captures semantic similarity between functions. The objective is to determine whether related cryptographic functions are placed near one another in the embedding space. This is important because the proposed tool relies on nearest-neighbor retrieval.

For each query function, the system ranks all candidate functions by embedding similarity. The main retrieval metrics are HitRate@K, Precision@K, and Mean Reciprocal Rank (MRR). HitRate@K measures whether at least one correct result

appears within the top K retrieved functions, Precision@K measures how many of the top K results are correct, and MRR measures how early the first correct result appears.

Table 6.1 shows the retrieval performance on the rule-based silver-label evaluation set and on the manually reviewed benchmark. The latter is smaller, but its labels were manually validated and ambiguous helper or wrapper functions were excluded from the benchmark.

Evaluation Set	Queries	Hit@1	Hit@10	Prec.@1	Prec.@10	MRR
Silver-label	5000	0.983	0.989	0.983	0.979	0.985
Manual-review benchmark	1000	0.950	0.974	0.950	0.934	0.955

Table 6.1: Retrieval performance on the source-options silver-label evaluation and manually reviewed benchmark.

The high Hit@1, Precision@1, and MRR indicate that the nearest neighbor is usually from the relevant category and appears early in the ranking. The Hit@10 values are also high for both benchmarks, showing that even when the first retrieved result is not correct, a relevant function is usually present within the top ten candidates. This suggests that the fine-tuned embedding model learned a meaningful representation of decompiled cryptographic functions rather than relying only on superficial features such as function size.

6.4 Phase 2: PQ-Vulnerability Identification

The second evaluation phase measures whether the retrieval system can support the primary objective of this work, which is identifying functions relevant to post-quantum migration. In this setting, the goal is to distinguish between classical public-key cryptography, post-quantum cryptography, and non-cryptographic code.

The category labels used in this phase, as previously presented, are `noncrypto`, `classical_pq_vulnerable`, `pqc_kem`, and `pqc_signature`. In this context, the `classical_pq_vulnerable` category includes classical public-key algorithms such as RSA, finite-field Diffie-Hellman, DSA, ECDH, ECDSA, and EdDSA. The `pqc_kem` and `pqc_signature` categories represent post-quantum cryptographic functionality, while `noncrypto` represents functions from libraries that are not cryptographic targets.

Table 6.2 reports the manually reviewed category-level retrieval results. The strongest performance is observed for non-cryptographic functions, where the model achieves perfect Hit@1 and Precision@1 on the sampled manual benchmark, but also with the smallest amount of queries.

Category	Queries	Hit@1	Hit@10	Prec.@1	Prec.@10	MRR
Classical PQ-vulnerable	238	0.920	0.954	0.920	0.934	0.929
PQC KEM	251	0.932	0.952	0.932	0.913	0.936
PQC signature	474	0.970	0.994	0.970	0.947	0.974
Noncrypto	37	1.000	1.000	1.000	0.908	1.000
Overall	1000	0.950	0.974	0.950	0.934	0.955

Table 6.2: Category-level retrieval results on the manually reviewed v3 source-options evaluation set.

For the primary target category, `classical_pq_vulnerable`, the model achieves 0.920 Precision@1 and 0.929 MRR, showing that classical public-key cryptographic functions are usually retrieved near other migration-relevant functions.

The non-cryptographic category is important for estimating false positives. A practical migration-assessment tool should not flag large numbers of unrelated functions from libraries such as sqlite, zlib, libpng, or libxml2 as cryptographic migration targets. The manually reviewed benchmark contains only 37 non-cryptographic queries after ambiguous rows were excluded, so this result should be interpreted cautiously. Nevertheless, the observed Hit@1 and Precision@1 indicate that, within this sample, non-cryptographic functions were not confused with migration-relevant cryptographic code at the top-ranked position.

The results suggest that the tool is effective as a prioritization system. It ranks functions according to semantic similarity and category evidence, helping an analyst focus on code regions likely to contain classical public-key cryptography or post-quantum cryptographic implementations.

6.5 Robustness Evaluation: Symbol Names and Cross-ISA Transfer

The previous experiments evaluate the setting where the decompiled pseudo-code retains the symbol information available in the analyzed binaries. However, real-world binaries may be stripped, partially stripped, or otherwise obfuscated. In such cases, function names may be unavailable or replaced by generic decompiler names. To evaluate how much the model relies on symbol-name cues, an additional robustness experiment was performed using a no-name representation. In this setting, function identifiers in the pseudo-code were normalized before tokenization, while the rest of the decompiled code was kept unchanged.

Table 6.3 compares the named-code baseline with several no-name variants. The first no-name setting evaluates the original source-options model directly on

name-normalized pseudo-code. The second setting uses a model trained on no-name pairs. The third setting additionally balances the training pairs by category. Finally, a 1024-token no-name variant was tested to determine whether the drop in performance was caused by insufficient context length.

Evaluation Setting	Queries	Prec.@1	Prec.@10	MRR	Classical Prec.@1	Classical MRR
Named pseudo-code (v3)	5000	0.983	0.979	0.985	0.924	0.931
No-name, unadapted model (v3)	5000	0.918	0.889	0.933	0.831	0.860
No-name trained model (v4)	5000	0.934	0.906	0.943	0.854	0.877
No-name balanced model (v5)	5000	0.934	0.911	0.945	0.856	0.880
No-name 1024-token model (v6)	5000	0.920	0.892	0.934	0.814	0.843

Table 6.3: Effect of removing function-name cues from the pseudo-code representation. The classical columns report performance on the `classical_pq_vulnerable` category.

The results show that symbol names contribute substantially to retrieval quality. When the v3 model is evaluated directly on no-name pseudo-code, overall Precision@1 decreases from 0.983 to 0.918, and the classical PQ-vulnerable category decreases more sharply. Training directly on no-name pairs improves the result, increasing classical Precision@1 from 0.831 to 0.854. Category balancing provides only a marginal additional gain. Increasing the token length to 1024 does not improve the no-name setting; in fact, it reduces classical Precision@1 to 0.814. This suggests that the remaining no-name error is not primarily caused by insufficient context length, but by the loss of useful semantic cues and by structural similarity between cryptographic helper functions.

A second robustness experiment evaluated cross-ISA transfer. For this test, the AArch64 corpus was evaluated with the no-name balanced model trained on the main x86-64 corpus. The AArch64 binaries were not used for training. This creates a zero-shot cross-ISA setting: the model must retrieve relevant functions from AArch64 decompiler output using a representation learned from x86-64 binaries.

Category	Queries	Prec.@1	Prec.@10	MRR
All labeled AArch64 queries	4247	0.838	0.780	0.877
Classical PQ-vulnerable	2695	0.889	0.847	0.916
PQC KEM	8	1.000	0.375	1.000
PQC signature	1544	0.750	0.667	0.807

Table 6.4: Zero-shot no-name AArch64 retrieval results using the x86-64-trained no-name model.

The overall AArch64 no-name result is lower than the x86-64 no-name baseline, mainly due to weaker performance on PQC signature functions and the imbalance of the AArch64 subset. The PQC KEM category contains only eight evaluated

queries, so it should not be interpreted as a stable estimate. The most important result for the goal of this work is the classical PQ-vulnerable category. In this category, the model achieves 0.889 Precision@1 and 0.916 MRR, which is higher than the x86-64 no-name classical baseline. This indicates that the model can still prioritize migration-relevant classical cryptographic functions across ISA boundaries, although broader cross-ISA evaluation with balanced non-cryptographic and PQC subsets remains necessary.

Overall, these robustness experiments show that PQMap is useful beyond the ideal named-symbol setting, but also expose an important limitation. Symbol names provide meaningful supervision and retrieval cues. Removing them reduces accuracy, especially for classical cryptographic functions. Training with name-normalized data partially mitigates this issue, but does not fully close the gap. Cross-ISA evaluation is encouraging for the target class, but the current AArch64 subset is not balanced enough to support broad conclusions for all categories.

6.6 Qualitative Analysis and Case Studies

In addition to the quantitative metrics, qualitative retrieval examples were inspected to determine whether the model's results would be useful to a human analyst. This is important because the intended use case is interactive binary analysis: the system should help an analyst discover related cryptographic functions, not only produce aggregate scores.

Case Study 1: Classical Cryptography Retrieval

A representative classical cryptography query was selected from an OpenSSL RSA function. When querying with an OpenSSL RSA public encryption function, the nearest neighbors included RSA-related functions from wolfSSL, mbedTLS, and OpenSSL itself. The top retrieved functions included wolfSSL RSA public encryption, wolfSSL RSA private encryption, OpenSSL RSA public encryption, and mbedTLS RSA public-key routines.

This result is significant because the retrieved functions are not merely copies from the same binary. They come from different libraries and different implementations of the same cryptographic family. For a reverse engineer, this behavior is useful: an unknown RSA-like function in one binary can be compared against known RSA implementations from other libraries, helping identify its likely role.

A similar pattern was observed for elliptic-curve signature verification. An OpenSSL ECDSA verification query retrieved wolfSSL ECDSA verification functions, OpenSSL ECDSA verification wrappers, and nettle ECDSA verification routines. Some DSA and EdDSA-related functions also appeared nearby. This

suggests that the embedding space captures common sign/verify behavior, while also showing a limitation: different public-key signature families can be close because they share hashing, encoding, and verification structure.

Case Study 2: Post-Quantum Cryptography Retrieval

A representative post-quantum query was selected from an ML-KEM function in liboqs. When querying with `pqcrystals_ml_kem_512_ref_ntt`, the nearest neighbors were other ML-KEM NTT functions across different liboqs builds and optimization settings. The model also retrieved related inverse NTT functions and nearby ML-DSA NTT functions.

This behavior is useful for two reasons. First, the model correctly groups the same ML-KEM operation across multiple binaries, showing robustness to build variation. Second, the retrieval of inverse NTT and ML-DSA NTT functions indicates that the model recognizes lower-level algorithmic structure shared between lattice-based schemes. For an analyst, this can help identify polynomial arithmetic routines that are part of post-quantum implementations even when top-level API symbols are absent.

6.7 Discussion and Key Findings

The experimental results show that the proposed embedding-based approach is effective for retrieving cryptographically related functions from decompiled binaries. In the manually reviewed benchmark, the system achieved 0.950 Precision@1 and 0.955 MRR overall. For the main target category, `classical_pq_vulnerable`, it achieved 0.920 Precision@1 and 0.929 MRR, indicating that classical public-key functions are usually retrieved near other migration-relevant functions.

The strongest category in the manually reviewed benchmark was `pqc_signature`, with 0.970 Precision@1 and 0.974 MRR. This result improved after ambiguous generic signature wrappers and classical signature helpers were excluded or corrected during manual review. PQC KEM functions also performed well, reaching 0.932 Precision@1 and 0.936 MRR. ML-KEM and related KEM functions tend to contain distinctive polynomial arithmetic and encapsulation/decapsulation patterns, which makes them suitable for embedding retrieval.

Classical PQ-vulnerable functions, which are the most relevant functions for the purpose of this tool, also performed well. RSA, DH, ECDSA, EdDSA, and related public-key routines were usually retrieved near functions from the same cryptographic family, including cross-library matches. The robustness experiments further show that even when function-name cues are removed, the model retains useful retrieval ability, although performance decreases. The no-name balanced

model achieved 0.856 Precision@1 for classical functions, and the AArch64 no-name cross-ISA test reached 0.889 Precision@1 for the same category.

For a reverse engineer or security analyst, the main value of the tool is prioritization. The system can suggest likely cryptographic neighbors for a function under investigation and can help identify whether the surrounding code is related to classical PQ-vulnerable cryptography or post-quantum cryptography. This is particularly useful when analyzing stripped or partially stripped binaries, where function names may be unavailable or incomplete.

Several edge cases remain. Small wrappers, initialization functions, size helpers, and generic sign/verify APIs can be difficult to categorize. In the PQC setting, lattice-based helper routines may be shared across KEM and signature schemes, which can create ambiguity between PQC categories. These issues motivate the limitations and future work discussed in Chapter 7, including larger manually reviewed benchmarks, finer-grained category labels, stronger evaluation on stripped binaries, and broader cross-ISA datasets.

Chapter 7

Limitations and Future Work

The results show that embedding-based retrieval is useful for identifying cryptographic functions relevant to post-quantum migration, but several limitations remain. The most important limitation is the system's dependence on labeled reference data. Labels are used to construct positive and negative training pairs, to evaluate retrieval quality, and to interpret retrieved neighbors. Without reliable labels in the reference corpus, the system cannot directly know whether a retrieved function represents RSA, ECDSA, ML-KEM, or non-cryptographic code.

However, labels are not required for the query function at inference time. Given an unknown function, the tool can decompile it with Ghidra, embed its pseudo-code, and retrieve similar functions from the labeled reference corpus. In this setting, the label is not attached to the unknown function itself; instead, the likely category is inferred from the nearest labeled neighbors. For example, if an unlabeled function retrieves several RSA, DH, or ECDSA functions, it can be flagged as likely related to classical post-quantum-vulnerable cryptography. This makes the tool useful for exploratory reverse engineering, but the result should be interpreted as evidence rather than proof.

A related limitation is reliance on symbol information. Many training and evaluation binaries contain function names, and these names can appear in the exported pseudo-code. The robustness experiments show that this information is useful: when function-name cues were removed, retrieval performance decreased. The no-name balanced model partially recovered this loss, reaching 0.856 Precision@1 for classical PQ-vulnerable functions, but it did not match the named-code setting. Increasing the context window to 1024 tokens did not solve the issue and reduced performance in the no-name experiment. This indicates that the remaining gap is not only caused by truncated pseudo-code, but also by the loss of semantic cues and by structural similarity between cryptographic helpers. Future work should therefore evaluate fully stripped binaries and develop representations that are less dependent on symbol names, for example by combining pseudo-code with

control-flow or instruction-level features.

Label noise is another limitation. The rule-based category labels provide broad coverage, but they are not perfect. Function names such as RSA, DH, ECDSA, or ML-KEM are strong signals, but they do not always indicate that the function implements a core cryptographic operation. Some functions are wrappers, lifecycle helpers, parameter encoders, or side routines. The manually reviewed benchmark reduces this problem by excluding ambiguous helpers and correcting clear rule-based mistakes, but it is still small compared with the full corpus. A larger manually reviewed benchmark would provide a stronger estimate of real-world performance.

The current category taxonomy is also coarse. The evaluation groups functions into `classical_pq_vulnerable`, `pqc_kem`, `pqc_signature`, and `noncrypto`. These categories are useful for post-quantum migration analysis, but they hide important distinctions. RSA encryption, RSA signatures, finite-field Diffie-Hellman, ECDH, ECDSA, and EdDSA are all grouped into one classical category. Future work should use a finer taxonomy that separates algorithm families and function roles, such as key generation, key exchange, encryption, signing, verification, encapsulation, and decapsulation.

The system also depends on decompiler quality. Ghidra pseudo-code provides a normalized representation of binary functions, but it is not always stable. Compiler optimizations, architecture differences, stripped symbols, and decompiler limitations can change the pseudo-code representation of the same source-level function. Some functions may be split, merged, or represented with generic temporary variables. The AArch64 no-name experiment showed encouraging cross-ISA behavior for classical PQ-vulnerable functions, but the AArch64 subset was not balanced across all categories. Broader cross-ISA evaluation is therefore still needed.

The tool should be understood as a retrieval and prioritization system, not as a formal vulnerability detector. A retrieved RSA, DH, or ECDSA function is not necessarily a software bug. It is considered relevant to post-quantum migration because the underlying classical public-key algorithm is threatened by large-scale quantum computers. The tool helps identify functions that deserve migration analysis, but final interpretation still requires analyst review.

Future work should expand both the corpus and the evaluation protocol. This includes more libraries, more architectures, stripped and obfuscated binaries, independently compiled implementations of the same algorithms, and larger manual benchmarks. It would also be useful to evaluate mixed representations that combine decompiled pseudo-code, control-flow graphs, call-graph context, and instruction-level features. Such representations may improve robustness when symbols are unavailable and may help separate cryptographic implementations from wrappers and helper routines.

Chapter 8

Conclusion

This work presented an embedding-based approach for identifying cryptographic functions relevant to post-quantum migration in compiled binaries. The motivation was that many existing binaries may contain classical public-key cryptography such as RSA, Diffie-Hellman, DSA, ECDH, ECDSA, or EdDSA, which are vulnerable in a post-quantum threat model. Manually finding these functions in large binary corpora is difficult, especially when libraries, versions, compiler settings, architectures, and symbol availability vary. The proposed tool addresses this problem by using decompiled pseudo-code and learned function embeddings to retrieve semantically related cryptographic functions.

The system was built as a multi-stage pipeline. First, binaries from cryptographic, post-quantum, and non-cryptographic libraries were analyzed with Ghidra. The exported pseudo-code was then used to construct a pairwise training dataset based on trusted canonical function identifiers. A contrastive model was fine-tuned to bring related functions closer together in the embedding space while separating unrelated functions. Finally, a function-level inference corpus was embedded and used for nearest-neighbor retrieval and category-level analysis.

The evaluation showed that the approach is effective as a retrieval and prioritization tool. In the manually reviewed benchmark, the model achieved 0.950 Precision@1 and 0.955 MRR overall. For the main category, `classical_pq_vulnerable`, it achieved 0.920 Precision@1 and 0.929 MRR. These results indicate that classical public-key cryptographic functions are usually retrieved near other migration-relevant functions. PQC KEM and PQC signature functions also achieved strong retrieval results, while the non-cryptographic category showed encouraging behavior on a smaller reviewed sample.

The results also demonstrated useful cross-library behavior. Queries from one cryptographic library retrieved related implementations from other libraries, such as RSA and ECDSA routines across OpenSSL, wolfSSL, mbedTLS, and nettle. For post-quantum cryptography, ML-KEM routines retrieved related ML-KEM

functions across different liboqs builds and optimization settings. This suggests that the model learned more than exact symbol matching: it captured useful similarity from decompiled code.

Additional robustness experiments clarified the role of symbol names and architecture. Removing function-name cues reduced performance, especially for classical public-key cryptography, but training on name-normalized data partially mitigated this drop. Increasing the token length to 1024 did not improve the no-name setting, suggesting that the remaining errors are not mainly caused by insufficient context length. A zero-shot AArch64 no-name evaluation showed encouraging transfer for the main target category, with 0.889 Precision@1 for classical PQ-vulnerable functions, although the AArch64 subset was not balanced enough to support broad conclusions for all categories.

In this work, a function is considered post-quantum vulnerable when it implements or supports a classical public-key algorithm that would require migration in a post-quantum setting. This is different from detecting a conventional software flaw. The tool provides evidence and prioritization: it helps analysts find functions likely to be relevant to post-quantum migration, but final interpretation still requires expert review.

Several limitations remain. The dataset and labels rely heavily on available symbols and trusted canonical identifiers. Although inference can be performed on unlabeled functions, the quality of the result depends on the labeled reference corpus used for comparison. The manually reviewed evaluation set is also smaller than the full silver-label dataset, and the current category taxonomy is coarse. Future work should expand the manually reviewed benchmark, evaluate fully stripped and obfuscated binaries, improve robustness when symbol names are unavailable, and add finer-grained labels for algorithm families and function roles.

Overall, the work shows that decompiler-based embeddings can support practical post-quantum cryptographic discovery in binaries. By combining Ghidra pseudocode extraction, supervised contrastive training, and nearest-neighbor retrieval, the system provides a scalable way to identify and prioritize cryptographic functions that may require attention during post-quantum migration.

Appendix A

Dataset Construction Details

This appendix provides additional implementation details about the construction of the binary and pseudo-code datasets used in this work. The main chapters describe the dataset at a higher level; here, the focus is on how the binaries were acquired, compiled, exported, and filtered.

A.1 Binary Sources

The dataset was built from two main sources. The first source was a BinAres-style [21] third-party library corpus organized under the `third_lib` directory. This provided multiple versions of common libraries, including OpenSSL and several non-cryptographic libraries used as negative examples. The second source was an expanded set of cryptographic libraries acquired or built specifically for this work, including libgcrypt, liboqs, mbedTLS, nettle, libsodium, and wolfSSL.

Non-cryptographic libraries such as zlib, sqlite, cJSON, libpng, libxml2, and libexpat were included to provide negative examples. These libraries are useful because they contain complex real-world C code, but they are not expected to implement public-key cryptography relevant to post-quantum migration.

A.2 Distribution Packages

Some binaries were obtained from Linux distribution packages. This was done to collect realistic library builds as they would appear on deployed systems. For example, Debian, Ubuntu, and Fedora packages were used for several cryptographic libraries. These binaries preserve distribution-default build settings, including the compiler, optimization flags, hardening flags, and packaging decisions chosen by the maintainers.

The advantage of distribution packages is realism: they represent binaries that analysts may encounter in practice. The limitation is that their exact compiler configuration is not controlled by this work. Therefore, distribution packages were complemented with source-built binaries where compiler and optimization settings could be explicitly varied.

A.3 Controlled Source Builds

To reduce dependence on a single compiler or optimization setting, additional binaries were compiled from upstream source code. The build script generated shared-library ELF binaries under paths of the form:

```
third_lib/<library>/source-<version>-<arch>-<compiler>-<opt>/install/
```

For the main source-options dataset, the controlled builds targeted Linux x86-64. Libraries were compiled with GCC and Clang using the optimization levels:

```
-O0, -O1, -O2, -O3, -Os
```

The source builds also used position-independent code for shared-library generation. This produced multiple binary variants for the same source-level library release, allowing the model to observe functions under different compiler and optimization conditions.

A.4 AArch64 Subset

In addition to the x86-64 corpus, a smaller AArch64 subset was generated for a cross-ISA robustness experiment. This subset was built from selected cryptographic libraries using GCC and the optimization levels:

```
-O0, -O2, -O3, -Os
```

The AArch64 binaries were not used for training the main embedding model. They were exported separately and used only for zero-shot inference evaluation. This allowed the evaluation to test whether a model trained on the main x86-64 corpus could still retrieve relevant functions from AArch64 decompiler output.

A.5 Ghidra Export

Each binary was analyzed using Ghidra's headless analyzer. A custom export script generated one JSON record per discovered function. Each record included metadata such as the function address, function name, size, basic-block count, opcode hash, and decompiled pseudo-code. The pseudo-code was the primary input to the embedding model.

The export process also filtered unsuitable binary inputs. Some files in the raw third-party library corpus were not valid ELF binaries, including zero-byte artifacts. Such files were skipped before analysis. Valid binaries could still produce noisy decompiler output, especially for tiny functions, runtime stubs, or compiler-generated helpers, so additional filtering was applied during dataset construction.

A.6 Function Filtering

Before pair construction, functions were filtered to remove low-information examples. The main filters removed functions with no decompiled pseudo-code, functions below the minimum size threshold, functions with fewer than two basic blocks, runtime stubs, and excessive duplicate opcode hashes. These filters reduced noise from trivial wrappers and compiler-generated routines.

For the source-options pair dataset, the final pair construction stage retained 47,598 functions after filtering and generated 16,000 training pairs and 4,000 evaluation pairs. The pair labels were created from trusted canonical function identifiers. Functions sharing the same canonical identifier formed positive pairs, while functions with different canonical identifiers formed negative pairs.

A.7 Manual Review

Automatic labels were used to scale dataset construction, but they are not perfect. Symbol names such as RSA, DH, ECDSA, ML-KEM, or SPHINCS are strong signals, but some matching functions are wrappers, lifecycle helpers, or generic encoding routines rather than core implementations. For this reason, a manual-review benchmark was created from the rule-based labels.

During manual review, each sampled function retained its seed category and was assigned a reviewed category. Ambiguous helper functions could be marked as `skip` and excluded from the strict manual benchmark. The final manual-review evaluation used 1,000 queries and provided a higher-confidence estimate of retrieval quality than the broader silver-label benchmark.

Bibliography

[1] PQMap, *Pqmap's github repository*. [Online]. Available: https://github.com/viniciusvianadp/pqmap.

[2] D. Conde-Torres, A. Seco-González, Piñeiro, and R. Garcia-Fandiño, "Mapping the quantum computing landscape: Growth, collaboration, and thematic convergence," en, *EPJ Quantum Technology*, vol. 13, no. 1, p. 7, Dec. 2026, ISSN: 2662-4400, 2196-0763. DOI: 10.1140/epjqt/s40507-026-00464-4. Accessed: Feb. 9, 2026. [Online]. Available: https://link.springer.com/10.1140/epjqt/s40507-026-00464-4.

[3] D. Castelvecchi, "IBM releases first-ever 1,000-qubit quantum chip," en, *Nature*, vol. 624, no. 7991, pp. 238–238, Dec. 2023, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/d41586-023-03854-1. Accessed: Feb. 9, 2026. [Online]. Available: https://www.nature.com/articles/d41586-023-03854-1.

[4] M. AbuGhanem, "IBM Quantum Computers: Evolution, Performance, and Future Directions," *The Journal of Supercomputing*, vol. 81, no. 5, p. 687, Apr. 2025, arXiv:2410.00916 [quant-ph], ISSN: 1573-0484. DOI: 10.1007/s11227-025-07047-7. Accessed: Feb. 9, 2026. [Online]. Available: http://arxiv.org/abs/2410.00916.

[5] Microsoft Azure Quantum et al., "Interferometric single-shot parity measurement in InAs–Al hybrid devices," en, *Nature*, vol. 638, no. 8051, pp. 651–655, Feb. 2025, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-024-08445-2. Accessed: Feb. 9, 2026. [Online]. Available: https://www.nature.com/articles/s41586-024-08445-2.

[6] C. Nayak, S. H. Simon, A. Stern, M. Freedman, and S. Das Sarma, "Non-Abelian anyons and topological quantum computation," en, *Reviews of Modern Physics*, vol. 80, no. 3, pp. 1083–1159, Sep. 2008, ISSN: 0034-6861, 1539-0756. DOI: 10.1103/RevModPhys.80.1083. Accessed: Feb. 9, 2026. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.80.1083.

[7] *What is Quantum Computing? - ibm*, [Accessed 09-02-2026]. [Online]. Available: https://www.ibm.com/think/topics/quantum-computing.

[8] C. Gidney and M. Ekerå, "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits," en, *Quantum*, vol. 5, p. 433, Apr. 2021, arXiv:1905.09749 [quant-ph], ISSN: 2521-327X. DOI: 10.22331/q-2021-04-15-433. Accessed: Feb. 9, 2026. [Online]. Available: http://arxiv.org/abs/1905.09749.

[9] W. Newhouse et al., "Migration to Post-Quantum Cryptography Quantum Readiness: Cryptographic Discovery," en,

[10] W. Newhouse, "Mappings of PQC Migration Project Capabilities to Risk Framework Documents," en, National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST CSWP 48 ipd, 2025, NIST CSWP 48 ipd. DOI: 10.6028/NIST.CSWP.48.ipd. Accessed: Oct. 19, 2025. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.48.ipd.pdf.

[11] *Ghidra software reverse engineering framework*, [Accessed 16-02-2026]. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra.

[12] National Institute of Standards and Technology, *Module-lattice-based key-encapsulation mechanism standard*. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.203.

[13] National Institute of Standards and Technology, *Module-lattice-based digital signature standard*. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.204.

[14] National Institute of Standards and Technology, *Stateless hash-based digital signature standard*. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.205.

[15] K. F. Hasan et al., "A Framework for Migrating to Post-Quantum Cryptography: Security Dependency Analysis and Case Studies," *IEEE Access*, vol. 12, pp. 23 427–23 450, 2024, arXiv:2307.06520 [cs], ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3360412. Accessed: Oct. 19, 2025. [Online]. Available: http://arxiv.org/abs/2307.06520.

[16] L. Zhang, *Toward Quantum-Safe Software Engineering: A Vision for Post-Quantum Cryptography Migration*, arXiv:2602.05759 [cs], Feb. 2026. DOI: 10.1145/3774748.3795653. Accessed: Mar. 2, 2026. [Online]. Available: http://arxiv.org/abs/2602.05759.

[17] C. Zhao, F. Kang, J. Yang, and H. Shu, "A review of cryptographic algorithm recognition technology for binary code," en, *Journal of Physics: Conference Series*, vol. 1856, no. 1, p. 012 015, Apr. 2021, ISSN: 1742-6596. DOI: 10.1088/1742-6596/1856/1/012015. Accessed: Nov. 3, 2025. [Online]. Available: https://doi.org/10.1088/1742-6596/1856/1/012015.

[18] Y. Fan, P. Biswas, and C. Garman, "R+R: A Systematic Study of Cryptographic Function Identification Approaches in Binaries," in *2024 Annual Computer Security Applications Conference (ACSAC)*, ISSN: 2576-9103, Dec. 2024, pp. 1092–1108. DOI: 10.1109/ACSAC63791.2024.00089. Accessed: Mar. 2, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/10918033/.

[19] I. U. Haq and J. Caballero, *A Survey of Binary Code Similarity*, arXiv:1909.11424 [cs], Sep. 2019. DOI: 10.48550/arXiv.1909.11424. Accessed: Oct. 18, 2025. [Online]. Available: http://arxiv.org/abs/1909.11424.

[20] D. Kryvosheieva, S. Sturua, M. Günther, S. Martens, and H. Xiao, *Efficient Code Embeddings from Code Generation Models*, arXiv:2508.21290 [cs], Aug. 2025. DOI: 10.48550/arXiv.2508.21290. Accessed: Dec. 4, 2025. [Online]. Available: http://arxiv.org/abs/2508.21290.

[21] J. Shi et al., *A Large Scale Study of AI-based Binary Function Similarity Detection Techniques for Security Researchers and Practitioners*, arXiv:2511.01180 [cs], Nov. 2025. DOI: 10.48550/arXiv.2511.01180. Accessed: Nov. 17, 2025. [Online]. Available: http://arxiv.org/abs/2511.01180.

[22] *jinaai/jina-embeddings-v2-base-code, Hugging Face — huggingface.co*, [Accessed 16-01-2026]. [Online]. Available: https://huggingface.co/jinaai/jina-embeddings-v2-base-code.

[23] A. Aigner, "FALKE-MC: A Neural Network Based Approach to Locate Cryptographic Functions in Machine Code," en, in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, Hamburg Germany: ACM, Aug. 2018, pp. 1–8, ISBN: 978-1-4503-6448-5. DOI: 10.1145/3230833.3230858. Accessed: Nov. 3, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3230833.3230858.

[24] G. D. Hill and X. J. A. Bellekens, *Deep Learning Based Cryptographic Primitive Classification*, arXiv:1709.08385 [cs], Sep. 2017. DOI: 10.48550/arXiv.1709.08385. Accessed: Mar. 2, 2026. [Online]. Available: http://arxiv.org/abs/1709.08385.

[25] L. Jia, A. Zhou, P. Jia, L. Liu, Y. Wang, and L. Liu, "A Neural Network-Based Approach for Cryptographic Function Detection in Malware," *IEEE Access*, vol. 8, pp. 23 506–23 521, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2966860. Accessed: Nov. 3, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8960303/.

[26] X. Shang et al., "FoC: Figure Out the Cryptographic Functions in Stripped Binaries with LLMs," *ACM Trans. Softw. Eng. Methodol.*, vol. 35, no. 1, 15:1–15:38, Dec. 2025, ISSN: 1049-331X. DOI: 10.1145/3731449. Accessed: Mar. 2, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3731449.

[27] N. Rattanavipanon, J. Suaboot, and W. Werapun, "A Toolchain for Assisting Migration of Software Executables Towards Post-Quantum Cryptography," *IEEE Access*, vol. 13, pp. 4368–4380, 2025, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3524309. Accessed: Oct. 19, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10819261/.

[28] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan. 1990, ISSN: 1937-4194. DOI: 10.1109/52.43044. Accessed: Mar. 4, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/43044.

[29] G. Canfora and M. Di Penta, "New Frontiers of Reverse Engineering," in *Future of Software Engineering (FOSE '07)*, May 2007, pp. 326–341. DOI: 10.1109/FOSE.2007.15. Accessed: Mar. 4, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/4221630/.

[30] C.-Y. Chen, K.-Y. Tai, and S.-S. Chong, "Quality Evaluation of Structural Design in Software Reverse Engineering: A Focus On Cohesion," *IEEE Access*, vol. 9, pp. 109 569–109 583, 2021, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3102295. Accessed: Mar. 4, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/9505641/.

[31] C. T. Ungureanu and R. Tataru, "The legality of reverse engineering or how to legally decipher trade secrets," en, *SHS Web of Conferences*, vol. 177, C. Ungureanu and I. Costea, Eds., p. 02 001, 2023, ISSN: 2261-2424. DOI: 10.1051/shsconf/202317702001. Accessed: Mar. 4, 2026. [Online]. Available: https://www.shs-conferences.org/10.1051/shsconf/202317702001.

[32] D. Pérez, *Ghidra Software Reverse Engineering for Beginners: Analyze, identify, and avoid malicious code and potential threats in your networks and systems*, en. Packt Publishing Ltd, Jan. 2021, Google-Books-ID: LTM-PEAAAQBAJ, ISBN: 978-1-80020-184-2.

[33] National Security Agency, *SLEIGH: A Language for Processor Specification*, [Accessed 04-03-2026]. [Online]. Available: https://ghidra.re/ghidra_docs/languages/html/sleigh.html.

[34] N. Naus, F. Verbeek, D. Walker, and B. Ravindran, "A Formal Semantics for P-Code," en, in *Verified Software. Theories, Tools and Experiments.*, A. Lal and S. Tonetta, Eds., Cham: Springer International Publishing, 2023, pp. 111–128, ISBN: 978-3-031-25803-9. DOI: 10.1007/978-3-031-25803-9_7.

[35] S. J. Pan and Q. Yang, "A Survey on Transfer Learning," en, *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010, ISSN: 1041-4347. DOI: 10.1109/TKDE.2009.191. Accessed: Dec. 12, 2025. [Online]. Available: http://ieeexplore.ieee.org/document/5288526/.

[36] C. Lee et al., *NV-Embed: Improved Techniques for Training LLMs as Generalist Embedding Models*, arXiv:2405.17428 [cs], Feb. 2025. DOI: 10.48550/arXiv.2405.17428. Accessed: Dec. 8, 2025. [Online]. Available: http://arxiv.org/abs/2405.17428.

[37] National Security Agency, *Ghidra's api overview*, [Accessed 25-03-2026]. [Online]. Available: https://ghidra.re/ghidra_docs/api/index.html.

[38] X. Huang et al., "CoSENT: Consistent Sentence Embedding via Similarity Ranking," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 32, pp. 2800–2813, 2024, ISSN: 2329-9304. DOI: 10.1109/TASLP.2024.3402087. Accessed: Dec. 21, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10531646/.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**